

Eszterházy Károly Főiskola  
Matematikai és Informatikai Intézet

# Programozási technológiák

A program kódja állandóan változik

Kusper Gábor

Eger, 2015

## Tartalom

1.	Bevezetés.....	10
1.1.	A jegyzetben tárgyalt elvek .....	10
1.1.1.	A program kódja állandóan változik.....	10
1.1.2.	A szoftver is csak egy termék, olyan, mint egy doboz müzli .....	11
1.1.3.	A szoftverfejlesztésnek nincs Szent Grálja .....	12
1.1.4.	További elvek.....	12
1.2.	Ajánlott irodalom .....	14
2.	Rendszerszervezés.....	15
2.1.	Szoftverkrízis.....	15
2.2.	A szoftverfejlesztés életciklusa.....	18
2.2.1.	A felhasználókban új igény merül fel.....	19
2.2.2.	Az igények elemzése, meghatározása .....	20
2.2.3.	Rendszerjavaslat kidolgozása .....	22
2.2.4.	Rendszerspecifikáció .....	23
2.2.5.	Logikai és fizikai tervezés.....	24
2.2.6.	Implementáció.....	26
2.2.7.	Tesztelés .....	26
2.2.8.	Rendszerátadás és bevezetés.....	30
2.2.9.	Üzemeltetés és karbantartás.....	30
2.2.10.	Az szoftverfejlesztés életciklusának dokumentumai.....	31
2.3.	Módszertanok.....	32
2.3.1.	Strukturált módszertanok .....	33
2.3.2.	Vízesés modell .....	34
2.3.3.	SSADM .....	36
2.3.4.	V-modell .....	38
2.3.5.	Prototípusmodell.....	40
2.3.6.	Spirálmodell.....	43
2.3.7.	Iteratív és inkrementális módszertanok.....	45
2.3.8.	Rational Unified Process – RUP .....	48
2.3.9.	Gyors alkalmazásfejlesztés – RAD .....	49
2.3.10.	Agilis szoftverfejlesztés.....	51
2.3.11.	Extrém programozás .....	53
2.3.12.	Scrum.....	56

2.4.	Kockázatmenedzsment .....	60
2.4.1.	COBIT .....	61
2.4.2.	Informatikai biztonsági módszertani kézikönyv ITB ajánlás .....	68
2.4.3.	Common Criteria .....	73
3.	Programozási technológiák – Tervezési alapelvek .....	76
3.1.	Objektumorientált programozás – OOP.....	76
3.1.1.	Bevezetés.....	76
3.1.2.	Egységbézárás – Encapsulation .....	78
3.1.3.	Öröklődés – Inheritance .....	78
3.1.4.	Többalakúság – Polymorphism.....	78
3.2.	Az OOP hasznos megoldásai.....	79
3.2.1.	Automatikus szemétgyűjtés .....	79
3.2.2.	A mező mint lokális-globális változó .....	79
3.2.3.	Többalakúság használata osztály behelyettesítésre .....	80
3.2.4.	Csatoltság csökkentése objektum-összetételel.....	81
3.3.	Az objektumorientált tervezés alapelvei.....	84
3.3.1.	A GOF könyv 1. alapelve – GOF1 .....	85
3.3.2.	A GOF könyv 2. alapelve – GOF2 .....	87
3.3.3.	Egy felelősség - egy osztály alapelve – SRP (Single Responsibility Principle) .....	92
3.3.4.	Nyitva-zárt alapelv – OCP (Open-Closed Principle) .....	93
3.3.5.	Liskov-féle behelyettesítési alapelv – LSP (Liskov Substitutional Principle).....	94
3.3.6.	Szerződésalapú programozás – Design by Contract.....	96
3.3.7.	Interfészszegregációs-alapelv – ISP (Interface Segregation Principle) .....	98
3.3.8.	Függőség megfordításának alapelve – DIP (Dependency Inversion Principle) .....	99
3.3.9.	További tervezési alapelvek .....	100
4.	Programozási technológiák – Tervezési minták .....	102
4.1.	Architekturális minták .....	102
4.1.1.	MVC – Model-View-Controller .....	102
4.1.2.	ASP.NET MVC Framework .....	104
4.1.3.	Többrétegű architektúra .....	107
4.2.	Létrehozási tervezési minták.....	109
4.2.1.	Egyke – Singleton.....	109
4.2.2.	Prototípus – Prototype .....	113
4.2.3.	Gyártómetódus – Factory Method.....	118

4.2.4.	Absztrakt gyár – Absztrakt Factory.....	121
4.3.	Szerkezeti tervezési minták .....	123
4.3.1.	Illesztő – Adapter.....	123
4.3.2.	Díszítő – Decorator .....	124
4.3.3.	Helyettes – Proxy.....	129
4.4.	Viselkedési tervezési minták .....	133
4.4.1.	Állapot – State .....	133
4.4.2.	Megfigyelő – Observer .....	137
4.4.3.	Sablonmetódus – Template Method.....	142
4.4.4.	Stratégia – Strategy .....	145
4.4.5.	Látogató – Visitor .....	150
5.	Programozási technológiák – Technológiák .....	153
5.1.	Tiszta kód.....	153
5.1.1.	Cserkészszabály .....	153
5.1.2.	Rothadó kód .....	154
5.2.	Egységeszt .....	155
5.2.1.	Egységeszt készítése Visual Studio 2013 esetén .....	155
5.2.2.	Egységeszt készítése Visual Studio 2008 és 2010 esetén .....	156
5.3.	Tesztvezérelt programozás – Test Driven Development – TDD .....	156
5.3.1.	TDD és a tiszta kód .....	157
5.3.2.	Piros – Zöld – Piros .....	157
5.3.3.	TDD esettanulmány .....	158
5.3.4.	Piros – Zöld – Kék – Piros.....	159
5.3.5.	TDD a szoftverfejlesztés kettős könyvelése .....	160
5.4.	Naplázás .....	160
5.5.	Aspektusorientált programozás .....	162
6.	Rendszerfejlesztés technológiája .....	165
6.1.	Követelményspecifikáció .....	165
6.1.1.	Szabad riport .....	165
6.1.2.	Irányított riport.....	166
6.1.3.	Követelménylista .....	166
6.1.4.	Félreértések elkerülését szolgáló módszerek.....	167
6.1.5.	Üzleti modellezés fogalmai.....	169
6.1.6.	Üzleti folyamatok modellezése .....	171

6.2.	Funkcionális specifikáció .....	174
6.2.1.	Követelményelemzés.....	174
6.2.2.	Használati esetek.....	175
6.2.3.	Képernyőtervezek.....	178
6.2.4.	Forgatókönyvek.....	179
6.2.5.	Olvasmányos dokumentum készítése .....	179
6.2.6.	Három olvasmányos példa .....	180
6.3.	Ütemterv .....	180
6.3.1.	Napidíj.....	181
6.3.2.	COCOMO modell .....	183
6.4.	Árajánlat .....	184
6.4.1.	Projektütemezés.....	186
6.4.2.	Kritikus út.....	187
6.5.	Megvalósíthatósági tanulmány .....	188
6.6.	A rendszerterv fajtái .....	189
6.7.	A rendszerterv elkészítése.....	191
6.7.1.	Határosztály-tervezés .....	193
6.7.2.	Menühierarchia-tervezés .....	193
6.7.3.	Storyboard .....	194
6.7.4.	Logikai rendszerterv .....	195
6.7.5.	Fizikai rendszerterv.....	196
6.8.	A tesztelés elmélete .....	196
6.8.1.	Tesztterv .....	198
6.8.2.	Tesztmodell, -eset, -eljárás.....	198
6.8.3.	Tesztszkriptek .....	198
6.8.4.	Adattesztelés .....	199
6.9.	Statikus tesztelési technikák.....	199
6.9.1.	Felülvizsgálat – Bevezetés .....	199
6.9.2.	Felülvizsgálat – Informális felülvizsgálat.....	200
6.9.3.	Felülvizsgálat – Átvizsgálás .....	200
6.9.4.	Felülvizsgálat – Technikai felülvizsgálat .....	202
6.9.5.	Felülvizsgálat – Inspekción .....	203
6.9.6.	Statikus elemzés - Bevezetés.....	204

6.9.7.	Statikus elemzés csak a forráskód alapján .....	204
6.9.8.	Statikus elemzés a forráskód és modell alapján.....	206
6.10.	Egyéb fogalmak .....	210
6.10.1.	DoD – Definition of Done .....	210
6.10.2.	POCOK .....	210
6.10.3.	Feature freeze .....	211
6.10.4.	Code freeze.....	211
7.	Rendszerfejlesztés technológiája - Eszközök.....	212
7.1.	Verziókövetés .....	212
7.1.1.	Közös szókincs .....	213
7.1.2.	Subversion .....	214
7.2.	Hibakövető rendszerek.....	223
7.2.1.	Bugzilla.....	223
7.2.2.	Mantis.....	225
7.3.	Modellező eszközök .....	230
7.3.1.	Enterprise Architect.....	230
7.3.2.	StarUML.....	236
8.	Összefoglalás .....	238
9.	Melléklet.....	242
9.1.	Jegyzőkönyvsablon .....	242
9.2.	Példakérőívek irányított riportohoz .....	242
9.2.1.	Példa 1.....	242
9.2.2.	Példa 2.....	243
9.2.3.	Példa 3., riportozó alrendszer kialakítása.....	244
9.3.	Követelménylista-sablon .....	246
9.4.	Példaütemterv.....	246
9.5.	Tesztpéldák.....	247
9.5.1.	Funkcionális tesztpélda .....	247
9.5.2.	Teszttervpélda .....	248
9.5.3.	Funkciótesztjegyzőkönyv-minta .....	251
9.5.4.	Teljesítményteszt jegyzőkönyvmintája .....	253
9.6.	Kulcsfogalmak.....	254



## Ábrajegyzék

1. ábra: Az ügyek szétválasztása elvet egy lángoló kardként képzelhetjük el.....	11
2. ábra: A szoftverfejlesztés életciklusa .....	19
3. ábra: A vízesés modell .....	35
4. ábra: Az SSADM felépítése .....	37
5. ábra: A V-modell.....	39
6. ábra: A prototípusmodell .....	41
7. ábra: A spirálmodell .....	44
8. ábra: Az inkrementális fejlesztés.....	45
9. ábra: Az inkrementumok.....	46
10. ábra: A RUP módszertan fázisai.....	49
11. ábra: A RAD módszertan .....	51
12. ábra: Az XP jól bevált módszerei .....	55
13. ábra: A SCRUM módszertan .....	57
14. ábra: Az MVC modell .....	103
15. ábra: Az MVVM modell .....	104
16. ábra: Az MVP modell .....	105
17. ábra: A háromrétegű szoftver architektúra.....	109
18. ábra: Példa UML ábra az Egyke tervezési minta .....	111
19. ábra: Példa UML ábra a prototípus tervezési mintára .....	117
20. ábra: Példa UML ábra a gyártómetódus tervezési mintára .....	120
21. ábra: Példa UML ábra a díszítő tervezési mintára.....	127
22. ábra: Példa UML ábra a helyettes tervezési mintára .....	131
23. ábra: Példa UML ábra az állapot tervezési mintára .....	136
24. ábra: Példa UML ábra a megfigyelő tervezési mintára .....	140
25. ábra: Példa UML ábra a stratégia tervezési mintára .....	149
26. ábra: Egy üzleti szereplő UML jelölése .....	170
27. ábra: Egy üzleti entitás UML jelölése .....	170
28. ábra: „Aktor vagy sem” folyamatábra.....	176
29. ábra: Egy függőségi gráf .....	187
30. ábra: Egy UML storyboard példa.....	195
31. ábra: Egy verzió fa .....	213
32. ábra: Beküldés TortoiseSVN segítségével .....	216
33. ábra: Hozzáadás TortoiseSVN segítségével .....	217
34. ábra: Összehasonlítás TortoiseSVN segítségével .....	218
35. ábra: Beküldés összefoglaló TortoiseSVN segítségével.....	218

36. ábra: Példa alkönyvtár szerkezete.....	219
37. ábra: Példa alkönyvtár szerkezete másolás után .....	221
38. ábra: Változási lista TortoiseSVN segítségével .....	222
39. ábra: A Bugzilla állapotgépe .....	224
40. ábra: A Mantis állapotgépe .....	226
41. ábra: Egy hibabejelentő ablak a Mantis programból .....	228
42. ábra: Bejelentett hibák ablak a Mantis programból .....	229
43. ábra: Bejelentett hiba adatai ablak a Mantis programból .....	229
44. ábra: Hibatörténet ablak a Mantis programból .....	230
45. ábra: Egy példa UML használatieset-diagram .....	231
46. ábra: Egy példa UML aktivitási diagram partíciókkal .....	232
47. ábra: Egy pelda UML szekvenciadiagram .....	233
48. ábra: Egy példa UML állapotgép diagram .....	234
49. ábra: Egy bejelentkező oldal UML állapotgépe .....	235
50. ábra: Egy példa UML osztálydiagram .....	236

## Köszönetnyilvánítás

Ennek a jegyzetnek az elkészítésében nagy segítséget nyújtott több tanítványom. Kiemelkedően sokat segített Csirmaz Péter, Dubovszki Martin, Fazekas Judit, Liktor János, Somogyi István, Szugyiczki Csaba, Takács Péter, Varga Balázs. Külön köszönet Tajti Tibornak és Márien Szabolcsnak, akik sok valós projekt tapasztalatát tárták elém.

Külön köszönetet szeretnék mondani feleségemnek, Juditnak, és gyermekinek, Lilinék és Péternek, akik nagy megértéssel fogadták a munkámból adódó kompromisszumokat. Köszönöm szüleimnek, hogy ők is messzemenőkig támogattak ebben a hatalmas munkában.

Külön köszönet Péter fiamnak, aki kedvemért rajzolt egy lángoló gyémántkardot az ügyek szétválasztása alapelvehez, ami a legnagyobb fegyver a kezünkben a szoftverkrízis ellen.

## 1. Bevezetés

Ez a jegyzet három tantárgy anyagát öleli fel kisebb-nagyobb részletességgel, ezek a következők:

- rendszerszervezés,
- programozási technológiák,
- rendszerfejlesztés technológiája.

A jegyzet áttekinti a három tárgy elméletét néhány gyakorlati példával alátámasztva.

Így, hogy három összefügő tantárgy anyaga került egy jegyzetbe, így mélyebb összefüggéseket fedezhet fel az olvasó, amellyel reméljük, hogy sok időt takarítunk meg számukra.

A jegyzetet elsősorban programterevő informatikus, mérnökinformatikus és gazdaságinformatikus hallgatóknak ajánljuk, illetve olyan gyakorló programozóknak, akik szeretnék megérteni, hogy a gyakorlatban működő módszerek milyen elméleti alapokon nyugszanak. A hallgatók számára sok esetben az egyes elemek csak néhány év gyakorlat után kerülnek a helyükre, hiszen ez a jegyzet az összefüggésekkel szól. Még akkor is, ha látszólag szoftverfejlesztési módszertanok, tervezési alapelvek és tervezési minták felsorolásának tűnik ez a jegyzet. Ez csak az első rétege az anyagnak. A valódi tudás az ezek közt lévő összefüggések felfedezése, amelyek alapelvekként jelennek.

Így érthető, hogy a jegyzet alapelvek köré szerveződik, amik nagyon magas szintű tanácsokat adnak a programozáshoz, a szoftverfejlesztéshez, illetve a rendszerszervezéséhez.

Ez a jegyzet két korábbi jegyzet, a „Jegyzet a projektlabor című tárgyhoz”, és a „Szoftvertesztelés” című jegyzetek a szerző által írt részeinek kibővített, néhány helyen rövidített, sok helyen javított és aktualizált változata.

### 1.1. A jegyzetben tárgyalt elvek

A jegyzetben tárgyalt három terület mindegyikének van egy-egy alapelve, amely segít tájékozódni a tárgyon belül. Ezek a következők:

- A program kódja állandóan változik.
- A szoftver is csak egy termék, olyan, mint egy doboz müzli.
- A szoftverfejlesztésnek nincs Szent Grálja.

#### 1.1.1. A program kódja állandóan változik

Az első elv a programozási technológiák alapelve. Kimondja, hogy a programunk forráskódja előbb vagy utóbb megváltozik. Ennek több oka lehet. Néhány lehetséges ok:

- Új igény merül fel.
- Valamely igény változik.
- A program környezete változik.
- A program valamely részét optimalizálni kell.
- A programban hibát találunk.
- A programot szépítjük (idegen szóval refaktoráljuk).

Ez a fő elvünk, amiből közvetve, vagy közvetlenül minden más elv következik. Ezért választottuk ezt az elvet a jegyzet alcímének.

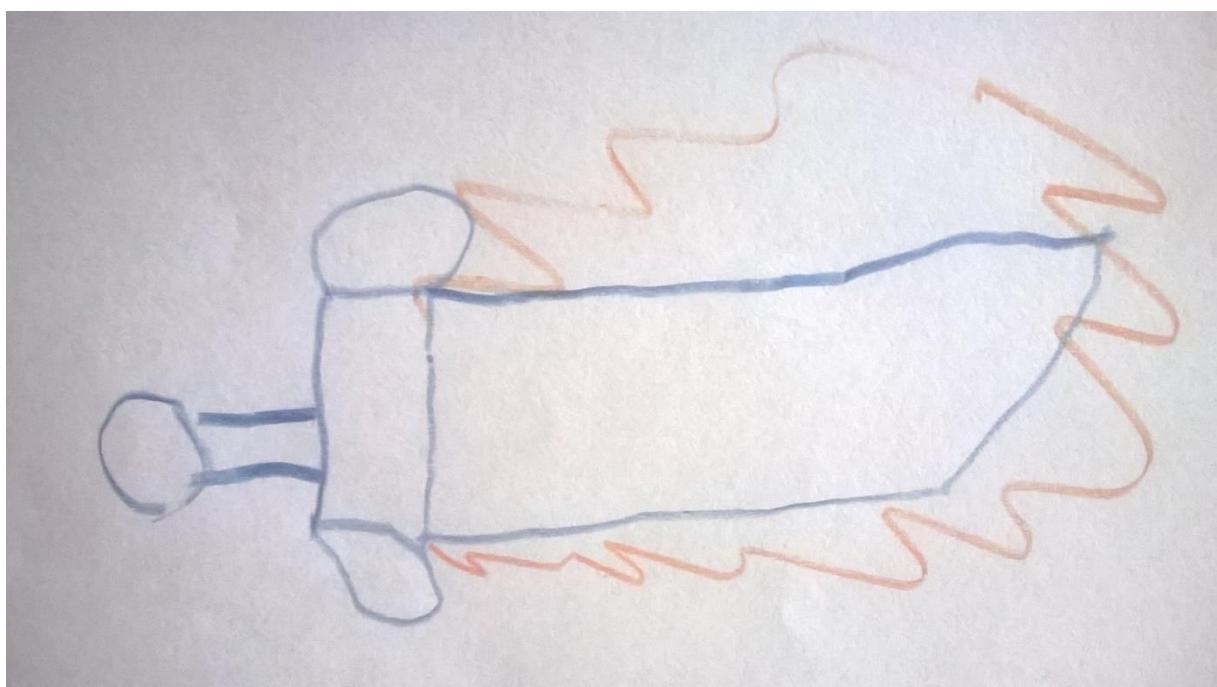
#### 1.1.1.1. Ügyek szétválasztása – Separation of Concerns

Az előző elv írja le, hogy mi a legnagyobb probléma a szoftverfejlesztés során, amivel minden programozónak szembe kell néznie: A program kódja állandóan változik! Szerencsére nem csak a problémát, hanem a probléma ellenszerét is ismerjük, ami szintén egy elv:

- Amit szét lehet választani, azt érdemes szétválasztani.

Ezt az elvet a szakirodalom az ügyek szétválasztása (angolul: Separation of Concerns) néven ismeri. Nagyon magas szintű, sok más elvben, tervezési mintában visszaköszön. Alapötlete az, hogy ha szétválasztjuk a szétválasztható dolgokat, akkor az így kapott kisebb részeket könnyebb kezelní.

Az ügyek szétválasztása a mi nagy, lángoló fegyverünk, ami néha kard, néha alabárd, néha láncfűrész, de a lényege, hogy szétválaszt, szétvág. Ehhez az elvhez érdemes visszatérni akkor, amikor már sikerült elmélyedni a tervezési elvekben, illetve a tervezési mintákban, és végiggondolni, hogy hol is bújik meg: mit választottunk el mitől és ez hogyan teszi rugalmasabbá a forráskódunkat.



1. ábra: Az ügyek szétválasztása elvet egy lángoló kardként képzelhetjük el

#### 1.1.2. A szoftver is csak egy termék, olyan, mint egy doboz müzli

A második elv a rendszerfejlesztés technológiájának alapelve. Kimondja, hogy a szoftver is csak egy termék, és mint ilyen, hasonló jellemzőkkel bír, mint egy doboz müzli. Ezen tulajdonságok közül néhány fontosabb:

- A szoftvernek van ára, gyártója, forgalmazója, vevője.
- A szoftvernek van minősége, doboza, designja, dokumentációja.

- A szoftverre vonatkoznak törvényi előírások, szabványok és ajánlások.

#### [1.1.3. A szoftverfejlesztésnek nincs Szent Grálya](#)

A harmadik elv a rendszerszervezés alapelve. Eredeti angol megfogalmazása: „There is no Silver Bullet”. Kimondja, hogy a szoftverfejlesztés területén nincs olyan kitüntetett módszertan, technológia vagy szemléletmód, amely egymaga megoldaná a minőségi és egyben olcsó szoftverfejlesztés gondját (lásd még a szoftverkritízisről szóló részeket). Ugyanakkor a modern módszertanok kombinációja nagyságrendekkel jobb és olcsóbb szoftverfejlesztést tesz lehetővé.

#### [1.1.4. További elvek](#)

A jegyzet további elveket is tartalmaz, ezek:

- az objektumorientált programozás alapelvei,
- az objektumorientált tervezés alapelvei.

Az objektum orientált programozás alapelvei:

- Egységbezárás (angolul: encapsulation)
- Öröklődés (angolul: inheritance)
- Többletakúsgág (angolul: polymorphism)

Ebben a könyvben a debreceni és egri programozó képzés hagyományait követve a fenti 3 objektum orientált programozási alapvetőt használjuk. Ezeket részletesen kifejtjük a későbbi fejezetekben.

Ugyanakkor meg kell jegyezni, hogy az angol nyelvű irodalom általában 4 objektum orientált programozás alapvetőt használ. A fenti hármat kiegészítik az absztrakció elvével. Illetve sok forrás az objektum orientált programozás alapelveihez sorolja az objektum-orientált programozás alapvető technikai megoldásait is, mint az osztály és az objektumok használatát, a késői kötést, a név túlterhelést, illetve a metódusok felülírását.

A miskolci programozó képzés hagyományosan az információ rejtést (angolul: information hiding) külön objektum orientált programozási alapelveknek tekinti. Ebben a könyvben az információ rejtést az egységbezárás részeként értelmezzük.

Az objektumorientált tervezés alapelveinek köre nem teljesen letisztult terület. Néhány elvnek több, az irodalomban elterjedt neve is van. Az egyes alapelvegyüjtemények szűkebbek, mások bővebbek. A leginkább elfogadott alapelvek a következők:

- GOF1: Programozz felületre implementáció helyett.
- GOF2: Használj objektum-összetételt öröklés helyett, ha csak lehet.
- HP (angolul: Hollywood Principle): Ne hívj, majd mi hívunk.
- OCP (angolul: Open-Closed Principle): Az osztályok legyenek nyitottak a bővítésre, de zártak a módosításra.
- LSP (angolul: Liskov Substitutional Principle): Az alosztály legyen egyben altípus is.
- DIP (angolul: Dependency Inversion Principle): Absztrakciótól függj, ne függj konkrét osztályoktól.

Ezeket részletesen kifejtjük a későbbi fejezetekben.

A jegyzet megírását nagyban inspirált a „Joel on Software” című blog. Ennek elérhető két magyar fordítása is:

- <http://hungarian.joelonsoftware.com/>
- <http://www.js.hu/jos/>

Innen kerültek be a jegyzetbe a következő elvek:

- A szivárgó absztrakció elve.
- Joel teszt.
- Módosított 80/20 elv.

A szivárgó absztrakció elve kimondja, hogy minél magasabb absztrakciós szinten álló programozási nyelvet használ a programozó, annál nagyobb rálátással kell rendelkeznie a szoftverfejlesztésre, mert egy hiba adódhat abból, hogy egy alsóbb absztrakciós szintet nem tökéletesen fed le a felette lévő. Az ilyen hibák megértéséhez és javításához az alsóbb szintek ismerete szükséges.

A Joel teszt, habár nem elv, szintén ide került. Ez egy szoftverfejlesztő cég termelékenységi potenciáját hivatott tesztelni. Ismertetjük a tesztet, amely eredetiben a [http://js.hu/jos/joel\\_test](http://js.hu/jos/joel_test) oldalon található meg:

1. Használsz forráskövető rendszert?
2. Megy a fordítás egy lépésben?
3. Naponta fordítod a szoftvered?
4. Van hibakövető rendszered?
5. Új kód írása előtt kijavítod a hibákat?
6. Van jól karbantartott ütemezésed?
7. Van specifikációd?
8. A programozók nyugodt körülmények között dolgozhatnak?
9. Az elérhető legjobb segédeszközököt használjátok?
10. Vannak tesztelőid?
11. Íratsz kódot a felvételi elbeszélgetésen?
12. Végzel folyosói használhatósági tesztet?

Reméljük, hogy jegyzetünk segít jó eredmények elérésében a Joel-teszten!

A módosított 80/20 elv az eredeti 80/20 elv alkalmazása a szoftver termékre. Az eredeti elv szoftverek esetében kimondja, hogy a szoftver utolsó 20%-ának elkészítése (más értelmezésben a legkritikusabb 20% elkészítése) a fejlesztés idejének 80%-át emészti fel (más értelmezésben a hibakeresés az idő 80%-át emészti fel). A módosított 80/20 elv kimondja, hogy a felhasználók 80%-a a szoftvertermék funkcióinak csak 20%-át használja, de mindegyikük más-más 20%-át.

Ez az elv arról szól, hogy csak komplex, sok-sok funkcióval bíró szoftverek lehetnek sikeresek a piacon. Ez az elv ellentmondani látszik a Unix fejlesztési filozófiájának: „Olyan programokat készíts, ami csak

egy dolgot csinál, de azt jól csinálja”. Eredeti megfogalmazásában: „Write programs that do one thing and do it well.”

Ezen túl tárgyaljuk Fred Brooks megfigyelését, miszerint: új munkatárs felvétele egy késésben lévő szoftverprojekthez csak további késést okoz. Eredeti megfogalmazása: „Adding manpower to a late software project makes it later.”

## 1.2. Ajánlott irodalom

A jegyzet elkészítésénél az alábbi könyvek inspiráltak, így ezeket nyugodt szívvvel ajánljuk:

- Robert C. Martin: Túlélőkönyv programozóknak, Hogyan váljunk igazi szakemberré?, Kiskapu Kiadó Kft., 2011.
- Henrik Kniberg, Mattias Skarin: Kanban és Scrum, mindenből a legjobbat, Fordította: Csutorás Zoltán és Marhefka István, C4Media Inc., 2010.
- Robert C. Martin: Tiszta kód, Az agilis szoftverfejlesztés kézikönyve, Kiskapu Kiadó Kft., 2010.
- Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra: Head First Design Patterns, O'Reilly Media, 2004.
- Ian Sommerville: Szoftverrendszer fejlesztése, Panem Kiadó Kft., 2007.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Programtervezési minták, Kiskapu Kiadó Kft., 2004.
- Robert C. Martin: Agile Software Development, Principles, Patterns, and Practices, Pearson Education, 2002.

## **2. Rendszerszervezés**

A rendszerszervezés téma körök külön jegyzetet érdemlő téma kör. Itt csak a programozás-technológiához szorosan kapcsolódó részeit ismertetjük a rendszerszervezésnek.

A rendszerszervezés feladata, hogy az implementálás előtt jó alaposan gondoljuk végig a rendszert, amit tervezünk. Miért? Mert minél hamarabb derül ki egy tervezési hiba, annál olcsóbb azt javítani. Ez egy fontos felismerés, mert olcsón szeretnénk magas minőségű szoftvert készíteni. Ez nehéz feladat, hiszen a mai napig szoftverkrízisről beszélhetünk. A szoftverkrízisre adott egyik válasz a rendszerszervezés.

A rendszerszervezés egy körfolyamatot ír le, a szoftverfejlesztés életciklusát. Ez alatt sok termék (szoftver, dokumentáció) keletkezik, amelyek elkészítésére a módszertanok tesznek ajánlást. A legfontosabb dokumentum a rendszerterv. A korai módszertanok nagy hangsúlyt fektettek arra, hogyan kell részletes, előrettekintő rendszertervet írni. Ha egy fejlesztő nem értett valamit, akkor megnézte a rendszertervet. A mai módszertanok erre kevésbé figyelnek, inkább az iterációban és a fejlesztők közti kommunikációban bíznak. Ha egy fejlesztő nem ért valamit, megkérdezi a másik fejlesztőt.

A módszertanok a programfejlesztés divatjai. Ma az agilis módszertanok a divatosak, régebben az iteratív, azt megelőzően a strukturálisak voltak divatosak. Valószínűsíthető, hogy a módszertanok területén is lesz egyfajta ciklikusság, mint ahogy a divat területén is visszatér néha egy-egy őrület. Könnyen elképzelhető, hogy a jövő módszertanai ismét hangsúlyt fektetnek majd a rendszertervre.

A fejezet foglalkozik még a kockázatkezeléssel. Néhány tipikus informatikai rendszert érintő kockázaton keresztül mutatjuk be, hogyan kell a kockázatokat elfogadható szintre süllyeszteni.

### **2.1. Szoftverkrízis**

A rendszerszervezés fő problémája az úgynevezett szoftverkrízis (software crisis). A szoftverkrízis alatt azt értjük, hogy a szoftverprojektek jelentős része sikertelen. Sikertelen a következő értelemben:

- Vagy a tervezettnél drágábban készül el (over budget),
- Vagy a tervezetnél hosszabb idő alatt (over time),
- Vagy nem az igényeknek megfelelő,
- Vagy nagyon rossz minőségű / rossz hatásfokú / nehezen karbantartható,
- Vagy anyagi / környezeti / egészségügyi kárhoz vezet,
- Vagy átadásra sem kerül.

A szoftverkrízis egyidős a számítógépek elterjedésével. Mivel maga a hardver szoftver nélkül csak egy használhatatlan vas, ezért kezdetektől nagy az igény a felhasználóbarát, magas minőségű, olcsó szoftverek iránt. Ezt az igényt a szoftverfejlesztő ipar a mai napig képtelen kielégíteni.

A sikertelen szoftverprojektek száma csökken. A hetvenes évek 80-90%-os sikertelen projektaránya napjainkra azért jócskán 50% alá esett, de a mai napig elmondható, hogy minden harmadik szoftverprojekt sikertelen. Szerencsére a sikertelen szoftverprojektek sikertelenségének okai kevésbé súlyosak. Míg a hetvenes években a sikertelen projektek túlnyomó többsége átadásra sem került, addig manapság a sikertelenség oka inkább az idő vagy a pénzügyi keretek túllépése. Gyakori még,

hogy a megrendelő nem az igényeinek teljesen megfelelő szoftvert kapja, de ez inkább a megrendelő és a szoftvercégek elégtelen kommunikációjának tudható be.

A szoftverkriszisnek több ismert oka is van:

- Elégtelen hatékonyaság: A szoftvercégek nem elég hatékonyak, azaz adott idő alatt kevesebb jó minőségű kódot fejlesztenek, mint az elvárható lenne.
- Művészlelkű programozók: A programozók „programozóművésznek” tekintik magukat, akik a programozást öncélú megvalósítási formának tekintik, amiért jól fizetnek.
- Félreérzés: A szoftvercégek nem ismerik azt a szakterületet (angolul: domain), ahonnan a megrendelő jön és így nem értik szaknyelvét. Ez félreérzéseket szülhet.
- Gyorsan változó környezet / igények: Egy hosszú szoftverprojekt ideje alatt megváltozhat a megrendelő igénye. Ennek oka lehet például egy új jogszabály, azaz a program környezetének megváltozása.
- A fejlesztési idő nehezen becsülhető: A szoftverprojektek sikertelenségének legfőbb oka, hogy az előre kitűzött időpontra nem készül el a program. Ennek fő oka, hogy rendkívül sok váratlan nehézségbe ütközhet egy programozó („szívás” nélkül nem lehet programot fejleszteni), ami nehezen becsülhető.
- Kevéssé specifikált feladat: Gyakori probléma, hogy a specifikáció egyetlen oldalas. Sok követelményre csak a fejlesztés során derül fény.

Miután a kiváltó okokat többé-kevésbé megismertük, ismerjük meg a szoftverkriszisre adott főbb válaszokat is:

- A rendszerszervezés válasza a módszertanok bevezetése. A módszertanok szigorúan vagy kevésbé szigorúan, de előírják a szoftverfejlesztés lépéseiinek sorrendjét. Meghatározzák, mikor kell a megrendelőnek és a fejlesztőnek kommunikálnia, ezek alapján milyen dokumentumoknak kell létrejönniük. minden lépés néhány dokumentumra épül és általában egy új dokumentum vagy programrészlet az eredménye. A lépések a szoftverfejlesztés életciklusának a lépései. A módszertanokkal részletesen foglalkozunk a későbbiekbén.
- A rendszerszervezés másik válasza a kockázatmenedzsment. A kockázatmenedzsment kimondja, hogy a kockázatokat fel kell mérni, azokat a valószínűségük és okozott idő / pénz veszteségük szerint osztályozni és a legsúlyosabb kockázatokra készülni kell. Ez általában redundáns erőforrások biztosításával lehetséges.
- A rendszerszervezés következő válasza a megrendelő és a fejlesztő kommunikációját segítő vizuális nyelvek bevezetése, ezek egységesítése. Az UML, és főleg a használati esetek (angolul: use case) elterjedése egy olyan jelölésrendszert biztosít, amelyet a megrendelő szakemberei és a programozók is könnyen megértenek. Ez segíti a félreérzések elkerülését a két fél között.
- A programozási technológiák első válasza, hogy a programozási nyelvek fejlődésével egy utasítás egyre több gépi kódú utasításnak felel meg. Ez az arány assembler nyelveknél egy az egyhez (1:1), azaz egy assembler mnemonic egy gépi kódú utasításnak felel meg. A második generációs nyelvek (Fortran, COBOL, PL/1) esetén egy utasítás néhány tucat gépi kódú utasítást vált ki (1:10). A harmadik generációs procedurális nyelveknél (Pascal, Ada, C/C++) egy utasítás néhány száz gépi kódú utasításnak felelhet meg (1:100). A negyedik generációs OOP

nyelvek (Java, C#) esetén ez a szám néhány ezer is lehet (1:1000). Ezzel a módszerrel nő a programozók hatékonysága.

- A programozási technológiák második válasza a program modulokra bontása (lásd: Separation of Concerns). Már az assembler nyelvek is megengedték, hogy a forráskódot több állományba tároljuk, ezek között szubrutinokat hívjunk. minden állományt külön kellett fordítani (angolul: compile) tárgykódú programmá, ami már gépi kódú, de amiben a címzések még nem feloldottak. A tárgykódú programokat egy szerkesztő (angolul: linker) segítségével kellett futtatható programmá összeszerkeszteni. A modularitás a programozási nyelvek fejlődésével egyre nagyobb támogatást kapott. Megjelentek a függvények és az eljárások (együtt alprogramok), a modulok (fordítási alegységek), végül az osztályok, amik az adattagokat és a rajtuk végzett metódusokat zárja egységebe. A modularitás nagy áldása az, hogy megengedi, hogy több programozó készítse a programot. minden programozónak csak a saját modulját kell átlátnia, a többit nem. Ez azért fontos, mert egy programozó csak néhány ezer, esetleg tízezer kódsort lát át, azaz egy modul maximális mérete kb. 10 000 sor. A modularitás megengedi, hogy ennél nagyobb programot is fejleszthessünk azáltal, hogy azt kisebb, átlátható modulokra bontjuk. A moduloknak természetesen kommunikálniuk kell. Erre később térünk ki.
- A programozási technológiák fő válasza a tervezési alapelvek és a tervezési minták bevezetése. A tervezési alapelvek magas szintű jó tanácsok, hogy hogyan érdemes programot fejleszteni. A tervezési minták alacsonyabb szintű ajánlások, egy-egy gyakori problémára nyújtanak kiforrott, mégis általános megoldást. Jól megfigyelhető, hogy a tervezési minták követik a tervezési alapelveket.
- A programozási technológiák legújabb válasza a szakterület specifikus (angolul: domain specific) keretrendszerök, programozási nyelvek megjelenése, illetve olyan technológiák megjelenése, amelyekkel ezek könnyen elkészíthetők. A területspecifikus fejlesztés ígérete az, hogy egy konkrét területre specifikált nyelven a fejlesztés sokkal hatékonyabb. Gondoljunk például arra, milyen egyszerű CMS rendszerekkel webportált készíteni. A területspecifikus technológiák közül ezek a legismertebbek:
  - Modell vezérelt architektúra, vagy másik nevénmodel vezérelt programozás (angolul: Model Driven Architecture / Development, röviden: MDA),
  - Szakterület specifikus modellezés (angolul: Domain-Specific Modeling, röviden: DSM),
  - Szakterület vezérelt tervezés (angolul: Domain-Driven Design, röviden: DDD).
- A szoftverfejlesztés technológiájának első válasza a programozó munkájának segítése. Nagyon fontos lépés volt az editorok, nyomkövető (debugger) rendszerek integrálása egy integrált fejlesztési környezetbe (angolul: Integrated Development Environment, röviden: IDE), ami többek közt szintaxis-kiemeléssel (angolul: syntax highlight) segíti a programozó munkáját. Ide sorolható minden olyan szoftver, ami a programozók munkáját segíti.
- A szoftverfejlesztés technológiájának további válasza (ami az előző általánosítása) a csapatmunkát segítő technológiák kifejlesztése. Láttuk, hogy a program modulokra bontható, a modulokat más-más programozó készítheti. A fejlesztőknek sokat kell kommunikálniuk, hiszen továbbra is egy szoftvert fejlesztenek, a modulok függnek egymástól, a moduloknak hívníuk kell egymást. Ide tartozik minden olyan szoftver, ami több programozó együttes munkáját segíti. A fő csapatmunkát segítő technológiák:
  - Verziókövetés

- Hibakövetés
- Modellező eszközök
- Fordítássegítő „make” eszközök
- A szoftverfejlesztés technológiájának második válasza a tesztelés segítése, automatizálása. Itt nagy szerepet kap az egységtesztek (angolul: unit-tests) elterjedése, ami lehetővé tette az agilis módszertanok elterjedését. Ezen a területen belül külön kiemelendő a tesztvezérelt fejlesztés (Test-Driven Development, TDD), amely előírja, hogy először írunk egységtesztet, csak utána a tesztelendő metódust.

A válaszok közül többet mélységében ismertetünk a jegyzetben, ezért ezekre a fenti felsorolásban kevés helyet szenteltünk. Néhányat csak felszínesen ismertetünk a jegyzet további részeiben, ezért ezekre a fenti felsorolásban több helyet szenteltünk.

## 2.2. A szoftverfejlesztés életciklusa

A szoftverfejlesztés életciklusa (angolul: Software Development Life Cycle, röviden: SDLC) a szoftverrel egyidős fogalom, a szoftver életének állomásait írja le az igény megszületésétől az átadásig. Tipikus, hogy a felhasználók előbb vagy utóbb felmerülő új ötletei a szoftver továbbfejlesztését teszik szükségessé. Tehát egy szoftver soha sincs kész, ciklikusan meg-megújul. Ezt nevezzük szoftverfejlesztés életciklusának.

A szoftverfejlesztés életciklusának lépései a módszertanok határozzák meg. A módszertanok célja, hogy magas színvonalú szoftvert fejlesszünk minél kisebb költséggel. A módszertanokat később fejtjük ki. Itt egy általános szoftverfejlesztési életciklust tekintünk át.

A szoftverfejlesztés életciklusa (kerek zárójelben a legfontosabb elkészítendő termékek) [szöglletes zárójelben az angol elnevezés]:

- A felhasználókban új igény merül fel [user would like to have a new feature].
- Az igények, követelmények elemzése, meghatározása (követelményspecifikáció) [Requirement Analysis].
- Rendszerjavaslat kidolgozása (funkcionális specifikáció, ütemterv, szerződéskötés) [Functional Analysis].
- Rendszerspecifikáció (megvalósíthatósági tanulmány, nagyvonalú rendszerterv) [Analysis & Design].
- Logikai és fizikai tervezés (logikai- és fizikai rendszerterv) [Analysis & Design].
- Implementáció (szoftver) [Implementation].
- Tesztelés (teszter, teszesetek, tesztnapló, validált szoftver) [Testing & Validation].
- Rendszerátadás és bevezetés (felhasználói dokumentáció) [Delivery & Deployment].
- Üzemeltetés és karbantartás (rendszeres mentés) [Operating & Maintenance].
- A felhasználókban új igény merül fel [user would like to have a new feature].



2. ábra: A szoftverfejlesztés életciklusa

Látható, hogy az első lépés és az utolsó ugyanaz. Ez biztosítja a ciklikusságot. Elvileg egy hasznos szoftvernek végtelen az életciklusa. Gyakorlatilag a szoftver és futási környezete előregszik. Előbb-utóbb már nem lesz programozó, aki ismerné a programozási nyelvet, amin íródott (ilyen probléma van manapság a COBOL programokkal), a futtató operációs rendszerhez nincsenek frissítések, a meghibásodott hardverelemekeit nem lehet pótolni. Az ilyen IT rendszereket hívjuk kioregedett, hagyaték rendszernek (angolul: legacy system). Valahol itt van vége a szoftverfejlesztés életciklusának. Az életciklus egyes lépései részletesebben is kifejtjük. A lépésekkel megvalósító technikákat a Rendszerfejlesztés technológiája című fejezetben ismertetjük.

Érdekes megfigyelni, hogy két lépésnek, a „Rendszerspecifikáció”-nak és a „Logikai és fizikai tervezés”-nek ugyanaz az angol neve: „Analysis & Design”. Ez ezért van így, mert az első lépésben inkább az analízisen, a következőben inkább a tervezésen van a hangsúly, de mind a kettőben van ez is, az is. Ezért ezt egy lépésnek tekintik az angol nyelvű szakirodalomban.

#### 2.2.1. A felhasználókban új igény merül fel

Ez legelőször azt jelenti, hogy igény merül fel egy új szoftverre. Kétféle szoftverről beszélhetünk:

- egyedi szoftver,
- dobozos szoftver.

Az egyedi szoftver esetén ismert a megrendelő. Az ō számára készül a szoftver, a fejlesztőnek igyekeznie kell minden igényét kielégíteni. A megrendelő fizeti ki a fejlesztés árát. Dobozos szoftver esetén ilyen értelemben nincs megrendelő, reménybeli vásárlók vannak, akiket együtt célcsoportnak nevezünk. A célcsoportot tekinthetjük megrendelőnek, innentől fogva a kétféle fejlesztés ugyanaz. Egyetlen különbség a megrendelővel folytatott kommunikációban van. Egyedi szoftver esetén általában van egy, a megrendelő részéről kinevezett szakember, akitől kommunikálhatunk. Dobozos szoftver esetén kérdőívekkel érhetjük el a célcsoportot, így nehezebb a kommunikáció.

Általában minden szoftver egyedi szoftverként kezdi életciklusát. Ha sikeres a megoldás és sikerül több hasonló megrendelést szerezni, akkor kialakulhat egy termékvonal, amiből relatíve kis ráfordítással lehet dobozos szoftvert fejleszteni. Ebben az esetben nyílik meg az „aranybánya”: dobozos szoftver forgalmazása esetén a sokszorosítás és a terjesztés nagyon olcsó, míg maga a termék lehet viszonylag drága.

Minden szoftvercégnek törekednie kell termékvonal, illetve dobozos szoftver kifejlesztésére. Egyébként csak projektcég, jobb esetben tanácsadó cég marad, ami sokkal kisebb elérhető profitot jelent. A szoftvercégeknek ez a törekvése egyfajta válasznak is tekinthető a szoftverkrízisre, mert egy dobozos szoftver minden nagyságrendekkel olcsóbb, mint egy egyedi szoftver. Egy egyedi szoftvernél elfogadott a több millió forintos ár, míg dobozos szoftver esetén a néhány százezer forintos ár már drágának számít.

Az egyedi és a dobozos szoftver között beszélhetünk néhány átmenetről:

- egyedi szoftver,
- újból felhasznált osztályokat is tartalmazó egyedi szoftver,
- kész (angolul: off-the-shelf) komponenseket is tartalmazó egyedi szoftver,
- keretrendszer,
- keretrendszerrel fejlesztett egyedi szoftver,
- testre szabható szoftver,
- dobozos szoftver.

Az egyedi szoftverek egy különleges fajtája a keretrendszer. Keretrendszert azért fejlesztünk, hogy sokkal gyorsabban tudunk egyedi szoftvert fejleszteni. Általában ez csak akkor lehetséges, ha jól definiált feladatkörre (angolul: domain) specializáljuk a keretrendszert. Keretrendszer például minden CMS (angolul: Content Management System) rendszer.

## 2.2.2. [Az igények elemzése, meghatározása](#)

Tehát van egy megrendelőnk, aki szerencsés esetben a felhasználónk lesz. A felhasználókban a használat során újabb és újabb igények merülnek fel. Ezeket rögzíteni és elemezni kell. Az igények általában egy új funkció (feature) bevezetésére szólnak, de lehetnek nemfunkcionális kérések is, például legyen gyorsabb a keresés. Az igények felméréséhez riportot kell készítenünk a megrendelővel. Kétfajta riportot ismerünk:

- szabad riport,
- irányított riport.

Szabad riport alatt azt értjük, hogy a megrendelő saját szavaival mondja el, milyen rendszert szeretne. A kérdező, aki általában rendszerszerző, esetleg üzletelemző, ezt igyekszik szó szerint rögzíteni. Szabad riport esetén általában csak egy kérdést teszünk fel: Milyen elvárásokat támaszt a rendszerrel kapcsolatban; hogyan működjön a rendszer?

Tovább kérdéseket csak a következő esetekben teszünk fel:

- ha valamelyik fogalom nem világos,
- ha ellentmondást vélünk felfedezni, vagy
- ha gyanús, hogy nem minden részletre tért ki a megrendelő, azaz lyukas a történet.

Ha a kérdezőnek kezd összeállni a kép a fejében, hogy mit is szeretne a megrendelő, akkor további kérdéseket tesz fel irányított riportok formájában, hogy jól érti-e a megrendelőt.

Az irányított riport során egy előre megírt kérdőívet töltetünk ki a megrendelővel. Ez általában a szabad riportok során felmerült kérdések pontosítására vagy egy alrendszerre vonatkozó követelmények felderítésére szolgál.

A riportokkal két célunk van:

- a megrendelő üzleti folyamatainak felderítése,
- a megrendelő igényeinek (követelményeinek) felderítése.

Az üzleti folyamatok alatt azt értjük, hogy a megrendelő cége hogyan működik. A sok folyamat közül csak azokat kell megértenünk, amikkel a megrendelt program kapcsolatban áll. A kapcsolat többféle lehet. Az új rendszer egy vagy több folyamatot:

- kiválthat,
- módosíthat,
- adatforrásként használhat,
- adatokkal láthat el.

Ahhoz, hogy a megrendelő új „folyamatszövetét” megszöhhessük, érteni kell a régit. Ezen túl azt is tudnunk kell, hogy az új szövettel szemben mik az elvárások. Tehát, ahogy már írtuk, a riportokon keresztül meg kell értenünk a megrendelő üzleti folyamatait és követelményeit az új rendszerre vonatkozóan.

#### 2.2.2.1. [Követelményspecifikáció](#)

Ebben a részben készül el a követelményspecifikáció (requirement specification). A követelményspecifikáció alapját a megrendelővel készített riportok képezik. Főbb részei:

- Jelenlegi helyzet leírása.
- Vágylomrendszer leírása.
- A rendszerre vonatkozó pályázat, törvények, rendeletek, szabványok és ajánlások felsorolása.
- Jelenlegi üzleti folyamatok modellje.
- Igényelt üzleti folyamatok modellje.
- Követelménylista.

- Irányított és szabad szöveges riportok szövege.
- Fogalomszótár.

A fenti listából csak a követelménylista kötelező, de a többi segíti ennek megértését. Egy példa követelménylista található a mellékletben. Nagyon fontos megérteni, hogy nem minden követelmény kerül bele a program következő verziójába. Erről az ütemterv leírásában szólunk részletesen.

A követelménylistában funkcionális és nemfunkcionális követelmények vannak. A legtipikusabb nemfunkcionális követelmények ezek:

- helyesség,
- használhatóság,
- megbízhatóság,
- adaptálhatóság / hordozhatóság,
- karbantarthatóság,
- hatékonyság / magas teljesítmény,
- hibatűrés / robosztusság,
- bővíthetőség / flexibilitás,
- újrahasznosíthatóság,
- kompatibilitás,
- könnyen megvásárolható vagy letölthető.

### 2.2.3. Rendszerjavaslat kidolgozása

Ebben a részben dolgozzuk ki a funkcionális specifikációt a követelményspecifikáció alapján. A funkcionális specifikáció alapján készül az ütemterv és az árajánlat. Általában ennek a szakasznak a végén szerződünk. Ha nem sikerül a szerződés, akkor sok munkánk mehet kárba, de ez a szoftvercégek üzleti kockázata.

#### 2.2.3.1. Funkcionális specifikáció

A funkcionális specifikáció a felhasználó szemszögéből írja le a rendszert. A követelményelemzésből ismerjük az elkészítendő rendszer üzleti folyamatait. Ezeket kell átalakítanunk funkciókká, azaz menükké, gombokká, lenyíló listákká. A funkcionális specifikáció központi elemei a használati esetek (angolul: use case). A használati esetek olyan egyszerű ábrák, amelyet a megrendelő könnyen megért mindenféle informatikai előképzettség nélkül. A funkcionális specifikáció fontosabb részei:

- Jelenlegi helyzet leírása.
- Vágylomrendszer leírása.
- A rendszerre vonatkozó pályázat, törvények, rendeletek, szabványok és ajánlások felsorolása.
- Jelenlegi üzleti folyamatok modellje.
- Igényelt üzleti folyamatok modellje.
- Követelménylista.
- Használati esetek.
- Képernyőtervezek.
- Forgatókönyvek.
- Funkció – követelmény megfeleltetés.

- Fogalomszótár.

Látható, hogy a követelményspecifikációhoz képest sok ismétlődő fejezet van. Ezeket nem fontos átemelni, elég csak hivatkozni rájuk. Az egyes módszertanok eltérnek abban, hogy mely fejezeteket és milyen mélységen kell elkészíteni. Általában elmondható, hogy a modern módszertanok használatieset-központúak.

A funkcionális specifikáció fontos része az úgynevezett megfeleltetés (angolul: traceability), ami megmutatja, hogy a követelményspecifikációban felsorolt minden követelményhez van-e azt kielégítő funkció.

A megrendelőnek küldjük el a kész specifikációt. Érdemes néhány megbeszélésen bemutatni a képernyőterveket, a forgatókönyveket. minden megbeszélésről készítsünk jegyzőkönyvet. Ha a funkcionális specifikációt elfogadta a megrendelő, akkor következik az árajánlat kialakítása.

#### 2.2.3.2. Ütemterv

Az árajánlat legfontosabb része az ütemterv. Az ütemterv határozza meg, hogy mely funkciók kerülnek be a rendszer következő verziójába és melyek maradnak ki. Egy példa ütemterv a mellékletben található. Az ütemtervről részletesen az 6.3 fejezetben írunk.

#### 2.2.3.3. Árajánlat

Az árajánlat kialakításához legalább két dolgot kell tudnunk:

- Hány embernapig fog tartani a fejlesztés, illetve
- mekkora a napidíjunk.

Az első adat az ütemtervből kiolvasható. A második adatot általában úgy számoljuk ki, hogy 50%-os kihasználtság mellett legyen a cégünk nullszaldós. Az árajánlat formájáról és elkészítéséről részletesen az 6.4 fejezetben írunk. A napidíj kiszámításáról részletesen az 6.3.1 fejezetben írunk.

#### 2.2.4. Rendszerspecifikáció

Ebben a fázisban már általában szerződés van a kezünkben. Ha mégsem, akkor valószínűleg egy nagy pályázatot írunk, amihez magvalósíthatósági tanulmány is kell. Ha ezek egyike sem, akkor nagyon nagy kockázatot vállalunk, mert nem biztos, hogy a megrendelő meg is rendeli a rendszert.

Ebben a fázisban két dokumentumot szoktunk elkészíteni. Ezek a következők:

- megvalósíthatósági tanulmány,
- nagyvonalú rendszerterv.

##### 2.2.4.1. Megvalósíthatósági tanulmány

A projekt megvalósíthatósági tanulmánya általában egy 10-50 oldalas dokumentum a projekt nagyságától függően. A megvalósíthatósági tanulmány célja, hogy megfelelő információkkal lássa el a döntéshozókat a projekt indításával, finanszírozásával kapcsolatban. A megvalósíthatósági tanulmányról részletesen az 6.5 fejezetben írunk.

##### 2.2.4.2. Nagyvonalú rendszerterv

A rendszerterv egy írásban rögzített specifikáció, amely leírja

- mit (rendszer),
- miért (rendszer célja),
- hogyan (terv),
- mikor (időpont),
- és miből (erőforrások)

akarunk a jövőben létrehozni. Fontos, hogy reális legyen, azaz megvalósítható lépéseket írjon elő. A rendszerterv hasonló szerepet játszik a szoftverfejlesztésben, mint a tervrajz az építkezéseken, tehát elég részletesnek kell lennie, hogy ebből a programozók képesek legyenek megvalósítani a szoftvert.

Háromfajta rendszertervet különböztetünk meg:

- konceptuális (mit, miért),
- nagyvonalú (mit, miért, hogyan, miből),
- részletes (mit, miért, hogyan, miből, mikor).

A rendszerterv fajtáiról részletesen az 6.6 fejezetben írunk.

#### 2.2.5. Logikai és fizikai tervezés

A logikai és fizikai rendszertervek nagyvonalú vagy részletes rendszertervek, amelyek leírják a rendszer megvalósításának (hogyan) részleteit. A logikai rendszerterv az ideális rendszer terve, amely nem veszi figyelembe a fizikai megszorításokat, csak a rendszer saját megszorításait. Például fizikai megszorítás, hogy mekkora sávszélesség áll rendelkezésünkre, de logikai, hogy minden terméknek van ára.

A fizikai rendszerterv a logikai rendszerterv finomítása. A finomítás során figyelembe vesszük a fizikai megszorításokat. Gyakran úgy tűnik, hogy a logikai és a fizikai modell nagyon távol áll egymástól, mert ha a logikai modellben vázolt megoldás például túl lassú lenne, akkor a fizikaiban egy teljesen másik megoldást kell leírni. Ugyanakkor ez is csak finomítás, hiszen a két megoldás ugyanarra a kérdésre ad választ.

A rendszerterv általában UML ábrákból és szöveges részekből áll. Az UML ábrák önmagukban nem egyértelműek, ezért kell szöveges magyarázattal is ellátni őket.

Sok modern módszertan már nem követeli meg a rendszerterv készítését, főleg az úgynevezett könnyűsúlyú (lightweight) módszertanok. Ugyanis a rendszerterv elkészítése sok időbe, akár a fejlesztési idő kétszeresébe is kerülhet, ami alatt rohanó világunkban gyakran megváltoznak a követelmények, így a rendszerterv érvényét veszti. A régebbi, strukturált módszertanok általában részletes rendszertervet írnak elő.

A rendszerterv alapja a funkcionális specifikáció. Míg a funkcionális specifikáció a felhasználó szemszögéből írja le a rendszert, addig a rendszerterv a programozók, illetve üzemeltetők szemszögéből.

Egy rendszerterv általában az alábbi fejezetekből és alfejezetekből áll:

- A rendszer célja
- Projektterv

- Projektszerepkörök, felelősségek
  - Projektmunkások és felelősségeik
  - Ütemterv
  - Mérföldkövek
- Üzleti folyamatok modellje
  - Üzleti szereplők
  - Üzleti folyamatok
  - Üzleti entitások
- Követelmények
  - Funkcionális követelmények
  - Nemfunkcionális követelmények
  - Törvényi előírások, szabványok
- Funktionális terv
  - Rendszerszerelők
  - Rendszerhasználati esetek és lefutásaik
  - Határosztályok
  - Menü-hierarchiák
  - Képernyőtervezek
- Fizikai környezet
  - Vásárolt softwarekomponensek és külső rendszerek
  - Hardver és hálózati topológia
  - Fizikai alrendszer
  - Fejlesztő eszközök
  - Keretrendszer (pl. Spring)
- Absztrakt domain modell
  - Domain specifikáció, fogalmak
  - Absztrakt komponensek, ezek kapcsolatai
- Architekturális terv
  - Egy architekturális tervezési minta (pl. MVC, 3-rétegű alkalmazás, ...)
  - Az alkalmazás rétegei, fő komponensei, ezek kapcsolatai
  - Változások kezelése
  - Rendszer bővíthetősége
  - Biztonsági funkciók
- Adatbázisterv
  - Logikai adatmodell
  - Tárolt eljárások
  - Fizikai adatmodellt legeneráló SQL szkript
- Implementációs terv
  - Perzisztencia-osztályok
  - Üzleti logika osztályai
  - Kliensoldal osztályai
- Tesztterv
- Telepítési terv
- Karbantartási terv

A rendszerterv elkészítéséről részletesen az 6.7 fejezetben írunk.

#### 2.2.6. Implementáció

Az implementáció során valósítjuk meg az előző fázisokban megtervezett rendszert. Az újabb módszertanok szerint a kódolást előbbre kell hozni, prototípusokat kell fejleszteni. Akár így, akár úgy, de itt már a programozóké a főszerep. Szerencsés esetben rendelkezésükre áll egy részletes rendszerterv, ami metódusszintig specifikálja a feladatot. Kevésbé szerencsés esetben csak képernyőtervek és az elkészítendő funkciók leírása adott.

A feladattól függően más-más nyelven érdemes implementálni a feladatot. Jó választás lehet a Java, ami platformfüggetlen, illetve a .NET Windows platform esetén. Mindkét nyelv kiválóan támogatott, rengeteg rendszeralkönyvtár áll szolgálatunkra. Ezen túl, minden nyelv általános célú. Ha nagyon jól meghatározott területre kell fejlesztenünk, például egy szakértői rendszert, akkor érdemes nem általános célú nyelvet, hanem egy szakterület specifikus nyelvet (angolul: Domain Specific Language, röviden: DSL) használni a fejlesztéshez.

Implementáció során felfedezhetjük, hogy a specifikáció nem megvalósítható vagy ellentmondásos. Ekkor egy előző fázisra kell visszalépnünk és módosítani a specifikációt.

Törekedni kell arra, hogy az elkészülő osztályok, modulok újrafelhasználhatók legyenek. A már meglévő modulokat érdemes újra felhasználni. Érdemes külső komponenseket is felhasználni, de ekkor időt vesz igénybe a komponens API-jának megtanulása.

Ha sok hasonló projektet csinálunk, akkor érdemes egy keretrendszert kifejleszteni, amiben egy újabb hasonló projekt elkészítése sokkal rövidebb ideig tart. Ekkor egy magasabb absztrakciós szintet vezetünk be. minden ilyen lépésnél ügyelnünk kell a szivárgó absztrakcióra. Lásd a bevezetésben.

Az implementáció során törekednünk kell arra, hogy könnyen újrafelhasználható és rugalmas kódot írunk. Erről nagy részletezzéggel írunk a programozás technológia fejezetben.

Létrejövő termékek:

- forráskód,
- programozói dokumentáció.

#### 2.2.7. Tesztelés

Tesztelésre azért van szükség, hogy a szoftvertermékben meglévő hibákat még az üzembehelyezés előtt megtaláljuk, ezzel növeljük a termék minőségét, megbízhatóságát. Abban szinte biztosak lehetünk, hogy a szoftverben van hiba, hiszen azt emberek specifikálják, tervezik és fejlesztik, és az emberek hibáznak. Gondolunk arra, hogy a legegyszerűbb programban, mondjuk egy szöveges menükezelésben, mennyi hibát kellett kijavítani, mielőtt működőképes lett. Tehát abban szinte biztosak lehetünk, hogy tesztelés előtt van hiba, abban viszont nem lehetünk biztosak, hogy tesztelés után nem marad hiba. A tesztelés után azt tudjuk elmondani, hogy a letesztelt részekben nincs hiba, így nő a program megbízhatósága. Ez azt is mutatja, hogy a program azon funkcióit kell tesztelni, amiket a felhasználók legtöbbször fognak használni. A fenti megállapításokat a következő elvekben, a tesztelés alapelveiben, foglalhatjuk össze:

1. A tesztelés hibák jelenlétéét jelzi: A tesztelés képes felfedni a hibákat, de azt nem, hogy nincs hiba. Ugyanakkor a szoftver minőségét és megbízhatóságát növeli.
2. Nem lehetséges kimerítő teszt: minden bemeneti kombinációt nem lehet letesztelni (csak egy 10 hosszú karakterláncnak  $256^{10}$  lehetséges értéke van) és nem is érdemes. Általában csak a magas kockázatú és magas prioritású részeket teszteljük.
3. Korai teszt: Érdemes a tesztelést a szoftverfejlesztés életciklusának minél korábbi szakaszában elkezdeni, mert minél hamarabb találunk meg egy hibát (mondjuk a specifikációban), annál olcsóbb javítani. Ez azt is jelenti, hogy nemcsak programot, hanem dokumentumokat is lehet tesztelni.
4. Hibák csoportosulása: A tesztelésre csak véges időnk van, ezért a tesztelést azokra a modulokra kell koncentrálni, ahol a hibák a legvalószínűbbek, illetve azokra a bementekre kell tesztelnünk, amelyre valószínűleg hibás a szoftver (pl. szélsőértékek).
5. A féregirtó paradoxon: Ha az újratesztelés során (lásd később a regressziós tesztet) minden ugyanazokat a teszteseteket futtatjuk, akkor egy idő után ezek már nem találnak több hibát, mintha a férgék alkalmazkodnának a teszthez. Ezért a tesztjeinket folyamatosan bővíteni kell.
6. A tesztelés függ a körülményektől: Másképp tesztelünk egy atomerőműnek szánt programot és egy beadandót. Másképp tesztelünk, ha a tesztre 10 napunk vagy csak egy éjszakánk van.
7. A hibátlan rendszer téveszméje: Hiába javítjuk ki a hibákat a szoftverben, azzal nem lesz elégedett a megrendelő, ha az nem felel meg az igényeinek. Azaz használhatatlan szoftvert nem érdemes tesztelni.

#### 2.2.7.1. Tesztelési technikák

A tesztelési technikákat csoportosíthatjuk a szerint, hogy a teszteseteket milyen információ alapján állítjuk elő. E szerint létezik:

- Feketedobozos (angolul: black-box) vagy specifikáció alapú, amikor a specifikáció alapján készülnek a tesztesetek.
- Fehérdobozos (angolul: white-box) vagy strukturális teszt, amikor a forráskód alapján készülnek a tesztesetek.
- Szürkedobozos (angolul: grey-box), amikor a forráskódnak csak egy része ismert és ez alapján készülnek a tesztesetek.

Tehát beszélünk feketedobozos tesztelésről, amikor a tesztelő nem látja a forráskódot, de a specifikációkat igen, fehérdobozos tesztelésről, amikor a forráskód rendelkezésre áll, illetve szürkedobozos tesztről, amikor a forráskódnak csak egy része, általában néhány interfész áll rendelkezésre.

A feketedobozos tesztelést specifikáció alapúnak is nevezzük, mert a specifikáció alapján készül. Ugyanakkor a teszt futtatásához szükség van a lefordított szoftverre. Leggyakoribb formája, hogy egy adott bemenetre tudjuk, milyen kimenetet kellene adni a programnak. Lefuttatjuk a programot a bemenetre és összehasonlítjuk a kapott kimenetet az elvárttal. Ezt alkalmazzák pl. az ACM versenyeken is.

A fehérdobozos tesztelést strukturális tesztelésnek is nevezzük, mert minden lehetséges bemenetet le kell vizsgálni. A strukturális teszt esetén értelmezhető a (struktúra)lefedettség. A

lefedettség azt mutatja meg, hogy a struktúra hány százalékát tudjuk tesztelni a meglévő teszesetekkel. Általában ezeket a struktúrákat teszteljük:

- kód sorok,
- elágazások,
- metódusok,
- osztályok,
- funkciók,
- modulok.

Például az egységeszt (angolul: unit-test) a metódusok struktúratesztje.

#### 2.2.7.2. A tesztelés szintjei

A tesztelés szintjei a következők:

- komponensteszt,
- integrációs teszt,
- rendszerteszt,
- átvételi teszt.

A komponensteszt a rendszernek csak egy komponensét teszteli önmagában. Az integrációs teszt kettő vagy több komponens együttműködési tesztje. A rendszerteszt az egész rendszert, tehát minden komponenst együtt, teszteli. Ez első három tesztszintet együttesen fejlesztői tesztnek hívjuk, mert ezeket a fejlesztő cég alkalmazottai vagy megbízottai végzik. Az átvételi teszt során a felhasználók a kész rendszert tesztelik. Ezek általában időrendben is így követik egymást.

A komponensteszt a rendszer önálló részeit teszteli általában a forráskód ismeretében (fehér dobozos tesztelés). Gyakori fajtái:

- egységeszt,
- modulteszt.

Az egységeszt, vagy más néven unit-teszt, a metódusokat teszteli. Adott paramétereire ismerjük a metódus visszatérési értékét (vagy mellékhatását). Az egységeszt megvizsgálja, hogy a tényleges visszatérési érték megegyezik-e az elvárttal. Ha igen, sikeres a teszt, egyébként sikertelen. Elvárás, hogy magának az egységesztnak ne legyen mellékhatása.

Az egységesztelést majd minden fejlett programozási környezet (angolul: Integrated Development Environment, röviden: IDE) támogatja, azaz egyszerű ilyen teszteket írni. A jelentőségüket az adja, hogy a futtatásukat is támogatják, így egy változtatás után csak lefuttatjuk az összes egységesztet, ezzel biztosítjuk magunkat, hogy a változás nem okozott hibát. Ezt nevezzük regressziós tesztnek.

A modulteszt általában a modul nemfunkcionális tulajdonságát teszteli, pl. sebességét, vagy, hogy van-e memóriasivárgás (angolul: memory leak), van-e szűk keresztmetszet (angolul: bottleneck).

Az integrációs teszt során a komponensek közti interfészket, az operációs rendszer és a rendszer közti interfészt, illetve más rendszerek felé nyújtott interfészket tesztelik. Az integrációs teszt legismertebb típusai:

- Komponens – komponens integrációs teszt: A komponensek közötti kölcsönhatások tesztje a komponensteszt után.
- Rendszer – rendszer integrációs teszt: A rendszer és más rendszerek közötti kölcsönhatásokat tesztje a rendszerteszt után.

Az integrációs teszt az összeillesztés során keletkező hibákat keresi. Mivel a részeket más-más programozók, csapatok fejlesztették, ezért az elég telen kommunikációból súlyos hibák keletkezhetnek. Gyakori hiba, hogy az egyik programozó valamit feltételez (pl. a metódus csak pozitív számokat kap a paraméterében), amiről a másik nem tud (és meghívja a metódust egy negatív értékkal). Ezek a hibák kontraktus alapú tervezéssel (angolul: design by contract) elkerülhetők.

Az integrációs teszteket érdemes előrehozni, amennyire lehet, mert minél nagyobb az integráció mértéke, annál nehezebb meghatározni, hogy a fellelt hiba (általában egy futási hiba) honnan származik. Ellenkező esetben, azaz amikor már minden komponens kész és csak akkor tesztelünk, akkor ezt a „nagy bumm tesztnek” (angolul: big bang test) nevezzük, ami rendkívül kockázatos.

A rendszerteszt a már kész szoftverterméket teszteli, hogy megfelel-e:

- a követelményspecifikációnak,
- a funkcionális specifikációnak,
- a rendszertervnek.

A rendszerteszt nagyon gyakran feketedobozos teszt. Gyakran nem is a fejlesztő cég, ahol esetleg elfogultak a tesztelők, hanem egy független cég végzi. Ilyenkor a tesztelők és a fejlesztők közti kapcsolattartást egy hibabejelentő (angolul: bug tracking) rendszer látja el. A rendszerteszt feladata, hogy ellenőrizze, hogy a specifikációknak megfelel-e a termék. Ha például a követelményspecifikáció azt írja, hogy lehessen jelentést készíteni az éves forgalomról, akkor ezt a tesztelők kipróbálják, hogy lehet-e, és hogy helyes-e a jelentés. Ha hibát találnak, azt felviszik a hibabejelentő rendszerbe.

Fontos, hogy a rendszerteszthez használt környezet a lehető legjobban hasonlítsa megrendelő környezetére, hogy a környezet-specifikus hibákat is sikérüljön felderíteni.

Az átvételi teszt, hasonlóan a rendszerteszthez, az egész rendszert teszteli, de ezt már a végfelhasználók végzik. Az átvételi teszt legismertebb fajtái a következők:

- alfa teszt,
- béta teszt,
- felhasználói átvételi teszt,
- üzemeltetői átvételi teszt.

Az alfa teszt a késztermék tesztje a fejlesztő cégnél, de nem a fejlesztő csapat által. Ennek része, amikor egy kis segédprogram több millió véletlen egérkattintással ellenőrzi, hogy össze-vissza kattintgatva sem lehet kifejtetni a programot.

Ezután következik a béta teszt. A béta tesztet a végfelhasználók egy szűk csoportja végzi. Játékoknál gyakori, hogy a kiadás előtt néhány fanatikus játékosnak elküldik a játékot, akik rövid idő alatt sokat játszanak vele. Cserébe csak azt kérik, hogy a felfedezett hibákat jelentsék.

Ezután jön egy sokkal szélesebb béta teszt, amit felhasználói átvételi tesztnak nevezünk. Ekkor már az összes, vagy majdnem az összes felhasználó megkapja a szoftvert és az éles környezetben használatba veszi. Azaz installálják és használják, de még nem a termelésben. Ennek a tesztnak a célja, hogy a felhasználók meggyőződjenek, hogy a termék biztonságosan használható lesz majd éles körülmények között is. Itt már elvárt, hogy a fő funkciók minden működjenek, de előfordulhat, hogy az éles színhelyen előjön olyan környezetfüggő hiba, ami a teszkörnyezetben nem jött elő. Lehet ez pl. egy funkció lassúsága.

Ezután már csak az üzemeltetői átvételi teszt van hátra. Ekkor a rendszergazdák ellenőrzik, hogy a biztonsági funkciók, pl. a biztonsági mentés és a helyreállítás, helyesen működnek-e.

#### 2.2.8. Rendszerátadás és bevezetés

A szoftverbevezetés angol elnevezése a deployment. A bevezetést megelőzően gondoskodni kell a megfelelő rendszerkörnyezetről. Ez egyszerűt történhet a meglévő infrastruktúra felhasználásával, vagy új eszközpark beszerzésével, telepítésével. Az erre vonatkozó követelményeket is össze kell gyűjtenünk a követelményspecifikáció készítésekor. A rendszerkörnyezet leírása a fizikai rendszerterv része. A rendszertesztek tervezésekor figyelembe kell venni a fizikai környezetet is.

A rendszer bevezetésekor szükség lehet a személyzet képzésére, mely szintén megállapodás tárgyát képezi, figyelembe véve a ráfordítások nagyságát. Ez a lépés nagyban javíthatja a rendszer hatékonyságát, illetve segíti a szoftver „személyre szabását”, a jogosultsági szintek helyes beállítását.

Fontos rögzíteni a rendszer bevezetésére szánt időt, illetve a rendszerátadás várható idejét. Miután a rendszerátadás ténylegesen megtörtént, dokumentálva lett, befejeződött a fejlesztési folyamat. Biztosítani kell a fellépő garanciális problémák elhárítását, esetleg a rendszer üzemeltetését, mely megállapodás kérdése.

Az átadás dokumentumai:

- Felhasználói dokumentáció
- Üzembehelyezési kézikönyv
- Átadás-átvételi jegyzőkönyv

#### 2.2.9. Üzemeltetés és karbantartás

Az üzemeltetés a rendszergazda feladata. A rendszergazda lehet a megrendelő munkatársa, vagy külső munkaerő is. Az üzemeltetés során elvárás:

- a rendszeres biztonsági mentés
- meghibásodás esetén az utolsó konzisztens állapot visszaállítása
- a rendszer frissítése
- az általános átvizsgálás során észlelt hibák javítása
- bejelentések során érkezett hibák javítása
- a munkatársak igényeinek folyamatos figyelése és felmérése

- biztonsági beállítások folyamatos felülvizsgálata és szükség esetén korrigálásuk.

A szoftverrendszernek támogatnia kell ezeket a tevékenységeket.

Az üzemeltetés során fellépő hibákat általában egy évig ingyenesen kell javítania a megoldást szállító cégnak. Ettől szerződésben el lehet tértani.

Kisebb informatikai szolgáltatási zavarok is károsan befolyásolhatják a szervezeti tevékenységeket, ezért a szolgáltatásokat úgy kell megtervezni és fenntartani, hogy minimálisra csökkentsük bármely hiba hatását. A rendelkezésre állás feladata, hogy a rendszerek általános rendelkezésre állását javítsa a felhasználók szolgáltatási igényeinek kielégítése érdekében, a megelőző és a javító karbantartási tevékenység optimalizálása révén.

A rendelkezésre állás négy fő területe:

- Megbízhatóság (reliability): egy információtechnológiai összetevő azon képessége, hogy ellásson egy megkívánt funkciót meghatározott körülmények között, egy meghatározott időtartamra.
- Karbantarthatóság (maintainability): egy számítógépes komponens vagy szolgáltatás azon képessége, hogy meg lehet tartani egy olyan állapotban, vagy vissza lehet állítani egy olyan állapotba, amelyben képes ellátni a megkívánt funkciót.
- Szolgáltatási képesség (serviceability): szerződéses kikötés, amely meghatározza az informatikai komponens rendelkezésre állását az adott összetevőket szolgáltató és karbantartó külső szervezettel való megegyezés szerint.
- Biztonság (security): lehetővé teszi a számítógépes komponensek vagy informatikai szolgáltatások elérését biztonságos körülmények között.

Figyelni kell arra, hogy az informatikai szolgáltatások hibáinak száma és időtartama igazolható költséghatáron belül maradjon.

Az informatikai rendszereket és szolgáltatásokat úgy kell tervezni, hogy megbízhatóak, hibatűrők és karbantarthatóak legyenek a szoftverfejlesztés teljes életciklusán keresztül, a tervezéstől a megszüntetésükig. Évről évre jobban támaszkodunk az informatikai szolgáltatásokra. Ez a függőség olyan naggyá vált, hogy:

- a kézi rendszerekre való visszaállás gyakorlatilag lehetetlen,
- a felhasználók hatékonysága és eredményessége erősen függ az informatikai szolgáltatások rendelkezésre állásától és megbízhatóságától,
- a szervezeti felhasználók tevékenysége az informatikán alapul, amely nélkül a szervezet működésképtelen.

#### 2.2.10. [Az szoftverfejlesztés életciklusának dokumentumai](#)

Habár a fejezetben részletesen felsoroltuk és leírtuk az életciklus során keletkező dokumentumokat és termékeket, mégis hasznosnak tartjuk ezeket egy helyen is felsorolni:

- követelményspecifikáció,
- funkcionális specifikáció,

- ütemterv,
- árajánlat,
- szerződés,
- megvalósíthatósági tanulmány,
- nagyvonalú rendszerterv,
- szoftver,
- programozói dokumentáció,
- teszterterv,
- tesztelt szoftver,
- felhasználói dokumentáció,
- üzembe helyezési kézikönyv,
- átadás-átvételi jegyzőkönyv,
- biztonsági mentés.

### 2.3. Módszertanok

A módszertanok feladata, hogy meghatározzák, hogy a szoftverfejlesztés életciklusának egyes lépései milyen sorrendben kövessék egymást, milyen dokumentumokat, szoftvertermékeket előállítunk elő és hogyan. Egy nagy szabálykönyvre emlékeztetnek, ami pontosan leírja, hogyan kell szoftvert „főzni”. Ha betartjuk a receptet, akkor egy átlagos minőségű szoftvert kapunk, de az átlagos minőség garantált. Erről bővebben olvashatunk a „Joel on Software” című blog „A Big Mac és a Mezítelen Sér” című bejegyzésében: <http://hungarian.joelonsoftware.com/Articles/BigMacvs.TheNakedChef.html>.

A módszertanokat több szempontból is osztályozhatjuk. Az első szempont, hogy milyen sorrendben követik egymást a szoftverfejlesztés életciklusának fázisai. E szerint van:

- lineáris,
- spirális
- iteratív (vagy inkrementális).

A második szempont, hogy milyen implementációs nyelvet részesít előnyben a módszertan. E szerint van:

- folyamatorientált,
- adatközpontú,
- strukturált (ezen belül lehet top-down és button-up),
- objektumorientált,
- szervizorientált,
- esetleg ezek keveréke.

A harmadik szempont, hogy milyen megközelítést használ a módszertan a célja eléréséhez. Mivel a cél a sikeres szoftverprojekt, ezért a megközelítés alatt általában egy olyan technikát értünk, ami a félreértek elkerülésére szolgál. Ugyanakkor a megközelítés alatt a programozásra tett ajánlást is érhetjük. E szerint van:

- jól dokumentált,

- prototípus alapú,
- rapid,
- agilis,
- extrém,
- esetleg ezek keveréke.

A negyedik szempont, hogy mennyire szigorúan követelik meg a jól dokumentáltságot. E szerint van:

- könnyűsúlyú (angolul: lightweight) módszertan,
- nehézsúlyú (angolul: heavyweight) módszertan.

Az ötödik szempont, hogy mit helyez a modell középpontjába a módszertan. E szerint van:

- adatközpontú,
- folyamatközpontú,
- követelményközpontú,
- használatieset-központú,
- tesztközpontú,
- felhasználóközpontú,
- emberközpontú,
- csapatközpontú,
- esetleg ezek keveréke.

Sok módszertan nem sorolható be egyetlen kategória alá, mert például az adatbázistervezéshez adatközpontú módszereket ír elő, egyébként használatieset-központú.

Általában minden módszertanra jellemzők a következők:

- Az elemzés és tervezés szétválasztása.
- A logikai és fizikai tervezés szétválasztása.

Ebben a fejezetben röviden bemutatjuk a legelterjedtebb módszertanokat.

### 2.3.1. [Strukturált módszertanok](#)

A strukturált módszertanok a feladatot modulokra bontják, az implementációban is ajánlott a program modulokra bontása. Ezért a strukturált programozási nyelveket részesítik előnyben. Mivel az objektumorientált nyelvek is strukturáltak, ezért azokkal együtt is használhatók. Jellemzője, hogy a szoftverfejlesztés életciklusának lépései merev sorrendben követik egymást, azaz lineáris. Ezek a módszertanok nagy, hosszú ideig tartó projektek megvalósítására valók. Közös jellemzőjük, hogy nehézsúlyúak, azaz sok és részletes dokumentációval próbálják meg elkerülni a félreértéseket. Az egyes dokumentumok elkészítéséhez általában adatközpontú vagy folyamatközpontú technikákat ajánl.

Tehát a strukturált módszertanok:

- lineárisak,
- jól dokumentáltak,

- nehézsúlyúak,
- adat- és folyamatközpontúak.

A strukturális módszertanok egy gyűjtőfogalom. Jelentőségüket az adja, hogy ezek jelentek meg elsőnek és terjedtek el széles körben. Megjelenésük idején azt gondolták, ezek a módszertanok lesznek a szoftverkriszis megoldói. Sok, ma is népszerű technika, pl. adatfolyammodellezés, ezekben a módszertanokban jelent meg.

Strukturális módszertanok:

- Vízesés modell,
- SSADM,
- V-modell.

### 2.3.2. [Vízesés modell](#)

A vízesés modell (waterfall model) volt az első módszertan, ami széles körben elterjedt. Ez egy strukturális módszertan, ezek közül is a legismertebb. Nagy megrendelők nagy projektjeihez alakították ki. Mivel a nagy megrendelők általában rugalmatlanok, ezért előnyös, hogy a módszertan kevés döntési pontot definiál.

A modell lineáris, azaz a szoftverfejlesztés életciklusának lépései egymás után, átfedés nélkül következnak. A módszertan ipari termelésből ered, ahol nincs lehetőség a követelmények változtatására, ha már egyszer azokat meghatároztuk. Ezért a vízesés modell eredeti változata nem engedi, hogy visszatérjünk egy már lezárt fázisba. Ezt azért teheti meg, mert csak akkor léphetünk a következő fázisba, ha az előzőt már tökéletesen sikerült lezárni. A modell rengeteg dokumentum elkészültét írja elő. Mivel ezeket el kell fogadtatni a megrendelővel, ezzel véli biztosítani a módszertan, hogy nincs félreértés a megrendelő és a szoftvercég közt.

Sajnos a sok dokumentumot ritkán olvassa el tüzetesen a megrendelő. Ezért ez a technika nem csak itt, hanem más módszertanok esetén sem alkalmas a félreértések elkerülésére. Ugyanakkor a modell feltételezi, hogy precíz mérnökökkel lesz dolgunk, akik kritikusan olvassák a dokumentumokat. Ez a feltételezés jogos. Tehát a vízesés modell igazi kritikája, hogy nem engedi a követelmények menet közbeni változtatását, ami a mai felgyorsult világunkban nem jogos megszorítás.

Tehát a vízesés modell:

- lineáris,
- strukturált,
- jól dokumentált,
- nehézsúlyú,
- nem agilis.

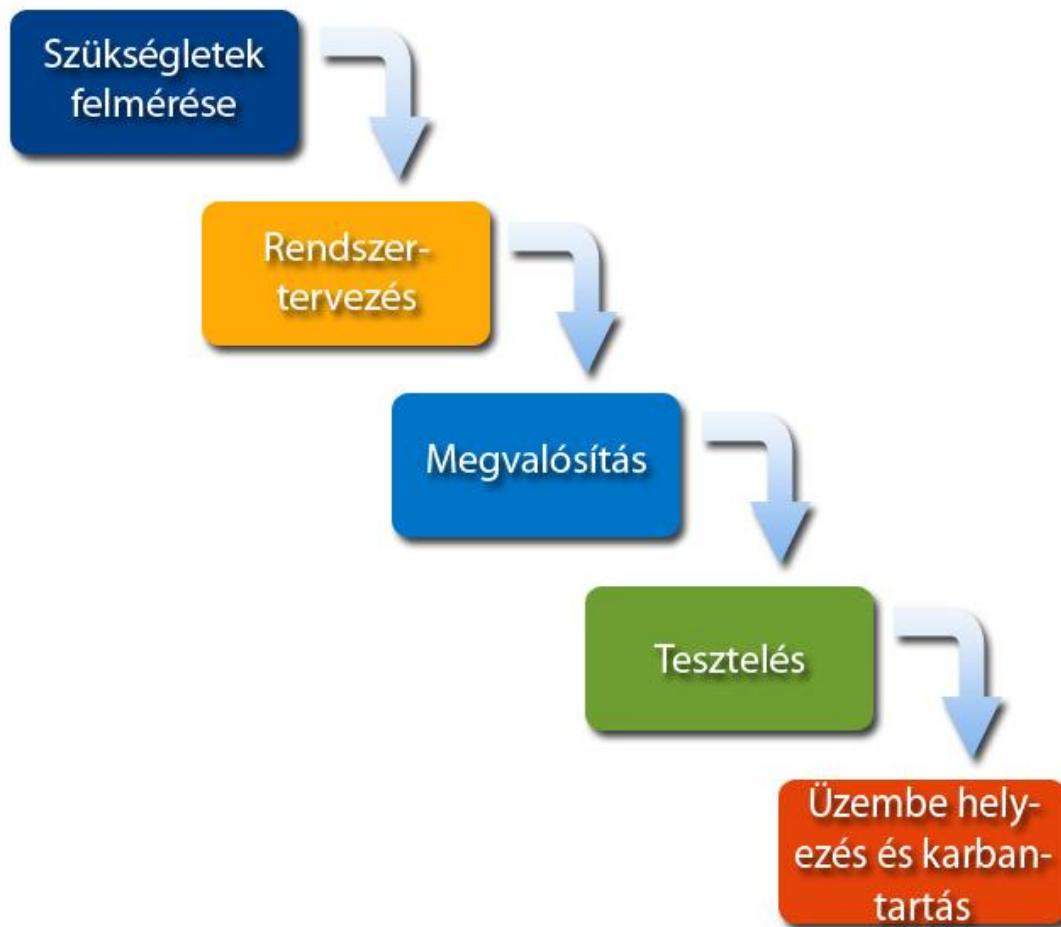
A vízesés modell nem határoz meg technikákat, de általában adat- és folyamatközpontú technikákkal együtt alkalmazták. Mostanában használatieset-központú technikákkal is szokták használni.

A modell a következő fázisokat írja elő:

- szükségletek felmérése (angolul: Requirements Analysis),
- rendszertervezés (angolul: System Design),
- megvalósítás (angolul: Implementation),
- tesztelés (angolul: Testing),
- üzembe helyezés és a karbantartás (angolul: Delivery).

Ezek a folyamatok egymást követik, fel nem cserélhetőek, valamint akkor következik az egyik fázis után a másik, ha az előző teljesen befejeződött, a kitűzött célokat elértük. Ezért is hívják vízesés modellnek.

## Vízesés modell



3. ábra: A vízesés modell

A vízesés modell előnye az erős kontroll a folyamatok felett. Ütemtervet lehet készíteni, hogy melyik fázis mikor készül el. A hátránya, hogy nem engedi meg az újragondolást, áttervezést. A fejlesztés végig megy az egyes fázisokon, szigorúan, csak lineáris sorrendben, nincsenek ismétlődő lépések.

### 2.3.3. SSADM

Az SSADM (angol: Structured Systems Analysis and Design Method) Strukturált Rendszerelemzési és Tervezési Módszertan rövidítése. Az SSADM egy strukturált módszertan, a vízesés módszertan egy jól kidolgozott változata. Első változata 1984-ben jelent meg, az utolsó 1995-ben. A brit kormány szabvánnyal alkalmazta szoftverprojektekre. Magyarországon 1989-ben kezdett elterjedni, miután a britek nyilvánossá tették a módszertant, illetve a nagy lökést még az ebben ez évben elkészített magyar fordítás (<http://www.itb.hu/ajanlasok/a4>) adta. Az államigazgatásban azóta is ajánlott módszertan.

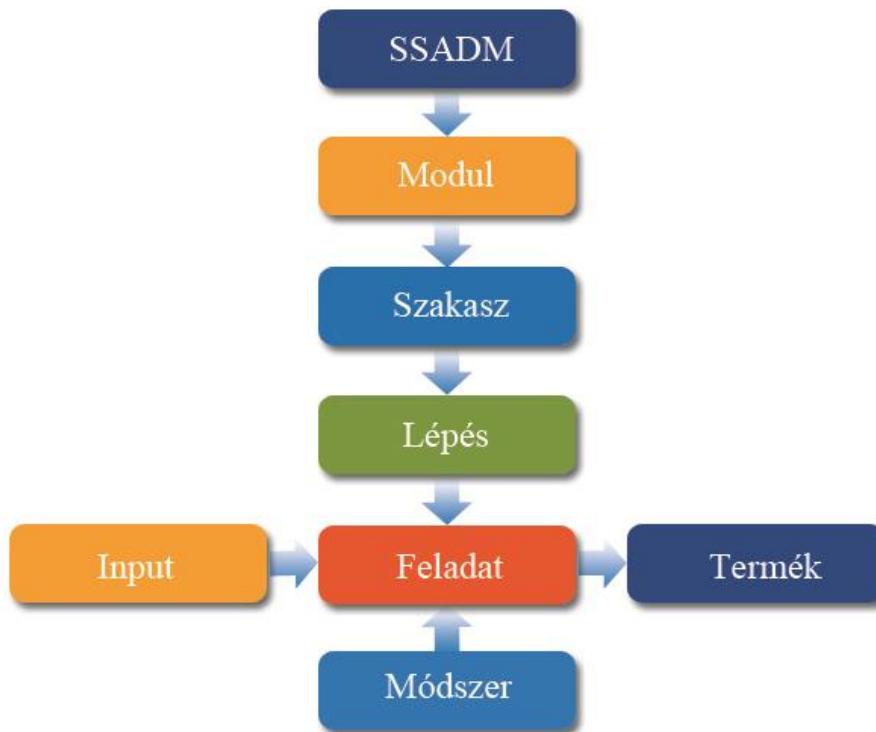
Az SSADM alapelvei:

- Adatközpontúság: A legfontosabb modellezési technika az adatmodell-elemzés (angolul: entity model, entity relation model, database model). A top-down megközelítést ajánlja, tehát az egész rendszert bontjuk kisebb komponensekre.
- Termékszemlélet: Az elemzési és tervezési feladatok kimenetét terméknek tekintjük, amikkel szemben minőségi követelményeket szabunk meg, ezeket ellenőrizzük.

Tehát az SSADM módszertan:

- lineáris,
- strukturált,
- jól dokumentált,
- nehézsúlyú,
- adat-, folyamat- és termékközpontú, top-down megközelítés,
- nem agilis.

A módszertan öt modult és hét szakaszt tartalmaz. A szakaszokat lépésekre, a lépéseket feladatokra bontja.



4. ábra: Az SSADM felépítése

Minden feladathoz előírja a kötelezően alkalmazandó módszert, amelynek megadja, hogy milyen - már kész - termékből kell kiindulnia, és milyen terméket kell előállítania. A feladatokhoz előírja a kötelezően alkalmazandó módszert, amelynek adott az inputja, és meghatározott az outputja. A három legfontosabb módszere:

- Logikai adatmodellezés (angolul: Logical Data Modeling)
- Adatfolyam-modellezés (angolul: Data Flow Modeling)
- Egyed-esemény modellezés (angolul: Entity Behavior Modeling)

Moduljai és szakaszai a következők:

- 1. Modul: Megvalósíthatósági elemzés
  - 0. Szakasz: Megvalósíthatóság
- 2. Modul: Követelményelemzés
  - 1. Szakasz: Jelenlegi helyzet vizsgálata
  - 2. Szakasz: Rendszerszervezási alternatívák
- 3. Modul: Követelményspecifikáció
  - 3. Szakasz: Követelmények meghatározása
- 4. Modul: Logikai rendszerspecifikáció
  - 4. Szakasz: Rendszertechnikai változat kiválasztása

- 5. Szakasz: Logikai rendszertervezés
- 5. Modul: Fizikai rendszertervezés
  - 6. Szakasz: Fizikai rendszertervezés

Annak szemléltetésére, hogy milyen részletes az SSADM módszertan, itt idézzük a magyar fordításból a 320. lépést, amelynek a neve „Igényelt rendszer adatmodelljének kidolgozása”:

A lépés célja: olyan logikai adatmodell kialakítása, amelyre az igényelt rendszer folyamatai támaszkodhatnak, a logikai adatmodellhez kapcsolódó nemfunkcionális követelmények meghatározása.

Leírás: Ez a lépés a 310. lépéssel párhuzamos. A jelenlegi környezet logikai adatmodelljét ki kell egészíteni a követelményjegyzékbeli új követelményeknek megfelelően. Az első szakaszban csak a legfontosabb adatalemeket kellett meghatározni az egyes egyedekhez, így ennek a lépésnek a feladata az egyedek és kapcsolataik teljes meghatározása. A megfelelő nemfunkcionális követelményeket a logikai adatmodellbe be kell illeszteni.

Részt vesznek a követelményspecifikációs csoport tagjai, adatmodellezők, adatalemzők és más szakértők (pl. adatbiztonság).

Kiindulási alapok: Jelenlegi logikai adatmodell, Adatjegyzék, Igényelt rendszer adatfolyam-modellje, Követelményjegyzék, Választott rendszerszervezási alternatíva

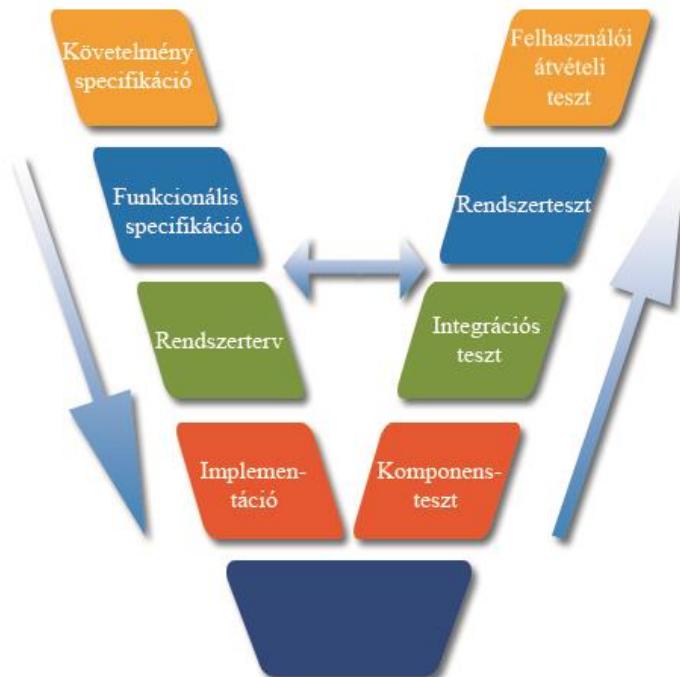
Feladat	Leírás
10	Meg kell vizsgálni a választott rendszerszervezási alternatívákat és a jelenlegi környezet logikai adatmodelljéből csak azokat a részeket kell meghagyni, amelyek a választott alternatívát támogatják. A logikai adatmodellt ki kell egészíteni az új rendszer követelményeinek megfelelően. Ezen a ponton kell a fennmaradó attribútumokat megadni minden egyedhez. Az új követelmények feldolgozását a követelményjegyzékből fel kell jegyezni.
20	Ellenőrizni kell, hogy a logikai adatmodell megfelelően támogatja-e az elemi folyamatok leírásait. Az adatelérési utakat még nem kell formálisan meghatározni ebben a lépésben.
30	A logikai adatmodellt ki kell egészíteni a követelményjegyzékbeli nemfunkcionális követelményeknek megfelelően (pl. hozzáférési korlátozások, biztonsági követelmények, archiválási követelmények).

Előállított vagy módosított termékek: Adatjegyzék, Igényelt rendszer logikai adatmodellje, Követelményjegyzék.

#### 2.3.4. [V-modell](#)

A V-modell (angolul: V-Model vagy Vee Model) a nevét onnan kapta, hogy két szára van és így egy V betűhöz hasonlít. Az egyik szára megegyezik a vízesés modellel. Ez a fejlesztési szár. A másik szára a létrejövő termékek tesztjeit tartalmazza. Ez a tesztelési szár. Az egy szinten lévő fejlesztési és tesztelési lépések összetartoznak, azaz a tesztelési lépés a fejlesztési lépés során létrejött dokumentumokat

használja, vagy a létrejött terméket teszteli. Ennek megfelelően az előírt fejlesztési és tesztelési lépések a következők:



5. ábra: A V-modell

A V-modell a vízesés modell kiegészítése teszteléssel. Ez azt jelenti, hogy először végre kell hajtani a fejlesztés lépésein, ezután jönnek a tesztelés lépései. Ha valamelyik teszt hibát talál, akkor vissza kell menni a megfelelő fejlesztési lépékre.

A V-modell hasonlóan a vízesés modellhez nagyon merev, de alkalmazói kevésbé ragaszkodnak ehhez a merevséghez, mint a vízesés modell alkalmazói. Ennek megfelelően jobban elterjedt. Fő jellemzője a teszt központi szerepe.

Tehát a V-modell:

- lineáris,
- strukturált,
- jól dokumentált,
- nehézsúlyú,
- tesztközpontú.

Egy tipikus V-modell változatban először felmérjük az igényeket és elkészítjük a követelményspecifikációt. Ezt üzleti elemzők végzik, akik a megrendelő és a fejlesztők fejével is

képesek gondolkozni. A követelményspecifikációban jól meghatározott átvételi kritériumokat fogalmaznak meg, amik lehetnek funkcionális és nemfunkcionális igények is. Ez lesz majd az alapja a felhasználói átvételi tesztnek (angolul: User Acceptance Test, röviden: UAT). Magát a követelményspecifikációt is tesztelik. A felhasználók tüzetesen átnézik az üzleti elemzők segítségével, hogy ténylegesen minden igényüket lefedi-e a dokumentum. Ez lényeges része a modellnek, mert a folyamatban visszafelé haladni nem lehet, és ha rossz a követelményspecifikáció, akkor nem az igényeknek megfelelő szoftver fog elkészülni. Ezzel szemben például a prototípusmodellben lehet pongyola az igényfelmérés, mert az a prototípusok során úgyis pontosításra kerül.

Ezután következik a funkcionális specifikáció elkészítése, amely leírja, hogyan kell majd működnie a szoftvernek. Ez lesz a rendszerteszt alapja. Ha a funkcionális specifikáció azt írja, hogy a „Vásárol gomb megnyomására ki kell írni a kosárban lévő áruk értékét”, akkor a rendszertesztben lesz egy vagy több teszeset, amely ezt teszteli. Például, ha üres a kosár, akkor az árnak nullának kell lennie.

Ezután következik a rendszerterv, amely leírja, hogy az egyes funkciókat hogyan, milyen komponensekkel, osztályokkal, metódusokkal, adatbázissal fogjuk megvalósítani. Ez lesz a komponensteszt egyik alapja. A rendszerterv leírja továbbá, hogy a komponensek hogyan működnek együtt. Ez lesz az integrációs teszt alapja.

Ezután a rendszertervnek megfelelően következik az implementáció. minden metódushoz egy vagy több egységesztet, vagy más néven unit-tesztet kell készíteni. Ezek alapja nem csak az implementáció, hanem a rendszerterv is. A nagyobb egységeket, osztályokat, al- és főfunkciókat is komponensteszt alá kell vetni az implementáció és a rendszerterv alapján.

Ha ezen sikeresen túl vagyunk, akkor az integrációs teszt következik a rendszerterv alapján. Ha itt problémák merülnek fel, akkor visszamegyünk a V betű másik szárára a rendszertervhöz. Megnézzük, hogy a hiba a rendszertervben vagy az implementációban van-e. Ha kell, megváltoztatjuk a rendszertervet, majd az implementációt is.

Az integrációs teszt után jön a rendszerteszt a funkcionális specifikáció alapján. Hasonlóan, hiba esetén a V betű másik szárára megyünk, azaz visszalépünk a funkcionális specifikáció elkészítésére. Majd jön az átvételi teszt a követelményspecifikáció alapján. Remélhetőleg itt már nem lesz hiba, mert kezdhetnénk az egészet előlről, ami egyenlő a sikertelen projekttel.

Ha a fejlesztés és tesztelés alatt nem változnak a követelmények, akkor ez egy nagyon jó, kiforrott, támogatott módszertan. Ha valószínű a követelmények változása, akkor inkább iteratív, vagy még inkább agilis módszert válasszunk.

Az egyes tesztfajtákról részletesen a Tesztelés című részben írunk.

### 2.3.5. Prototípusmodell

A prototípusmodell válasz a vízesésmodell sikertelenségére. A fejlesztőcégek rájöttek, hogy tarthatatlan a vízesésmodell megközelítése, hogy a rendszerrel a felhasználó csak a projekt végén találkozik. Gyakran csak ekkor derült ki, hogy a szoftverfejlesztés életciklusának elején félreértezték egymást a felek és nem a valós követelményeknek megfelelő rendszer született. Ezt elkerülendő a prototípusmodell azt mondja, hogy a végső átadás előtt több prototípust is szállítunk le, hogy mihamarabb kiderüljenek a félreértések, illetve a megrendelő lássa, mit várhat a rendszertől.

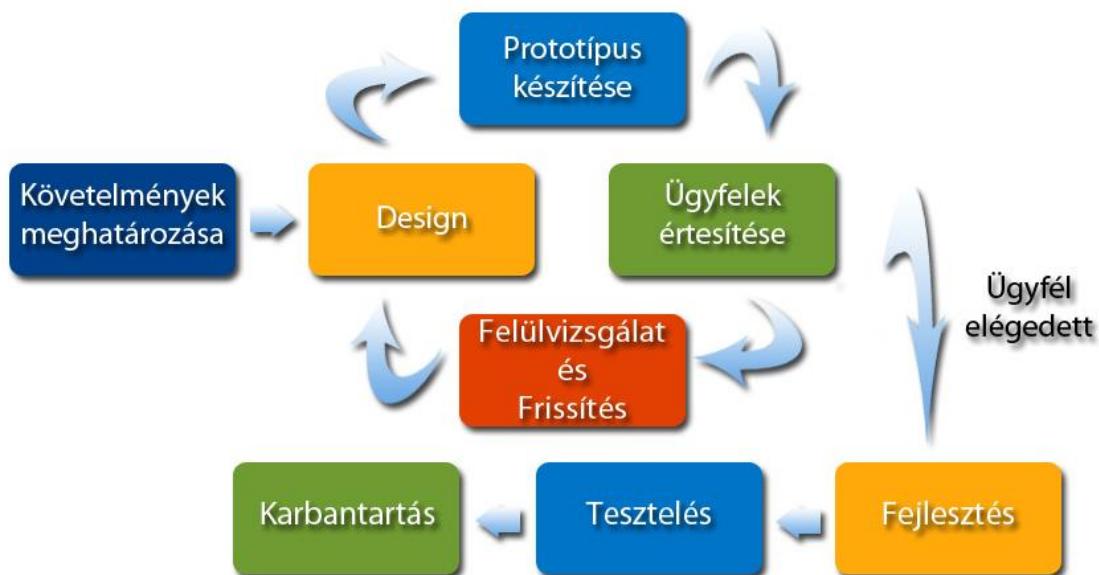
A prototípus alapú megközelítése a fejlesztésnek azon alapszik, hogy a megrendelő üzleti folyamatai, követelményei nem ismerhetők meg teljesen. Már csak azért sem, mert ezek az idővel változnak (lásd az agilis módszertanokat). A követelményeket érdemes finomítani prototípusok segítségével. Ha a felhasználó használatba vesz egy prototípust, akkor képes megfogalmazni, hogy az miért nem felel meg az elvárásainak és hogyan kellene megváltoztatni. Ebben a megközelítésben a leszállított rendszer is egy prototípus.

Ez a megközelítés annyira sikeres volt, hogy a modern módszertanok majd mindegyike prototípus alapú. Az iteratív módszerek általában minden mérföldkőhöz kötnek egy prototípust. Az agilis módszertanok akár minden nap új (lásd: napi fordítás) prototípust állítanak elő.

A kezdeti prototípus fejlesztése általában a következő lépésekkel áll:

- 1. lépés: Az alapkötetelmények meghatározása: Olyan alapkötetelmények meghatározása, mint a bemeneti és kimeneti adatok. Általában a teljesítményre vagy a biztonságra vonatkozó követelményekkel nem foglalkozunk.
- 2. lépés: Kezdeti prototípus kifejlesztése: Csak a felhasználói felületeket fejlesztjük le egy erre alkalmas CASE eszközzel. A mögötte lévő funkciókat nem, kivéve az új ablakok nyitását.
- 3. lépés: Bemutatás: A végfelhasználók megvizsgálják a prototípust, és jelzik, hogy mit gondolnak másként, illetve mit tennének még hozzá.
- 4. lépés. A követelmények pontosítása: A visszajelzéseket felhasználva pontosítjuk a követelményspecifikációt. Ha még mindig nem elég pontos a specifikáció, akkor a prototípust továbbfejlesztjük és ugrunk a 3. lépékre. Ha elég pontos képet kaptunk arról, hogy mit is akar a megrendelő, akkor az egyes módszertanok más és más írnak elő.

## Prototípus modell



6. ábra: A prototípusmodell

A prototípuskészítést akkor a legcélszerűbb használni, ha a rendszer és a felhasználó között sok lesz a párbeszéd. A modell on-line rendszerek elemzésében és tervezésében nagyon hatékony, különösen a tranzakció feldolgozásnál. Olyan rendszereknél, ahol kevés interakció zajlik a rendszer és a felhasználó között, ott kevésbé éri meg a prototípusmodell használata, ilyenek például a számításigényes feladatok. Különösen jól használható a felhasználói felület kialakításánál.

Tehát a prototípusmodell:

- általában iteratív,
- általában nem strukturált
- prototípus alapú,
- gyakran rapid, agilis, esetleg extrém,
- általában könnyűsúlyú, és
- követelményközpontú.

#### 2.3.5.1. [Fő változatai](#)

Eldobható prototípus (angolul: Throwaway prototyping): Az elnevezés arra utal, hogy az elkészült modellt úgymond eldobják az elemzés után, vagyis nem lesz része a végső kiadásnak. Arra használjuk ezt a prototípust, hogy a követelményeket minél gyorsabban, minél jobban fel tudjuk mérni, így a felhasználók is láthatják, hogy mire számíthatnak. Miután ez megtörtént, a prototípus eldobható, és a rendszer a prototípussal felderített követelményekre fog épülni. Ennek a változatnak a legnagyobb előnye, hogy a felhasználók gyorsan kipróbálhatják az első modellt, így újragondolhatják az előzetesen megadott követelményeket. Ezzel nagyon sok költséget és munkát meg lehet takarítani, mert rögtön az életciklus elején kiderülhet, hogy valamit máshogyan kellene csinálni. Ugyanakkor a gyors fejlesztés miatt nem szabad csodákat várni az eldobandó példánytól. A másik előnye, hogy a felhasználói felületet is hamar kialakíthatjuk, és azzal együtt tesztelhetik.

Ki kell hangsúlyozni, hogy ennek a változatnak csak akkor van értelme, ha az eldobható prototípus az életciklus elején gyorsan elkészül. A lényeg a gyorsaságon van. Ennek legszebb példája az extrém programozás (lásd később).

A felület kialakításának legismertebb módja:

- Kicsit valósághű eldobható prototípus vagy más néven papír prototípus: Papíron tollal rajzoljuk meg, hogy hogyan képzeljük el mégis a felületet.
- Nagyon valósághű eldobható prototípus: Egy grafikus felhasználófelület-szerkesztővel hozzuk létre a felületet, ami teljesen úgy néz ki, mint a végső kiadás, csak a funkciókból még semmi sem működik.

Evolúciós prototípus (angolul: Evolutionary prototyping): Az evolúciós prototípuskészítés jóval másabb, mint az előző változat. Itt már a fejlesztés elején egy nagyon robosztus prototípust kell kidolgozni, ami a lelke lesz a rendszernek, és a későbbiekben már csak finomítani kell rajta. Ugyanakkor itt is megvan a lehetősége a fejlesztőnek, hogy változtassa a beépített funkciókat. Az eldobható prototípussal szemben van egy előnye, mégpedig az, hogy itt már az elején egy működő rendszert kap a megrendelő elemzésre, még ha az összes követelmény nincs is beépítve a szoftverbe. Még az is előfordulhat, hogy a felhasználó már az egyik prototípust elkezdi használni a gyakorlatban, mert még

az is jobb, mint a semmi. Ennél a változatnál a fejlesztőknek nem kell az egész rendszert egyben fejleszteni. Megtehetik, hogy a rendszernek csak bizonyos részeire koncentrálnak. A kockázatok minimalizálása miatt a fejlesztők csak azt a részt fejlesztik a rendszernek, amit már tökéletesen megértettek, aztán a prototípust elküldik a megrendelőnek, aki dolgozik vele, és visszajelzéseket ad. Ha további követelmények tisztulnak le, akkor halad tovább a fejlesztés ezek megvalósításával.

### 2.3.5.2. Előnyei

Minőségnövelés: A prototípusmodell képes visszaszorítani a követelményspecifikációban lévő félreértések számát.

Költségcsökkenés: A prototípusmodell segít minél hamarabb kideríteni, hogy a megrendelő valójában mit is akar, és így csökkenti a költségeket, mert minél később derül fény egy félreértésre, annál többe kerülnek a változtatások.

Erősíti a felhasználó bevonását a fejlesztésbe: A prototípusmodellnél szükség van a felhasználók bevonására a fejlesztésbe. A prototípusok elemzésével már a fejlesztés közben jelzéseket tudnak adni a fejlesztők felé. Ezáltal kiderülhetnek az esetleges félreértések a fejlesztők és a megrendelő között, ugyanis a végfelhasználók tudják a legjobban, hogy mit is kellene csinálnia a szoftvernek. Így végül a termék magasabb minőségi szintet érhet el a fejlesztés végére, és valószínűleg jobban kielégíti a felhasználók igényeit is mindenféle tekintetben.

### 2.3.5.3. Hátrányai

Probléma az elemzés miatt: Ha a fejlesztők csak a prototípusra figyelnek, az elterelheti a figyelmüket a rendszer részletes analíziséről. Ezáltal lehet, hogy nem találják meg a legjobb megoldásokat és így a végző termék:

- teljesítménye gyenge lesz,
- nehezen lesz karbantartató,
- nehezen lesz skálázható.

Ezek a veszélyek csökkennek, ha már az első prototípusban egy rugalmas architektúrát alkalmazunk. Ennek ellenmond, hogy az architektúrát a nemfunkcionális követelmények alapján kell kialakítani, amiket lehet, hogy még nem ismerünk az első prototípus elkészülténél. Erre megoldás lehet keretrendszer alkalmazása, mint pl. a Spring keretrendszer vagy az ASP.NET MVC csomagja.

A prototípus- és a befejezettségi rendszer összekeveredése a felhasználók fejében: A felhasználók egy prototípus tesztelésekor azt hihetik, hogy ennek már közel úgy kéne működnie, mint a végleges rendszernek, és ezért rossz visszajelzéseket adhatnak a fejlesztőknek.

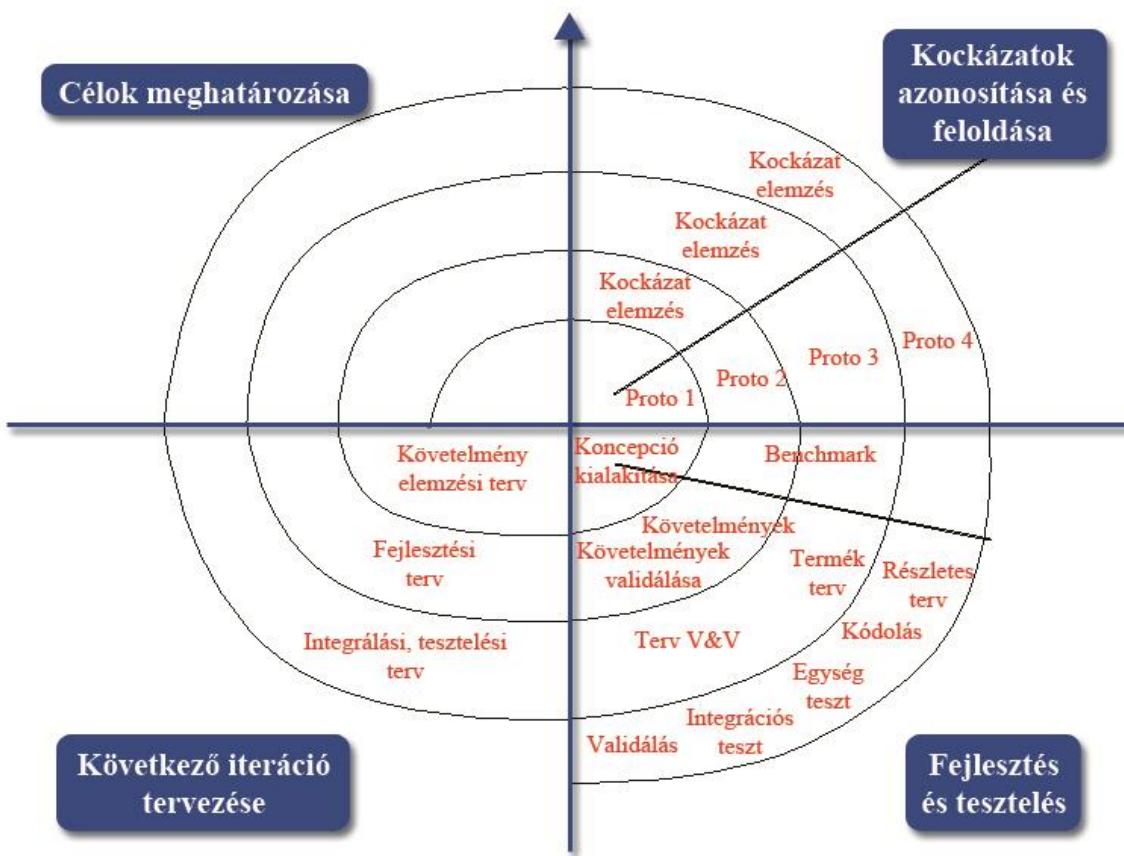
A fejlesztők ragaszkodása a prototípushoz: Előfordulhat, hogy miután a fejlesztők sok energiát fektettek egy prototípus elkészítésébe, túlságosan is ragaszkodnak hozzá, és abból akarnak egy végső kiadást készíteni. Még akkor is, ha nem is megfelelő a prototípus architektúrája. Ilyen szempontból jobb az eldobható prototípus alkalmazása az evolúcióossal szemben.

### 2.3.6. Spirálmódellek

A spirálmódellet Barry Boehm 1988-ban publikálta. A spirálmódellet a gyakorlatban csak elvétve használják, inkább az elvi jelentősége nagy. A modell a prototípusmóddel és a vízesésmóddal egyes

tulajdonságait kombinálja. Nagy, bonyolult, drága rendszerekhez ajánlott. A jelentőségét az adja, hogy ez az első modell, amely egyfajta iterációt használ. Habár a prototípusmodellben is van ciklus, de vagy csak a követelmények felderítésére szolgál, vagy az egész életciklus megismétlését jelenti. A spirálmodell esetén a spirál mindenkoron a négy fázison halad át:

- célok meghatározása,
- kockázatok azonosítása és feloldása,
- fejlesztés és tesztelés
- következő iteráció tervezése.



7. ábra: A spirálmodell

A fázisok kezdetén meg kell határozni a célokat és a fázis végére egy működőképes prototípust kell előállítani, úgy, hogy a prototípusok minden fázis után egyre inkább közelítsenek az elérődő végtermék felé, azaz evolúciós prototípus megközelítést használ. Nagyon fontos része a kockázati elemzés a modell minden fázisában, hiszen a megrendelő a prototípusra azt is mondhatja, hogy ez így nem jó és az elvégzett munkánk kárba veszhet. Az utolsó fázisban a spirál- és a vízeséses módszer nagyon hasonló, ugyanis ekkorra már a prototípusok készítésével pontosan tudjuk, hogy hogyan kell kinéznie, működnie a szoftver végleges verziójának és bár az elnevezések nem egyeznek meg, maguk a tevékenységek közel azonosak.

### 2.3.7. Iteratív és inkrementális módszertanok

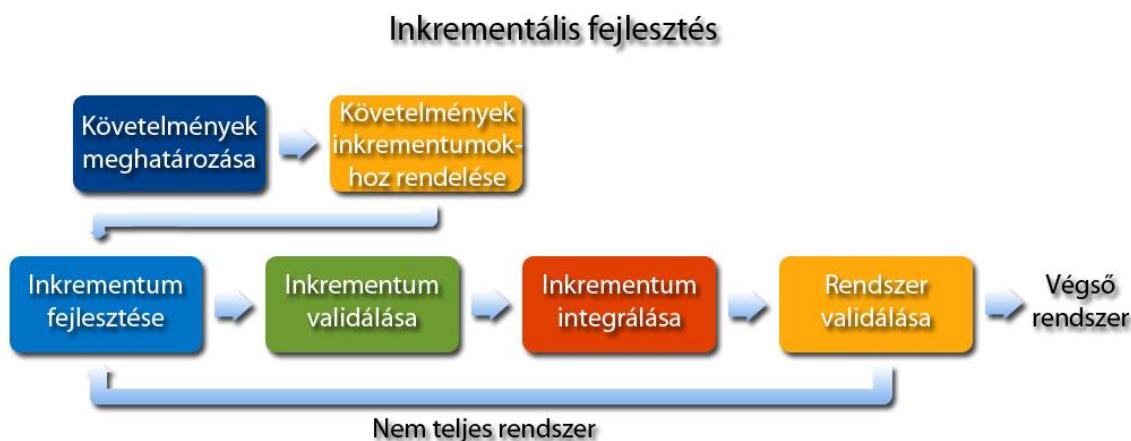
Az iteratív módszertan előírja, hogy a fejlesztést, kezdve az igényfelméréstől az üzemeltetésig, kisebb iterációk sorozatára bontsuk. Eltérően a vízesésmódlottól, amelyben például a tervezés teljesen megelőzi az implementációt, itt minden iterációban van tervezés és implementáció is. Lehet, hogy valamelyik iterációban az egyik sokkal hangsúlyosabb, mint a másik, de ez természetes.

A folyamatos finomítás lehetővé teszi, hogy mélyen megértsük a feladatot és felderítsük az ellentmondásokat. minden iteráció kiegészíti a már kifejlesztett prototípust. A kiegészítést inkrementumnak is nevezzük. Azokat a módszertanokat, amelyek a folyamatra teszik a hangsúlyt, azaz az iterációra, iteratív módszertanoknak nevezzük. Azokat, amelyek az iteráció termékére, az inkrementumra teszik a hangsúlyt, inkrementális módszertanoknak hívjuk. A mai módszertanok nagy része, kezdve a prototípusmódlottól egészen az agilis modellekig, ebbe a családba tartoznak.

Ezen módszertanok tulajdonságai:

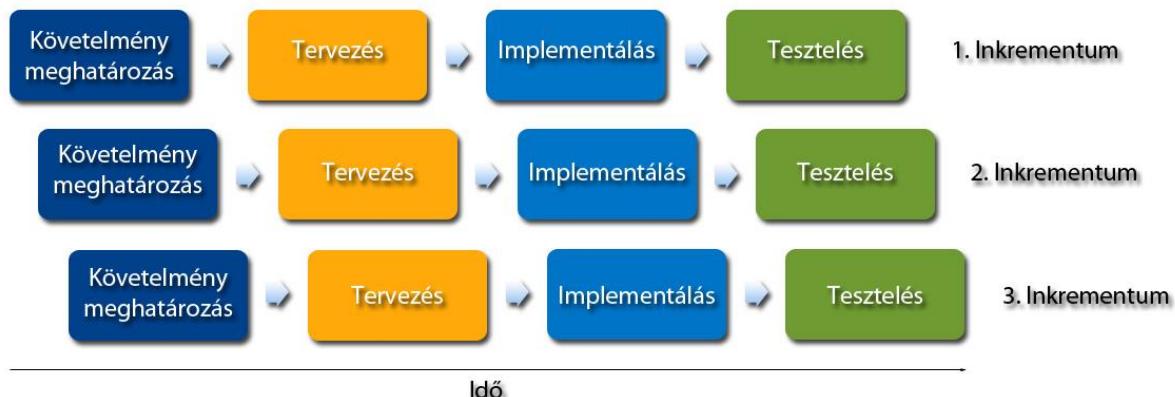
- iteratív (vagy inkrementális),
- általában objektumorientált,
- általában prototípus alapú, de van rapid, agilis, sőt extrém változata is,
- általában könnyűsúlyú,
- követelményközpontú vagy használatieset-központú.

A kiegészítés hozzáadásával növekvő részrendszer jön létre, amelyet tesztelni kell. Az új kódöt egységesztől teszteljük. Regressziós teszttel kell ellenőrizni, hogy a régi kód továbbra is működik-e az új kód hozzáadása és a változások után. Az új és a régi kód együttműködését integrációs teszttel teszteljük. Ha egy mérföldkőhöz vagy prototípus bemutatásához érkezünk, akkor van felhasználói átvételi teszt is. Egyébként csak egy belső átvételi teszt van az iteráció végén.



8. ábra: Az inkrementális fejlesztés

## Inkrementális fejlesztés



9. ábra: Az inkrementumok

Ezt a megközelítést több módszertan is alkalmazza, például a prototípusmodell, a gyors alkalmazásfejlesztés (RAD), a Rational Unified Process (RUP) és az agilis fejlesztési modellek. Itt ezeknek a módszertanoknak a közös részét, az iterációt ismertetjük. Egy iteráció a következő feladatokból áll:

- Üzleti folyamatok elemzése
- Követelményelemzés
- Elemzés és tervezés
- Implementáció
- Tesztelés
- Értékelés

Az iteratív modell fő ereje abban rejlik, hogy a szoftverfejlesztés életciklusának lépései nem egymás után jönnek, mint a strukturált módszertanok esetén, hanem időben átfedik egymást. minden iterációban van elemzés, tervezés, implementáció és tesztelés. Ezért, ha találunk egy félreérteést, akkor nem kell visszalépni, hanem néhány iteráció segítségével oldjuk fel a félreérteést. Ez az jelenti, hogy kevésbé tervezhető a fejlesztés ideje, de jól alkalmazkodik az igények változásához.

Mivel a fejlesztés lépéseinet mindig ismételgetjük, ezért azt mondjuk, hogy ezek időben átfedik egymást, hiszen minden szakaszban minden lépést végre kell hajtani. A kezdeti iterációkban több az elemzés, a végéhez közeledve egyre több a tesztelés. Már a legelső szakaszban is van tesztelés, de ekkor még csak a teszttervet készítjük. Már a legelső szakaszban is van implementáció, de ekkor még csak az architektúra osztályait hozzuk létre. És így tovább.

A feladatot több iterációra bontjuk. Ezeket általában több kisebb csapat implementálja egymással versengve. Aki gyorsabb, az választhat iterációt a meglévők közül. A választás nem teljesen szabad, a legnagyobb prioritású feladatok közül kell választani. A prioritás meghatározása különböző lehet, általában a leggyorsabban megvalósítható és legnagyobb üzleti értékű, azaz a legnagyobb üzleti megtérüléssel (angolul: return of investment) bíró feladat a legnagyobb prioritású.

**Üzleti folyamatok elemzése:** Első lépésben meg kell ismerni a megrendelő üzleti folyamatait. Az üzleti folyamatok modellezése során fel kell állítani egy projekt-fogalomtárat. A lemodellezett üzleti folyamatokat egyeztetni kell a megrendelővel, hogy ellenőrizzük, jól értjük-e az üzleti logikát. Ezt üzleti elemzők végezik, akik a megrendelők és a fejlesztők fejével is képesek gondolkozni.

**Követelményelemzés:** A követelmény elemzés során meghatározzuk a rendszer funkcionális és nemfunkcionális követelményeit, majd ezekből funkciókat, képernyőterveket készítünk. Ez a lépés az egész fejlesztés elején nagyon hangsúlyos, hiszen a kezdeti iterációk célja a követelmények felállítása. Későbbiekben csak a funkcionális terv finomítása a feladata. Fontos, hogy a követelményeket egyeztessük a megrendelőkkel. Ha a finomítás során ellentmondást fedezünk fel, akkor érdemes tisztázni a kérdést a megrendelővel.

**Elemzés és tervezés:** Az elemzés és tervezés során a követelményelemzés termékeiből megpróbáljuk elemzni a rendszert és megtervezni azt. A nemfunkcionális követelményekből lesz az architekturális terv. Az architekturális terv alapján tervezük az alrendszeret és a köztük levő kapcsolatokat. Ez a kezdeti iterációk feladata. A funkcionális követelmények alapján tervezük meg az osztályokat, metódusokat és az adattáblákat. Ezek a későbbi iterációk feladatai.

**Implementáció:** Az implementációs szakaszra ritkán adnak megszorítást az iteratív módszertanok. Általában a bevett technikák alkalmazását ajánlják, illetve szerepkörököt írnak elő. Pl.: a fejlesztők fejlesztik a rendszert, a fejlesztők szoros kapcsolatban vannak a tervezőkkel, továbbá van egy kódellenőr, aki ellenőrzi, hogy a fejlesztők által írt programok megfelelnek-e a tervezők által kitalált tervezési és programozási irányelveknek. Ebben a szakaszban a programozók egységesen biztosítják a kód minőségét.

**Tesztelés:** A tesztelési szakaszban különböző tesztelési eseteket találunk ki, ezekre tesztelési osztályokat tervezünk. Itt vizsgáljuk meg, hogy az elkészült kód képes-e együttműködni a program többi részével, azaz integrációs tesztet hajtunk végre. Regressziós tesztek segítségével ellenőrizzük, hogy ami eddig kész volt, az nem romlott el.

**Értékelés:** A fejlesztés minden ciklusában el kell dönteni, hogy az elkészült verziót elfogadjuk-e, vagy sem. Ha nem, akkor újra indul ez az iteráció. Ha igen, vége ennek az iterációnak. Az így elkészült kódot feltöljük a verziókövető rendszerbe, hogy a többi csapat is hozzáférjen. Az értékelés magában foglal egy átvételei tesztet is. Ha a megrendelő nem áll rendelkezésre, akkor általában a csoportok munkáját összefogó vezető programozó / tervező helyettesíti.

**Támogató tevékenységek, napi fordítás:** Az iterációktól függetlenül úgynévezett támogató folyamatok is zajlanak a szoftvercégen belül. Ilyen például a rendszergazdák vagy a menedzsment tevékenysége. Az iterációk szemszögéből a legfontosabb az úgynévezett a napi fordítás (angolul: daily build). Ez azt jelenti, hogy minden nap végén a verziókövető rendszerben lévő forráskódot lefordítjuk. minden csapat igyekszik a meglévő kódhoz igazítani a sajátját, hogy lehetséges legyen a fordítás. Aki elrontja a napi fordítást, és ezzel nehezíti az összes csapat következő napi munkáját, az büntetésre számíthat. Ez a cég hagyományaitól függ, általában egy hétag ő csinálja a napi fordítás és emiatt sokszor sokáig bent kell maradnia.

Végül vagy elérjük azt a pontot, ahol azt mondjuk, hogy ez így nem elkészíthető, vagy azt mondjuk, hogy minden felmerült igényt kielégít a szoftverünk és szállíthatjuk a megrendelőnek.

### 2.3.8. Rational Unified Process – RUP

A Rational Unified Process, vagy röviden RUP, egy iteratív szoftvertervezési módszertan. Megmutatja, hogy az egyes tervezési szakaszokban milyen feladatok vannak és ezeket a feladatokat milyen képzettségű embereknek kell kiosztani ahhoz, hogy a projektet sikeresen elemezzük, megtervezzük, implementáljuk és teszteljük. A módszertan célja az, hogy egy minőségi terméket tudunk előállítani minél hamarabb, minél hatékonyabban és ellenőrizhető módon.

A RUP alapja az iteráció. Azaz a különböző szakaszokban ugyanazokkal a területekkel foglalkozunk, egyre finomabb megoldáshoz jutva. Az iterációk által könnyebben alkalmazkodhatunk az új követelményekhez és hamarabb tudjuk azonosítani és megoldani a projekt kockázatait.

A RUP használatieset-alapú. Szinte minden tervezési fázisban viszontlátjuk a használati eset fogalmát különböző nézetekből. A módszertan objektumorientált alapú és szokásos modellező nyelve az UML.

A módszertan konfigurálhatóságon alapszik. Alapelve az, hogy nem lehet egy általános leírást adni arra, hogy hogyan kell rendszertervet csinálni minden projektre. Ki lehet venni részleteket a módszertanból attól függően, hogy a projekt megköveteli-e ezt vagy sem.

A RUP több feladatra bontja a projektfejlesztési folyamatot (angolul: Core Process Workflows):

- Üzleti folyamatok elemzése (angolul: Business Modeling)
- Követelményelemzés (angolul: Requirements)
- Elemzés és tervezés (angolul: Analysis & Design)
- Implementáció (angolul: Implementation)
- Tesztelés (angolul: Test)
- Átadással kapcsolatos tevékenységek (angolul: Deployment)

Minden tevékenység minden iterációban előfordul, de különböző súlyval. Kezdetben az üzleti folyamatok elemzése a hangsúlyos, majd ez eltolódik követelményelemzés felé, és így tovább. Így az életciklus lépései nem egymást követik, hanem átfedik egymást. Könnyen belátható például, hogy sokkal hasznosabb már az implementáció alatt tesztelni, mint utána.

Ezen túl nevesíti a támogató tevékenységeket is (angolul: Core Supporting Workflows):

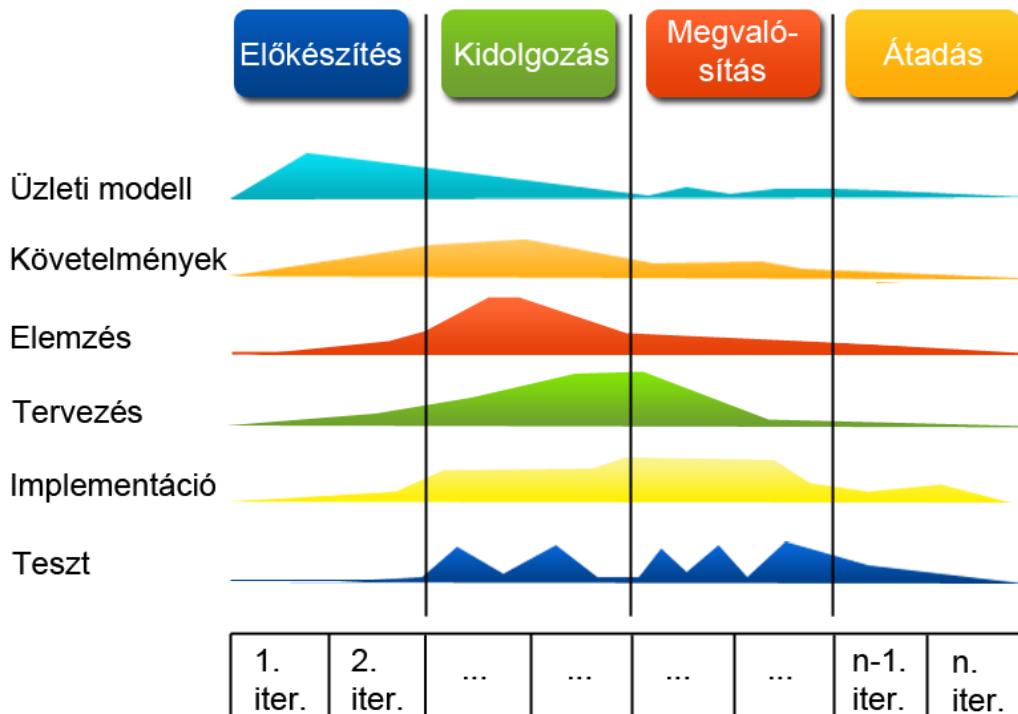
- Konfigurációmenedzsment és változás követés (angolul: Configuration & Change Management)
- Projektvezetés (angolul: Project Management)
- Fejlesztői környezet felállítása (angolul: Environment)

Illetve több szakaszra (angolul: Phases) bontja az életciklust:

- Előkészítés (angolul: Inception)
- Kidolgozás (angolul: Elaboration)
- Megvalósítás (angolul: Construction)

- Átadás (angolul: Transition)

Minden szakaszban minden feladatból valamennyit el kell végezni. Ezt az alábbi ábra szemlélteti:



10. ábra: A RUP módszertan fázisai

### 2.3.9. Gyors alkalmazásfejlesztés – RAD

A gyors alkalmazásfejlesztés vagy ismertebb nevén RAD (angolul: Rapid Application Development) egy olyan elgondolás, amelynek lényege a szoftver gyorsabb és jobb minőségű elkészítése. Ezt a következők által érhetjük el:

- Korai prototípuskészítés és ismétlődő felhasználói átvételi tesztek.
- A csapat - megrendelő és a csapaton belüli kommunikációban kevésbé formális.
- Szigorú ütemterv, így az újítások mindenkor csak a termék következő verziójában jelennek meg.
- Követelmények összegyűjtése fókuszcsoportok és munkaértekezletek használatával.
- Komponensek újrahasznosítása.

Ezekhez a folyamatokhoz több szoftvergyártó is készített segédeszközöket, melyek részben vagy egészben lefedik a fejlesztés fázisait, mint például:

- követelményösszegyűjtő eszközök,
- tervezést segítő eszközök,
- prototípuskészítő eszközök,
- csapatok kommunikációját segítő eszközök.

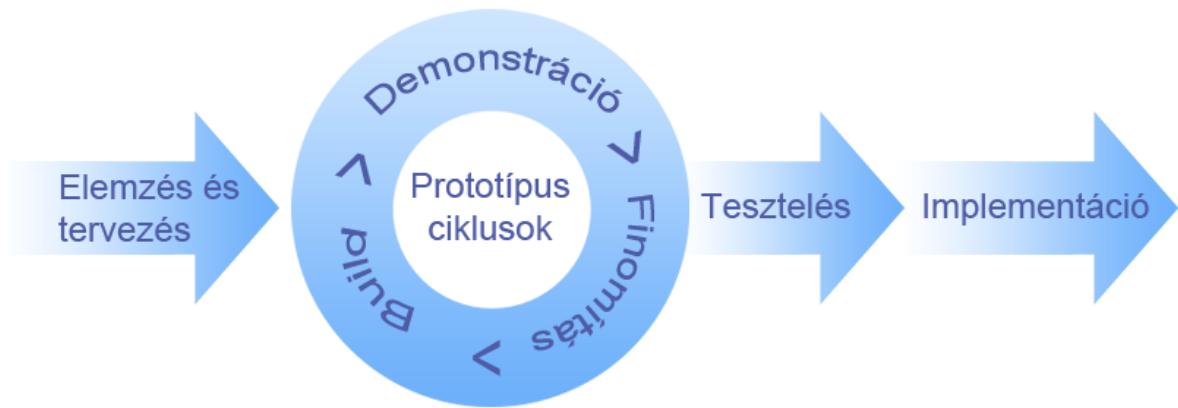
A RAD elsősorban az objektumorientált programozással kacsolódik össze, már csak a komponensek újrahasznosítása okán is. Összehasonlítva a hagyományos fejlesztési metódusokkal (pl.: vízesésmodell), ahol az egyes fejlesztési fázisok jól elkülönülnek egymástól, és szigorúan tilos a visszalépés, a RAD sokkal rugalmasabb. Gyakori probléma, hogy a tervezésbe hiba csúszik, és az csak a megvalósítási vagy a tesztelési fázisban jön elő, ráadásul az elemzés- és a tesztelési fázis között hat-hét hónap is eltelhet. Vagy ha menet közben megváltoznak az üzleti körülmények, és már a megvalósítási fázisban járunk, vagy csak rájöttek a megrendelők, hogy valamit mégis másképpen szeretnének, akkor szintén gondban vagyunk. A RAD válasza ezekre a problémákra a gyorsaság. Ha gyorsan hozzuk létre a rendszert, akkor ezen rövid idő alatt nem változnak a követelmények, az elemzés és tesztelés között nem hat-hét hónap, hanem csak hat-hét hét telik el.

A gyorsaság eléréséhez sok meglévő komponenst kell felhasználni, amit a csapatnak jól kell ismernie. A komponensek lehetnek saját fejlesztésűek vagy megvásároltak. Komponenst vásárolni nagy kockázat, mert ha hiba van benne, azt nem tudjuk javítani, ha nem kapjuk meg a forrást, de még úgy is nagyon nehéz. Ezért a komponensgyártók nagyon alaposan tesztelik termékeiket, illetve gyors hibajavítást vállalnak.

A RAD az elemzést, a tervezést, a megvalósítást és a tesztelést rövid, ismétlődő ciklusok sorozatába tömöríti, és ennek sok előnye van a hagyományos modellekkel szemben. A fejlesztés során általában kis csoportokat hoznak létre fejlesztőkből, végfelhasználókból, ez az úgynevezett fókuszcsoport. Ezek a csapatok az ismétlődő, rövid ciklusokkal vegyítve hatékonyabbá teszik a kommunikációt, optimalizálják a fejlesztési sebességet, egységesítik az elképzeléseket és célokat, valamint leegyszerűsítik a folyamat felügyeletét.

Öt fejlesztési lépés a RAD-ban:

- **Üzleti modellezés:** Az üzleti funkciók közötti információáramlást olyan kérdések feltevésével tudjuk felderíteni, mint hogy milyen információk keletkeznek, ezeket ki állítja elő, az üzleti folyamatot milyen információk irányítják, vagy hogy ki irányítja.
- **Adatmodellezés:** Az üzleti modellezéssel összegyűjtöttük a szükséges adatokat, melyekből adatobjektumokat hozunk létre. Beazonosítjuk az attribútumokat és a kapcsolatokat az adatok között.
- **Folyamatmodellezés:** Az előzőleg létrehozott adatmodellhez szükséges műveletek (bővítés, törlés, módosítás) meghatározása, úgy, hogy létrehozzuk a kellő információáramlást az üzleti funkciók számára.
- **Alkalmazás előállítása:** A szoftver előállításának megkönnyítése automatikus eszközökkel.
- **Tesztelés:** Az új programkomponensek tesztelése, a már korábban tesztelt komponenseket már nem szükséges újra vizsgálni. Ez gyorsítja a folyamatot. Ugyanakkor ez természetesen nem vonatkozik az integrációs és a rendszertesztre. Az új komponens és a régi komponensek együttműködését integrációs tesztekkel kell vizsgálni, valamint a rendszertesztet is meg kell ismételni, természetesen az új komponenssel kapcsolatos területekre helyezve a hangsúlyt.



11. ábra: A RAD módszertan

Tehát a RAD módszertan:

- általában iteratív,
- objektum orientált,
- rapid,
- könnyűsúlyú
- követelmény központú sok megrendelő – csapat kommunikációval.

Hátránya, hogy magasan képzett fejlesztőkre van szükség, emellett fontos a fejlesztők és a végfelhasználók elkötelezettsége a sikeres szoftver iránt. Ha a projekt nehezen modularizálható, akkor nem a legjobb választás a RAD. Nagyobb rendszerek fejlesztése ezzel a módszertannal kockázatos.

#### 2.3.10. Agilis szoftverfejlesztés

Az agilis szoftverfejlesztés valójában iteratív szoftverfejlesztési módszerek egy csoportjára utal, amelyet 2001-ben az Agile Manifesto nevű kiadványban öntöttek formába. Az agilis fejlesztési módszerek (nevezik adaptívnak is) egyik fontos jellemzője, hogy a résztvevők, amennyire lehetséges, megpróbálnak alkalmazkodni a projekthez. Ezért fontos például, hogy a fejlesztők folyamatosan tanuljanak.

Az agilis szoftverfejlesztés szerint értékesebbek:

- az egyének és az interaktivitás szemben a folyamatokkal és az eszközökkel,
- a működő szoftver szemben a terjedelmes dokumentációval,
- az együttműködés a megrendelővel szemben a szerződéses tárgyalásokkal,
- az alkalmazkodás a változásokhoz szemben a terv követésével.

Az agilis szoftverfejlesztés alapelvei:

- A legfontosabb a megrendelő kielégítése használható szoftver gyors és folyamatos átadásával.
- Még a követelmények kései változtatása sem okoz problémát.
- A működő szoftver / prototípus átadása rendszeresen, a lehető legrövidebb időn belül.
- Napi együttműködés a megrendelő és a fejlesztők között.

- A projektek motivált egyének köré épülnek, akik megkapják a szükséges eszközöket és támogatást a legjobb munkavégzéshez.
- A leghatékonyabb kommunikáció a szemtől-szembeni megbeszélés.
- Az előrehaladás alapja a működő szoftver.
- Az agilis folyamatok általi fenntartható fejlesztés állandó ütemben.
- Folyamatos figyelem a technikai kitűnőségeknek.
- Egyszerűség a minél nagyobb hatékonyságért.
- Önszervező csapatok készítik a legjobb terveket.
- Rendszeres időközönként a csapatok reagálnak a változásokra, hogy még hatékonyabbak legyenek.

Az agilis szoftverfejlesztésnek nagyon sok fajtája van. Ebben a jegyzetben csak ezt a kettőt tárgyaljuk:

- Scrum
- Extrém Programozás (XP)

Ezek a következő közös jellemzőkkel bírnak:

- Kevesebb dokumentáció.
- Növekvő rugalmasság, csökkenő kockázat.
- Könnyebb kommunikáció, javuló együttműködés.
- A megrendelő bevonása a fejlesztésbe.

**Kevesebb dokumentáció:** Az agilis metódusok alapvető különbsége a hagyományosakhoz képest, hogy a projektet apró részekre bontják, és minden egy kisebb darabot tesznek hozzá a termékhez, ezeket egytől négy héig terjedő ciklusokban (más néven keretekben vagy idődobozokban) készítik el, és ezek a ciklusok ismétlődnek. Ezáltal nincs olyan jellegű részletes hosszútávú tervezés, mint például a vízeséses modellnél, csak az a minimális, amire az adott ciklusban szükség van. Ez abból az elvből indul ki, hogy nem lehet előre tökéletesen, minden részletre kiterjedően megtervezni egy szoftvert, mert vagy a tervben lesz hiba, vagy a megrendelő változtat valamit.

**Növekvő rugalmasság, csökkenő kockázat:** Az agilis módszerek a változásokhoz adaptálható technikákat részesítik előnyben a jól tervezhető technikákkal szemben. Ennek megfelelően iterációkat használnak. Egy iteráció olyan, mint egy hagyományos életciklus: tartalmazza a tervezést, a követelmények elemzését, a kódolást és a tesztelést. Egy iteráció maximum egy hónap terjedelmű, így nő a rugalmasság, valamint csökken a kockázat, hiszen az iteráció végén átvételi teszt van, ami után a megrendelő megváltoztathatja eddigi követelményeit. minden iteráció végén futóképes változatot kell kiadniuk a csapatoknak a kezüköből.

**Könnyebb kommunikáció, javuló együttműködés:** Jellemző, hogy a fejlesztő csoportok önszervezők, és általában nem egy feladatra specializálódottak a tagok, hanem többféle szakterületről kerülnek egy csapatba, így például programozók és tesztelők. Ezek a csapatok ideális esetben egy helyen, egy irodában dolgoznak, a csapatok mérete ideális esetben 5-9 fő. Mindez leegyszerűsíti a tagok közötti kommunikációt és segíti a csapaton belüli együttműködést. Az agilis módszerek előnyben részesítik a szemtől szembe folytatott kommunikációt az írásban folytatott eszmecserével szemben.

A megrendelő bevonása a fejlesztésbe: Vagy személyesen a megrendelő vagy egy kijelölt személy, aki elkötelezi magát a termék elkészítése mellett, folyamatosan a fejlesztők rendelkezésére áll, hogy a menet közben felmerülő kérdéseket minél hamarabb meg tudja válaszolni. Ez a személy a ciklus végén is részt vesz az elkészült prototípus kiértékelésében. Fontos feladata az elkészítendő funkciók fontossági sorrendjének felállítása azok üzleti értéke alapján. Az üzleti értékből és a fejlesztő csapat által becsült fejlesztési időből számolható a befektetés megtérülése (Return of Investment, ROI). A befektetés megtérülése az üzleti érték és a fejlesztési idő hányadosa.

Tehát az agilis módszertan:

- elfogadja a gyakori változást,
- iteratív,
- gyakran objektumorientált,
- prototípus alapú,
- rapid,
- esetleg extrém,
- könnyűsúlyú,
- követelmény- és csapatközpontú.

Az agilis módszertanok nagyon jól működnek, amíg a feladatot egy közepes méretű (5-9 fős) csapat képes megoldani. Nagyobb csoportok esetén nehéz a csapatszellem kialakítása. Ha több csoport dolgozik ugyanazon a célon, akkor köztük a kommunikáció nehézkes. Ha a megrendelő nem hajlandó egy elkötelezetted munkatársát a fejlesztő csapat rendelkezésére bocsátani, akkor az kiváltható egy üzleti elemzővel, aki átlátja a megrendelő üzleti folyamatait, de ez kockázatos.

#### 2.3.11. Extrém programozás

Az extrém programozás (angolul: Extreme Programming, vagy röviden: XP) egy agilis módszertan. A nevében az extrém szó onnan jön, hogy az eddigi módszertanokból átveszi a jól bevált technikákat és azokat nem csak jól, hanem extrém jól alkalmazza, minden más feleslegesnek tekint. Gyakran összekeverik a „programozunk összesig” módszerrel, amivel egy-két 24 órás vagy akár 48 órás programozóversenyen találkozhatunk.

Az extrém programozás 4 tevékenységet ír elő. Ezek a következők:

- Kódolás: A forráskód a projekt legfontosabb terméke, ezért a kódolásra kell a hangsúlyt helyezni. Igazán kódolás közben jönnek ki a feladat nehézségei, hiába gondoltuk azt át előtte. A kód a legalkalmasabb a két programozó közötti kommunikációra, mivel azt nem lehet kétféleképpen érteni. A kód alkalmas a programozó gondolatainak kifejezésére.
- Tesztelés: Addig nem lehetünk benne biztosak, hogy egy funkció működik, amíg nem teszteltük. Az extrém felfogás szerint kevés tesztelés kevés hibát talál, extrém sok tesztelés megtalálja minden. A tesztelés játsza a dokumentáció szerepét. Nem dokumentáljuk a metódusokat, hanem egységeszteket fejlesztünk hozzá. Nem készítünk követelményspecifikációt, hanem átvételi teszteseteket fejlesztünk a megértett követelményekből.

- Odafigyelés: A fejlesztőknek oda kell figyelniük a megrendelőkre, meg kell érteniük az igényeket. El kell magyarázni nekik, hogy hogyan lehet technikailag kivitelezni ezeket az igényeket, és ha egy igény kivitelezhetetlen, ezt meg kell érteni a megrendelővel.
- Tervezés: Tervezés nélkül nem lehet szoftvert fejleszteni, mert az ad hoc megoldások átláthatatlan struktúrához vezetnek. Mivel fel kell készülni az igények változására, ezért úgy kell megtervezni a szoftvert, hogy egyes komponensei amennyire csak lehet függetlenek legyenek a többiből. Ezért érdemes pl. objektumorientált tervezési alapelveket használni.

Az extrém programozás 5 értéket vall. Ezek a következők:

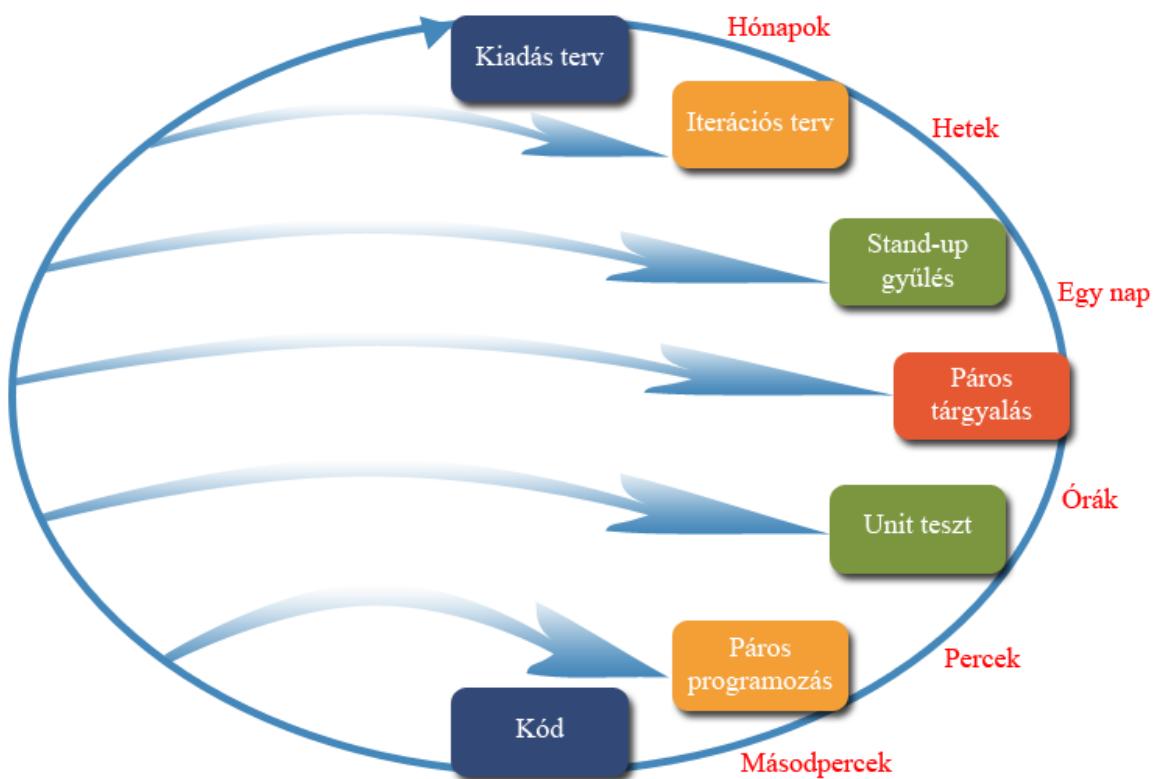
- Kommunikáció: Egy fejlesztőcsapat akkor tud hatékonyan működni, ha megbeszélik a terveket, a felmerülő problémákat és megoldásokat. Más módszertanokban ezt helyettesíti a dokumentáció. Az extrém programozás szerint a dokumentumokat nem olvassák el az emberek, ezért jobb, ha mindenki szóban osztja meg a tudását a kollégáival.
- Egyszerűség: Az extrém programozás azt ajánlja, hogy minden funkciót mindenki szóban osztja meg a kollégáival. Ez magában hordozza azt a lehetőséget, hogy egy-egy új funkció bevezetéséhez újra kell írni a rendszert, de ezt a lehetőséget kompenzálgja egy másik lehetőség. Lehet, hogy sok időt kellene eltölteni egy minden funkció megoldásához szükséges terv kidolgozásával egy problémás funkció miatt, amikor később kiderülhet, hogy a problémás funkcióra nincs is szükség.
- Visszacsatolás: Többféle visszacsatolásról beszélhetünk, amik alapján javíthatjuk a rendszert:
  - Visszacsatolás a rendszer felől: Mivel sok egységesztünk van, és folyamatos integrációs tesztet végezünk, ezért a programozók tudják, milyen állapotban van a kód, mennyi hiba van benne. Az elfogadási tesztek alapján lehet tudni, milyen messze vagyunk a kiszállítható megoldástól.
  - Visszacsatolás a megrendelő felől: A gyakori prototípus-bemutatások alkalmával a megrendelő elmondhatja, hogy mit vár másképp, illetve mit vár még el a rendszertől.
  - Visszacsatolás a csapat felől: Ha a megrendelőnek van egy új igénye, akkor az úgynevezett Planning Game alkalmával a csapat elmondja, hogy ennek mennyi a becsült kifejlesztési ideje.
- Bátorság: Sok extrém módszer igényel bátorságot. Például a „programozz a mának, ne a holnapnak” azt feltételezi, hogy elég bátrak vagyunk a legegyszerűbb megoldást választani, amikor lehet, hogy holnap az egészet újra kell írnunk egy új követelmény miatt. Bátorság kell a meglévő kód szépítéséhez is, hiszen gyakran valaki másnak a kódját kell szépíteni.
- Tisztelet: A csapat tagjai tisztelik egymást, mások és a saját munkáját. Ennek megfelelően nem szabad olyan kódot feltölteni a verziókövető rendszerbe, ami nem megy át egységesztéken, hiszen ekkor mindenki más elkezdi keresni, hogy mi lehet a hiba. A saját kódját az ember azzal tiszteli, hogy a legnagyobb minőségre, átláthatóságra, könnyen érthetőségre törekszik. A csapat egyik tagja sem érezheti magát másodrendűnek vagy megtűrtnek. Ez segíti, hogy a csapat tagjai motiváltak legyenek.

Néhány extrém programozásra jellemző technika:

- Páros programozás (angolul: pair programming): Két programozó ír egy kódot, pontosabban az egyik írja, a másik figyeli. Ha a figyelő hibát lát vagy nem érzi a kódot, akkor azonnal szól. A

két programozó folyamatosan megbeszéli, hogy hogyan érdemes megoldani az adott problémát.

- Tesztvezérelt fejlesztés (angolul: test driven development): Már a metódus elkészítése előtt megírjuk a hozzá tartozó egységesítéseket, vagy más néven unit-teszteket. Ezt néha hívják „először a teszt” (angolul: test-first) megközelítésnek is.
- Forráskód átnézése (angolul: code review): Az elkészült nagyobb modulokat, pl. osztályokat, egy vezető fejlesztő átnézi, hogy van-e benne hiba, nem érhető, nem dokumentált rész. A modul fejlesztői elmagyarázzák, mit és miért csináltak. A vezető fejlesztő elmondja, hogyan lehet ezt jobban, szebben csinálni.
- Folyamatos integráció (angolul: continuous integration): A nap (vagy a hét) végén, a verziókövető rendszerbe bekerült kódokat integrációs teszt alá vetjük, hogy kiderüljön, hogy azok képesek-e együttműködni. Így nagyon korán kiszűrhető a programozók közti félreérzés.
- Kódszépítés (angolul: refactoring): A már letesztelt, működő kódban lehet szépíteni, az esetleg lassú, rugalmatlan, vagy egyszerűen csak csúnya részeket. A kódszépítés előfeltétele, hogy legyen sokegységesítés. A szépítés során nem szabad megváltoztatni a kód funkcionálitását, de a szerkezet, pl. egy metódus törzse, szabadon változtatható. A szépítés után minden egységesítést le kell futtatni, nem csak a megváltozott kódhoz tartozókat, hogy lássuk, a változások okoztak-e hibát.



12. ábra: Az XP jól bevált módszerei

Az extrém programozás akkor működik jól, ha a megrendelő biztosítani tud egy munkatársat, aki átlátja a megrendelő folyamatait, tudja, mire van szükség. Ha a változó, vagy a menet közben kiderített követelmények miatt gyakran át kell írni már elkészült részeket, akkor az extrém programozás nagyon rossz választás. Kezdő programozók esetén az extrém programozás nem alkalmazható, mert nincs elég tapasztalatuk az extrém módszerek alkalmazásához.

Az extrém programozás legnagyobb erénye, hogy olyan fejlesztési módszereket hozott a felszínre, amik magas minőséget biztosítanak. Ezek, mint pl. a páros programozás, nagyon népszerűek lettek.

### 2.3.12. Scrum

A Scrum egy agilis szoftverfejlesztési metódus. Jellegzetessége, hogy fogalmait a rugby nevű csapatjátékból meríti. Ilyen fogalom maga a scrum is, amely dulakodást jelent. A módszertan jelentős szerepet tulajdonít a csoporton belüli összetartásnak. A csoporton belül sok a találkozó, a kommunikáció, lehetőség van a gondok megbeszélésre is. Az ajánlás szerint jó, ha a csapat egy helyen dolgozik és szóban kommunikál.

A Scrum által előírt fejlesztési folyamat röviden így foglalható össze: A Product Owner létrehoz egy Product Backlog-ot, amelyre a teendőket felhasználói sztoriként veszi fel. A sztorikat prioritással kell ellátni és megmondani, mi az üzleti értékük. Ez a Product Owner feladata. A Sprint Planning Meetingen a csapat tagjai megbeszélik, hogy mely sztorik megvalósítását vállalják el, lehetőleg a legnagyobb prioritásúakat. Ehhez a sztorikat kisebb feladatokra bontják, hogy megbecsülhessék mennyi ideig tart megvalósítani azokat. Ezután jön a sprint, ami 2-4 hétag tart. A sprint időtartamát az elején fixálja a csapat, ettől eltérni nem lehet. Ha nem sikerül befejezni az adott időtartam alatt, akkor sikertelen a sprint, ami büntetést, általában prémiummegvonást von maga után. A sprinten belül a csapat és a Scrum Master naponta megbeszélik a történteket a Daily Meetingen. Itt mindenki elmondja, hogy mit csinált, mi lesz a következő feladata, és milyen akadályokba (impediment) ütközött. A sprint végén következik a Sprint Review, ahol a csapat bemutatja a sprint alatt elkészült sztorikat. Ezeket vagy elfogadják, vagy nem. Majd a Sprint Retrospective találkozó következik, ahol a Sprint során felmerült problémákat tárgyalja át a csapat. A megoldásra konkrét javaslatokat kell tenni. Ezek után újra a Sprint Planning Meeting következik. A fejlesztett termék azelőtt piacra kerülhet, hogy minden sztorit megvalósítottak volna.

A csapatban minden szerepkör képviselője megtalálható, így van benne fejlesztő és tesztelő is. Téves azt gondolni, hogy a sprint elején a tesztelő is programot ír, hiszen, amíg nincs program, nincs mit tesztelni. Ezzel szemben a tesztelő a sprint elején teszttervet készít, majd kidolgozza a teszeseteket, végül, amikor már vannak kész osztályok, egységeszteket ír, a változásokat regressziós tesztel ellenőrzi.

A Scrum, mint minden agilis módszertan, arra épít, hogy a fejlesztés közben a megrendelő igényei változhatsanak. A változásokhoz úgy alkalmazkodik, hogy a Product Backlog folyamatosan változhat. Az erre épülő dokumentumok folyamatosan finomodnak, tehát könnyen változtathatók. A csapatok gyorsan megvalósítják a szükséges változásokat.

A Scrum tökélyre viszi az egy csapaton belüli hatékonyságot. Ha több csapat is dolgozik egy fejlesztésen, akkor köztük lehetnek kommunikációs zavarok, ami a módszertan egyik hátránya.



13. ábra: A SCRUM módszertan

A Scrum két nagyon fontos fogalma a sprint és az akadály.

**Sprint (vagy futam):** Egy előre megbeszélt hosszúságú fejlesztési időszak, általában 2-4 hétag tart, kezdődik a Sprint Planning-gel, majd a Retrospective-vel zárul. Ez a Scrum úgynevezett iterációs ciklusa, addig kell ismételni, amíg a Product Backlog-ról el nem tűnnek a megoldásra váró felhasználói sztorik. Alapelvek, hogy minden sprint végére egy potenciálisan leszállítható szoftvert kell előállítani a csapatnak, azaz egy prototípust. A sprint tekintethető két mérföldkő közti munkának.

**Akadály (angolul: Impediment):** Olyan gátló tényező, amely a munkát hátrálhatja. Csak és kizárolag munkahelyi probléma tekinthető akadálynak. A csapattagok magánéleti problémái nem azok. Akadály például, hogy lejárt az egyik szoftver licence, vagy szükség lenne egy plusz gépre a gyorsabb haladáshoz, vagy több memóriára az egyik géphez, vagy akár az is lehet, hogy 2 tag megsértődött egymásra. Ilyenkor kell a Scrum Masternek elhárítani az akadályokat, hogy a munka minél gördülékenyebb legyen.

Tehát a Scrum módszertan:

- agilis,
- iteratív (ahol az iteráció a sprint),
- objektum orientált vagy szerviz orientált programozási megoldásokat részesít előnyben,
- prototípus alapú, rapid és agilis,
- csapatközpontú,
- könnyűsúlyú (angolul: lightweight) módszertan.

A módszertan szerepköröket, megbeszéléseket és elkészítendő termékeket ír elő.

#### 2.3.12.1. Szerepkörök

A módszertan kétféle szerepkört különböztet meg, ezek a disznók és a csirkék. A megkülönböztetés alapja egy vicc:

A disznó és a csirke mennek az utcán. Egyszer csak a csirke megszólal: „Te, nyissunk egy éttermet!” Mire a disznó: „Jó ötlet, mi legyen a neve?” A csirke gondolkozik, majd rávágja: „Nevezzük Sonkástojásnak!” A disznó erre: „Nem tetszik valahogy, mert én biztosan minden beleadnék, te meg éppen csak hogy részt vennél benne.”

A disznók azok, akik elkötelezettek a szoftverprojekt sikereben. Ők azok, akik a „vérüket” adják a projekt sikereért, azaz felelősséget vállalnak érte. A csirkék is érdekeltek a projekt sikereben, ők a haszonelvezői a sikernek, de ha esetleg mégse sikeres a projekt, akkor az nem az ő felelősségük.

Disznók:

- Scrum mester (Scrum Master)
- Csapat (Team)

Csirkék:

- Terméktulajdonos (Product Owner)
- Üzleti szereplők (Stakeholders)
- Menedzsment (Managers)

*Scrum mester (Scrum Master):* A Scrum mester felügyeli és megkönnyíti a folyamat fenntartását, segíti a csapatot, ha problémába ütközik, illetve felügyeli, hogy mindenki betartja-e a Scrum alapvető szabályait. Ilyen például, hogy a Sprint időtartama nem térhet el az előre megbeszéltől, még akkor sem, ha az elvállalt munka nem lesz kész. Akkor is nemet kell mondania, ha a Product Owner a sprint közben azt találja ki, hogy az egyik sztorit, amit nem vállaltak be az adott időszakra, el kellene készíteni, mert mondjuk megváltoztak az üzleti körülmények. Lényegében ő a projektmenedzser.

*Csapat (Team):* Ők a felelősek azért, hogy az aktuális sprintre bevállalt feladatokat elvégezzék, ideális esetben 5-9 fő alkot egy csapatot. A csapatban helyet kapnak a fejlesztők, tesztelők, elemzők. Így nem a váltófutásra jellemző stafétaváltás (mint a vízesés modellnél), hanem a futballra emlékeztető passzolgatás, azaz igazi csapatjáték jellemzi a csapatot.

*Terméktulajdonos (Product Owner):* A megrendelő szerepét tölti be, ő a felelős azért, hogy a csapat mindenkor azonosítja a termék funkcióit, amelyeket a felhasználók számára fontosnak tartanak. A Product Owner és a Scrum Master nem lehet ugyanaz a személy.

*Üzleti szereplők, pl.: megrendelők, forgalmazók, tulajdonosok (Stakeholders):* A megrendelő által jön létre a projekt, ő az, aki majd a hasznát látja a termék elkészítésének, a Sprint Review során kap szerepet a folyamatban.

*Menedzsment (Managers):* A menedzsment feladata a megfelelő környezet felállítása a csapatok számára. Általában a megfelelő környezeten túl a lehető legjobb környezet felállítására törekszenek.

### 2.3.12.2. Megbeszélések

*Sprint Planning Meeting (futamtervező megbeszélés):* Ezen a találkozón kell megbeszálni, hogy ki mennyi munkát tud elvállalni, majd ennek tudatában dönti el a csapat, hogy mely sztorikat vállalja be a következő sprintre. Emellett a másik lényeges dolog, hogy a csapat a Product Owner-rel megbeszéli,

majd teljes mértékben megérte, hogy a vevő mit szeretne az adott sztoritól, így elkerülhetőek az esetleges félreértésekkel adódó problémák. Ha volt Backlog Grooming, akkor nem tart olyan sokáig a Planning, ugyanis a csapat ismeri a Backlog-ot, azon nem szükséges finomítani, hacsak a megrendelőtől nem érkezik ilyen igény. A harmadik dolog, amit meg kell vizsgálni, hogy a csapat hogyan teljesített az előző sprintben, vagyis túlvállalta-e magát vagy sem. Ha túl sok sztorit vállaltak el, akkor le kell vonni a következtetést, és a következő sprintre kevesebbet vállalni. Ez a probléma leginkább az új, kevésbé összeszokott csapatokra jellemző, ahol még nem tudni, hogy mennyi munkát bír elvégezni a csapat. Ellenkező esetben, ha alulvállalta magát egy csapat, akkor értelemszerűen többet vállaljon, illetve, ha ideális volt az előző sprint, akkor hasonló mennyiséget javasolt.

*Backlog Grooming/Backlog Refinement:* A Product Backlog finomítása a csapattal együtt, előfordulhat például, hogy egy taszk túl nagy, így story lesz belőle, és utána taszkokra bontva lesz feldolgozva. Ha elmarad, akkor a Sprint Planning hosszúra nyúlhat, valamint abban is nagy segítség, hogy a csapat tökéletesen megértsse, hogy mit szeretne a megrendelő.

*Daily Meeting/Daily Scrum:* A sprint ideje alatt minden nap kell tartani egy rövid megbeszélést, ami maximum 15 perc, és egy előre megbeszélt időpontban, a csapattagok és a Scrum Master jelenlétében történik (mások is ott lehetnek, de nem szólhatnak bele). Érdekesség, hogy nem szabad leülni, mindenki áll, ezzel is jelezve, hogy ez egy rövid találkozó. Három kérdésre kell válaszolnia a csapat tagjainak, ezek a következőek:

- Mit csináltál a tegnapi megbeszélés óta?
- Mit fogsz csinálni a következő megbeszélésig?
- Milyen akadályokba ütközött az adott feladat megoldása során?

*Sprint Review Meeting (Futamáttekintés):* minden sprint végén összeülnek a szereplők, és megnézik, hogy melyek azok a sztorik, amelyeket sikerült elkészíteni, illetve az megfelel-e a követelményeknek. Ekkor a sztori állapotát készre állítják. Fontos, hogy egy sztori csak akkor kerülhet ebbe az állapotba, ha minden taszka elkészült, és a Review-on elfogadták. Ezen a megrendelő is jelen van.

*Sprint Retrospective (Visszatekintés):* Ez az egyik legfontosabb meeting. A Scrum egyik legfontosabb funkciója, hogy felszínre hozza azokat a problémákat, amelyek hátráltatják a fejlesztőket a feladatmegoldásban, így ha ezeket az akadályokat megoldjuk, a csapat jobban tud majd alkalmazkodni a következő sprint alatt a feladathoz. Problémák a Daily Meetingen is előkerülnek, de ott inkább a személyeket érintő kérdések vannak napirenden, míg itt a csapatmunka továbbfejlesztése az elsődleges.

### 2.3.12.3. Termékek

*Product Backlog (termékteendő lista):* Ez az a dokumentum, ahol a Product Owner elhelyezi azokat az elemeket, más néven sztorikat, amelyeket el kell készíteni. Ez egyfajta kívánságlista. A Product Owner minden sztorihoz prioritást, fontossági sorrendet rendel, így tudja szabályozni, hogy melyeket kell elsősorban elkészíteni, így a Sprint Planning során a csapattagok láthatják, hogy ami a Backlogban legfelül van, azt szeretné a vevő leghamarabb készen látni, annak van a legnagyobb üzleti értéke. Emellett a csapatok súlyozzák az elemeket, aszerint, hogy melyek az elkezdéséhez kell a kevesebb munka, így azonos prioritás mellett a kevesebb munkát igénylő elemek nagyobb a

befektetésmegtérülése (Return of Investment, ROI). Az üzleti érték meghatározása a Product Owner, a munka megbecslése a csapat feladata. A kettő hányadosa a ROI.

*Sprint Backlog (futamteendő lista):* Ebben a dokumentumban az aktuális sprintre bevállalt munkák, storyk vannak felsorolva, ezeket kell adott időn belül a csapatnak megvalósítania. A sztorik tovább vannak bontva taszkokra, és ezeket a taszkokat vállalják el a tagok a Daily Meeting során. A feladatok feldarabolása azok minél jobb megértését segíti.

*Burn down chart (Napi Eredmény Kimutatása):* Ez egy diagram, amely segít megmutatni, hogy az ideális munkatempóhoz képest hogyan halad a csapat az aktuális sprinten belül. Könnyen leolvasható róla, hogy a csapat éppen elakadt-e egy ponton, akár arra is lehet következtetni, hogy ilyen iramban kész lesz-e minden a sprint végére. Vagy éppen ellenkezőleg, sikerült felgyorsítani az iramot, és időben, vagy akár kicsit hamarabb is kész lehet a bevállalt munka.

## 2.4. Kockázatmenedzsment

A földi élet egyre több és több ponton kerül kapcsolatba az informatikával. Ez a kapcsolat a kezdeti stadionban rendszerint csak valami egészen apró változást okoz, valamelyest könnyíti az emberek életét, esetleg növeli azok szabad idejét. Majd később, ahogyan a technológia fejlődik, mind inkább fokozza a felhasználók komfortérzetét, gyorsítja a legkülönfélébb folyamatokat (a teljesség igénye nélkül: adminisztrációs, ügyviteli, számviteli, gazdasági, társadalmi stb.), és ezzel párhuzamosan növekszik a függőség, nő a rendszer komplexitása, így nő a meghibásodás kockázata is, nem is beszélve a lehetséges kár mértékéről.

A kár mértéke megdöbbentően széles skálán mozoghat. Elég csak belegondolni, vajon milyen hatással lehet egy ország gazdasági életére a tőzsdei adatok elvesztése, vagy a jegybank informatikai rendszerének összeomlása.

Hogy mennyire sebezhetők a mai informatikai rendszerek, azt az elektronikus levelezés útján terjedő vírusok elterjedése is mutatja. Az ilyen vírusok hatalmas károkat okoztak a világgazdaságnak. A "Melissa" néven ismert vírus 1999-ben még csak 80 millió dollár kárt okozott, addig az "I Love You" már 10 milliárd dollárt egy évvel később!

A kockázatmenedzsment (más néven kockázatkezelés) 4 lépése:

- kockázat azonosítása,
- kockázat értékelése,
- kockázat csökkentése,
- kockázat kommunikációja.

Egy kockázatnak minden két vetülete van:

- bekövetkezésének valószínűsége (adott időszakban statisztikai adatok alapján hányszor fordul elő),
- bekövetkezés esetén az (anyagi és/vagy emberi) kár mértéke.

A kettő szorzata adja a kockázat súlyosságát. Ezért egy valószínűtlen, de nagy kárt okozó kockázat is lehet nagyon súlyos. A kockázatmenedzsment feladata ésszerű költséghatárok között a kockázatok

valószínűségének és az általuk okozott kár csökkentése. Ezért használunk pl. vírusirtó programokat, amik kis költséggel nagymértékben csökkentik egy esetleges adatvesztés valószínűségét.

Az informatikai rendszerek széleskörű alkalmazása miatt igen nehéz azok kockázatmenedzsmentjével foglalkozni, hiszen minden eset más és más.

- Egy állami intézmény szolgáltatásorientált informatikai rendszerénél figyelembe kell venni az adott intézmény működésével kapcsolatos jogszabályokat. Tegyük fel, elő van írva az ügyintézés maximális ideje, vagyis a legfontosabb a rendszer rendelkezésre állása.
- Egy banki rendszer esetén nagyon fontos a tranzakciók biztonsága, ami azt jelenti, hogy a tranzakciókat lehallgatni, módosítani harmadik fél csak nagyon nagy költségek árán legyen képes.
- Vagy vegyünk egy kutatóintézetet, ahol a kutatók napközben elvárják, hogy késlekedés nélkül hozzáférhessenek az adataikhoz, de nem érdekli őket, ha azok esetleg éjjel elérhetetlenek a rendszer meghibásodása miatt, ha másnap reggelre az adatokat visszaállítják, akkor tulajdonképpen észre sem veszik a hibát, a bekövetkezett kár nem okoz költséget.
- És ne feledkezzünk el arról a szönyegkereskedésről sem, ahol levelezésre használják az informatikai rendszert, és a cég szempontjából a legfontosabb szempont, hogy az internetkapcsolat viszonylag stabil és a lehető legolcsóbb legyen.

A gyakorlati kockázatmenedzsment célja az, hogy a kitűzött elveknek megfelelően mutasson lehetséges védelmi intézkedéseket a felmerülő veszélyekkel szemben, és azok közül kiválassza a lehető leghatékonyabb (mind költség, mind hatás szempontjából) intézkedéscsoportot.

Az informatikai rendszerek kockázatkezelésére nem létezik konkrét metodika, azonban léteznek ajánlások, melyek segítik a menedzsert a munkájában. Magyarországon létezik egy ilyen ajánlás, az Informatikai Tárcaközi Bizottság 8. sz. ajánlása, ami általanosságban nagyon fontos veszélyekre és azok elhárításának módjaira hívja fel a vezetők figyelmét, de sajnos nem ad konkrét menedzsmenttervet.

#### 2.4.1. COBIT

Nemzetközi viszonylatban az ISACA szervezet COBIT nevű menedzsmentterve széleskörűen elfogadott. Az ISACA (*The Information Systems Audit and Control Association & Foundation*, azaz *Az Információs Rendszerek Vizsgálatával és Felügyeletével Foglalkozó Szervezet és Alapítvány*) 1969-ben jött létre, az alapítók célja az informatikai rendszerek biztonságával foglalkozó szakemberek kutatásainak összehangolása. Az általuk kialakított COBIT ajánlás (*Control Objectives for Information and related Technologies*, azaz *Felügyeleti Eljárások Információs- és kapcsolódó Technikákhoz*) egy IT menedzsereknek szóló ajánlás (angolul: best practise) gyűjtemény. Az ajánlás egyik fejezete egy akciótervet tartalmaz az informatikai rendszerek kockázatmenedzsmentjéhez, lépésről lépésre végigvezetve a menedzsert a kockázatmenedzsmenttel kapcsolatos teendőkön. A COBIT részletes bemutatása nem célunk, az ajánlást egy egyszerűsített példán keresztül mutatjuk be.

A COBIT kockázatelemző lépései:

1. Lépés: Veszélyforrások feltérképezése.
2. Lépés: Valószínűségi kategóriák meghatározása.
3. Lépés: Veszélyforrások bekövetkezésének becslése.

4. lépés: Kárkategóriák meghatározása.
5. lépés: Okozott kár becslése.
6. lépés: Kockázatkategóriák meghatározása.
7. lépés: Szorzótábla meghatározása.
8. lépés: Elviselhetetlen kockázatok meghatározása.
9. lépés: Alternatív védelmi intézkedések felderítése.
10. lépés: Javasolt védelmi intézkedések meghatározása.

A COBIT lépéseit egy példán keresztül mutatjuk be.

#### 2.4.1.1. 1. lépés: Veszélyforrások feltérképezése

Vegyük egy képzeletbeli céget, és vizsgáljuk meg, hogy annak informatikai rendszerére milyen veszélyforrások leselkednek! A veszélyforrások megtalálása nagyon nehéz feladat, mindenre gondolni kell, beleértve itt az adott ország politikai helyzetét, a rendszer földrajzi adottságait (pl. földrengés), és még hosszan sorolhatnánk. Példánkban a következő veszélyforrások fenegetnek:

- Áramszünet
- Alaplapi meghibásodás
- Adathordozó-meghibásodás
- Betöréses lopás
- Lehallgatás
- Jogosulatlan módosítás
- Más nevében adott utasítás
- Számítógépes betörés
- Vírusfertőzés
- Tűzvész

#### 2.4.1.2. 2. lépés: Valószínűségi kategóriák meghatározása

Ki kell alakítani a modellben használt bekövetkezési valószínűségi kategóriákat:

Jelölés	Megnevezés	Előfordulások száma	Leírás
PVS	nagyon kicsi (Very Small)	0.0 ... 0.1 / Év	csak eseti előfordulás
PS	kicsi (Small)	0.1 ... 0.2 / Év	ritkán előfordul
PL	nagy (Large)	0.2 ... 1.0 / Év	évente
PVL	nagyon nagy (Very Large)	1.0 ... több / Év	évente többször is

#### 2.4.1.3. 3. lépés: Veszélyforrások bekövetkezésének becslése

Meg kell becsülni a bekövetkezési valószínűségeket, ebben nagy segítséget jelenthetnek a különböző statisztikai adatok.

ID	Veszélyforrás	P
F1	Áramszünet	PVL
F2	Alaplapi meghibásodás	PL
F3	Adathordozó-meghibásodás	PL
H1	Betöréses lopás	PL
L1	Lehallgatás	PL
L2	Jogosulatlan módosítás	PS
L3	Más nevében adott utasítás	PL
L4	Számítógépes betörés	PS
L5	Vírusfertőzés	PVL

#### 2.4.1.4. 4. lépés: Kárkategóriák meghatározása

Figyelembe kell venni a vizsgált cég gazdasági volumenét, hogy a meghatározott kárkategóriák megfelelően tág teret engedjenek a veszélyforrásoknak. A kár lehet anyagi vagy emberi, esetleg minden kettő.

Jelölés		Megnevezés	Anyagi kár	Emberi kár
DVS		elhanyagolható (angolul: Very Small)	10.000 Ft	-
DS		kicsi (angolul: Small)	100.000 Ft	-
DA		közepes (angolul: Average)	1.000.000 Ft	könnyű
DL		nagy (angolul: Large)	10.000.000 Ft	súlyos
DVL		nagyon nagy (angolul: Very Large)	üzletmenet időszakos megszakadása	halálos
DD		katasztrofális (angolul: Disaster)	üzletmenet hosszabb, teljes megszakadása	tömeges

#### 2.4.1.5. 5. lépés: Okozott kár becslése

Itt is sokat segíthetnek a statisztikai adatok, de például az is, ha a kár ból kifolyólag megsemmisült adatok értékét, vagy a rendszer kiesése által okozott produktivitás csökkenésből származó bevétel kiesést figyeljük. Hárrom szempontból kell az okozott kár mértékét meghatározni:

- Confidentiality, magyarul bizalmasság,
- Integrity, magyarul sérzetlenség, és
- Availability, magyarul rendelkezésre állás.

Az előzőekben meghatározott adatok táblázatba foglalva:

ID	Veszélyforrás	P	C	I	A
F1	Áramszünet	PVL	-	DS	DA
F2	Alaplapi meghibásodás	PL	-	-	DA
F3	Adathordozó-meghibásodás	PL	DS	-	DL
H1	Betöréses lopás	PL	DL	-	DL
L1	Lehallgatás	PL	DL	-	-
L2	Jogosulatlan módosítás	PS	-	DA	-
L3	Más nevében adott utasítás	PL	-	DA	-
L4	Számítógépes betörés	PS	DL	DL	DS
L5	Vírusfertőzés	PVL	-	DS	DL

#### 2.4.1.6. 6. lépés: Kockázatkategóriák meghatározása

A kár várható értéke alapján ki kell alakítani egy harmadik kategóriát, a kockázatkategóriát.

Jelölés		Meghatározás	Kár várható értékének nagyságrendje évente
RVS	nagyon kicsi (Very Small)	10.000 Ft	
RS	kicsi (Small)	100.000 Ft	
RA	közepes (Average)	1.000.000 Ft	
RL	nagy (Large)	10.000.000 Ft	
RVL	nagyon nagy (Very Large)	beláthatatlan	

#### 2.4.1.7. 7. lépés: Szorzótábla meghatározása

A lehetséges valószínűségi és becsült kárkategóriapárokhoz meg kell határozni a kockázatkategóriát.

P/D	DVS	DS	DA	DL	DVL	DD
-----	-----	----	----	----	-----	----

PVS	RVS	RVS	RS	RA	RL	RVL
PS	RVS	RS	RA	RL	RVL	RVL
PL	RVS	RS	RA	RL	RVL	RVL
PVL	RS	RS	RL	RVL	RVL	RVL

A veszélyforrásokat tartalmazó táblázat a kockázatkategóriával kiegészítve a károkozási adatok közül (C, I, A oszlop) mindenkor legnagyobbát kell figyelembe venni, azaz a legrosszabb esetet.

ID	Veszélyforrás	P	C	I	A	R
F1	Áramszünet	PVL	-	DS	DA	RL
F2	Alaplapi meghibásodás	PL	-	-	DA	RA
F3	Adathordozó-meghibásodás	PL	DS	-	DL	RL
H1	Betöréses lopás	PL	DL	-	DL	RL
L1	Lehallgatás	PL	DL	-	-	RL
L2	Jogosulatlan módosítás	PS	-	DA	-	RA
L3	Más nevében adott utasítás	PL	-	DA	-	RA
L4	Számítógépes betörés	PS	DL	DL	DS	RL
L5	Vírusfertőzés	PVL	-	DS	DL	RVL

#### 2.4.1.8. 8. lépés: Elviselhetetlen kockázatok meghatározása

Meg kell határozni a szorzótáblán azokat a mezőket, melyek elviselhetetlen mértékű kockázatot reprezentálnak. A módosított szorzótáblán a szürke hátterű mezők jelentik az elviselhetetlen kockázatokat.

P/D	DVS	DS	DA	DL	DVL	DD
PVS	RVS	RVS	RS	RA	RL	RVL
PS	RVS	RS	RA	RL	RVL	RVL
PL	RVS	RS	RA	RL	RVL	RVL
PVL	RS	RS	RL	RVL	RVL	RVL

Ezek alapján a következő kockázatok elviselhetetlenek:

ID	Veszélyforrás	P	C	I	A	R
F3	Adathordozó-meghibásodás	PL	DS	-	DL	RL
H1	Betöréses lopás	PL	DL	-	DL	RL
L1	Lehallgatás	PL	DL	-	-	RL
L5	Vírusfertőzés	PVL	-	DS	DL	RVL

#### 2.4.1.9. 9. lépés: Alternatív védelmi intézkedések felderítése

Számba kell venni a lehetséges védelmi megoldásokat, meg kell határozni, hogy azok az egyes veszélyforrásokra hogyan hatnak és milyen költségtényezővel rendelkeznek. A használt jelölések:

Az adott védelmi intézkedés alkalmazása:

- D: az okozott kár kategóriáját csökkenti eggyel.
- DD: az okozott kár kategóriáját csökkenti kettővel.
- P: a bekövetkezési valószínűségi kategóriát csökkenti eggyel.
- PP: a bekövetkezési valószínűségi kategóriát csökkenti kettővel.
- E: a veszélyforrás kockázatát megszünteti.

ID	Védelmi intézkedés	Beruházás	Éves költség	Hatás
V1	Szünetmentes táp	5.000.000	50.000	F1-D, F1-P
V2	Áramfejlesztő	10.000.000	200.000	F1-E
V3	Poroltó	1.000.000	200.000	F2-D
V4	Duplikálás	5.000.000	200.000	F2-PP, F3-PP
V5	Hibatűrő rendszer	30.000.000	2.000.000	F2-PP, F3-PP
V6	Biztonsági mentések	2.000.000	2.000.000	F3-D, L5-D
V7	Riasztórendszer	20.000.000	2.000.000	H1-D
V8	Élőerős őrzés-védelem	2.000.000	30.000.000	H1-D, H1-PP
V9	Biztosítás	0	10.000.00	H1-D
V10	Adatkommunikáció rejtjelezése	2.000.000	0	L1-E
V11	Hozzáférés-védelem	500.000	0	L2-P, L4-P

V12	Digitális aláírás	500.000	0	L3-E
V13	Aktív vírusvédelem	500.000	200.000	L5-P
V14	Floppyk kiszerelése	0	0	L5-P

#### 2.4.1.10. 10. lépés: Javasolt védelmi intézkedések meghatározása

Ebben a lépésben kell meghatározni a lehető legideálisabb védelmi intézkedéscsoportot. A cél a lehető legkisebb beruházási és éves költséggel megszüntetni az összes elviselhetetlen kockázatot. Elviselhetetlen kockázatot jelent az F3 (Adathordozó-meghibásodás), H1 (Betöréses lopás), L1 (Lehallgatás), és L5 (Vírusfertőzés). A fentiek közül elegendő, a szorzótábla figyelembevételével, a valószínűség vagy a kár egy szinttel történő csökkentése. Kivéve az L5 (Vírusfertőzés) kockázatot, ahol vagy legalább két szinttel kell csökkenteni a valószínűséget, vagy legalább eggyel a kárt és a valószínűséget.

A szóba jövő intézkedések:

- F3 esetén: V4, V5, V6
- H1 esetén: V7, V8, V9
- L1 esetén: V10
- L5 esetén: V6, V13, 14

A szóba jövő intézkedések adatai:

ID	Védelmi intézkedés	Beruházás	Éves költség	Hatás
V4	Duplikálás	5.000.000	200.000	F2-PP, F3-PP
V5	Hibatűrő rendszer	30.000.000	2.000.000	F2-PP, F3-PP
V6	Biztonsági mentések	2.000.000	2.000.000	F3-D, L5-D
V7	Riasztórendszer	20.000.000	2.000.000	H1-D
V8	Élőerős őrzés-védelem	2.000.000	30.000.000	H1-D, H1-PP
V9	Biztosítás	0	10.000.00	H1-D
V10	Adatkommunikáció rejtjelezése	2.000.000	0	L1-E
V13	Aktív vírusvédelem	500.000	200.000	L5-P
V14	Floppyk kiszerelése	0	0	L5-P

A legkisebb költségű intézkedések kiválasztásánál az éves költség általában nagyobb súlyal esik latba, mint a kezdeti beruházás költsége. Jelen esetben 3x (háromszoros) szorzóval számoltunk. Így az elviselhetetlen kockázatok legkisebb költségű megszüntetése a következő tevékenységekkel érhető el:

ID	Védelmi intézkedés	Beruházás	Éves költség	Hatás
V6	Biztonsági mentések	2.000.000	2.000.000	F3-D, L5-D
V7	Riasztórendszer	20.000.000	2.000.000	H1-D
V10	Adatkommunikáció rejtjelezése	2.000.000	0	L1-E
V14	Floppyk kiszerelése	0	0	L5-P

Érdekes, hogy a fenti intézkedések között nem maradt meg a vírusvédelem, ami egy általánosan elfogadott védekezési módszer. Ez arra utal, hogy valamely intézkedés hatását (pl. floppyk kiszerelése) túlbecsültük, vagy a vírusfertőzés kockázatának valószínűségét vagy okozott kárát alulbecsültük.

#### 2.4.2. Informatikai biztonsági módszertani kézikönyv ITB ajánlás

Az alábbiakban közöljük az Informatikai Tárcaközi Bizottság (ITB) 8. sz. „Informatikai biztonsági módszertani kézikönyv” című ajánlás általunk legfontosabbnak ítélt részeit. Az ajánlás a <http://www.itb.hu/ajanlasok/a8/> oldalon érhető el. Maga az ajánlás 1994-ben keletkezett, de mai napig érvényes útmutatásokat tartalmaz.

Az ajánlás nagy értéke, hogy felsorol nagyon sok kockázatot, amelyeket fenyegetettségnek nevez. A kockázat jelentése ebben a dokumentumban a fenyegetés valószínűségének és az általa okozott kárnak a szorzata. Felsorol továbbá biztonsági intézkedéseket is. Megadja, hogy mely intézkedés mely fenyegetésre van hatással. Az alábbi szemelvénnyben csak az adathordozókra vonatkozó fenyegetéseket és biztonsági intézkedéseket idézzük az ajánlásból.

Az ajánlás segít meghatározni az elviselhetetlen kockázatokat. Ez egy kereszttáblával történik, amelyben szerepelnek a gyakoriságok és a károk. Mindkét értéket nullától négyig (0-4) pontozhatjuk, ahol a nagyobb szám a nagyobb gyakoriságra, illetve a nagyobb kockázatra utal. Az elviselhetetlen kockázat megadható például úgy, hogy a két szám összege nagyobb vagy egyenlő, mint 5, vagy a szorzatuk nagyobb, mint 4. A feltétel megadásánál felhasználjuk korábbi tapasztalatainkat.

##### 2.4.2.1. Informatikai Biztonsági Koncepció (IBK)

Az ajánlás az Informatikai Biztonsági Koncepció (IBK) című dokumentum elkészítésére ad ajánlásokat.

Az IBK tartalmazza:

- az adott szervezet informatikai biztonságának követelményeit,
- az informatikai biztonság megteremtése érdekében szükséges intézkedéseket,
- ezek kölcsönhatásait és következményeit.

Az IBK főbb tartalmi összetevői:

- a védelmi igény leírása (meglévő állapot, fenyelgetettségek, fennálló kockázatok),
- az intézkedések fő irányai (kockázat-menedzselés),
- a feladatok és felelősségek megosztása (az intézkedések megvalósítása során),
- időterv (megvalósítási ütemekre és az IBK felülvizsgálatára).

#### 2.4.2.2. Az IBK kialakításának alapjai

Valamely informatikai rendszer biztonságának vizsgálata során elsőként a meglévő, potenciálisan fenyelgetett értékeket kell feltérképezni és újraértékelni. Ehhez meg kell határozni a felhasználó biztonsági követelményeit, amelyek teljesülése ahhoz szükséges, hogy lehetővé váljon az elhatározottaknak megfelelő rendeltetésszerű feldolgozás. Azután a következményeket kell feltárni, amelyek kialakulhatnak, ha ezek a követelmények (védelmi célok) az alapfenyelgetettségeket illetően nem teljesülnek.

Emlékeztetésül felidézzük, hogy „alapfenyelgetettség”-nek azon fenyelgető tényezők hatásösszegét nevezzük, amelyek az információk

- rendelkezésre állását,
- sértetlenségét,
- bizalmasságát,
- hitelességét,
- illetve az informatikai rendszer működőképességét veszélyeztetik.

Mindazonáltal az értékek nem korlátozódnak az adott hardverekre és szoftverekre, amelyek pénzbeli értéke (ár) ismert. Nagyobb jelentőségűek azok az értékek, amelyeket az informatikai rendszer alkalmazása és a feldolgozandó információk képviselnek. Az informatika-alkalmazás és az információk értéke azzal határozható meg, ha elköpzeljük, milyen utólagos következményekkel jár bizonyos események bekövetkezése - amelyeket egyébként fenyelgető tényezőknek nevezünk. Ezeket a következményeket általában értékvesztésnek vagy kárnak tekintjük, s arra kell törekednünk, hogy ne következzenek be. Az értékek - és ezzel párhuzamosan a károk - a következő területeken jelentkezhetnek:

- személyi biztonság,
- anyagi javak, vagyontárgyak,
- politika és társadalom,
- törvények és előírások,
- gazdaság.

A fenyelgető tényezők az informatikai rendszerelemekhez kapcsolódnak, és azokon keresztül okozhatnak károkat, miután az informatika-alkalmazás függ a rendszerelemektől. Éppen ezért kell megvédeni a rendszerelemeket a fenyelgető tényezők ellen. Valamennyi olyan rendszerelement védeni kell, amelyektől az informatikai rendszer működése és valamilyen módon az alkalmazásai függnek, és amelyeket valamely fenyelgető tényező negatív módon érinthat. Ehhez a következő meglévő rendszerelem-csoportokat kell áttekinteni:

- Tárgyiasult elemcsoportok
  - környezeti infrastruktúra

- hardver
- adathordozók
- dokumentumok, iratok
- Logikai elemcsoportok
  - szoftver
  - adatok
  - kommunikáció
- Személyi elemcsoport
  - személyzet, felhasználók, ellenőrök

Általában védeni szükséges azokat a rendszerelemeket is, amelyek maguk is a védelmi intézkedések körébe tartoznak vagy azok részét képezik, mivel ellenkező esetben az intézkedést akár hatályon kívül is helyezhetnénk. Ide tartoznak például a szervezési szabályozások, amelyek csak akkor hatnak, ha végrehajtják azokat, vagy a biztonsági szoftver, amelyet nem ésszerűen helyeztek el, ha manipulálható vagy megkerülhető.

A rendszerelemekhez rendelve egyedileg meg kell határozni a fenyelgető tényezőket, amelyek a vizsgált környezetben egyáltalán felléphetnek. Miután nem védekezhetünk valamennyi fenyelgető tényező ellen tökéletesen, meg kell ismerni a legfontosabbakat. Ehhez valamennyi feltárt fenyelgető tényezőt értékelni kell. Az értékelés függ a kár bekövetkezésének várható valószínűségétől és a bekövetkezett kár nagyságától, amennyiben a fenyelgető tényező kifejtí hatását. Ebből a két részből tevődik össze a kockázat.

A bekövetkezés valószínűsége olyan eseményknél, amelyeket emberek célzottan okoznak, a potenciális tettek felkutatásával és azok számának megadásával becsülhető meg, akik a megfelelő lehetőségekkel és ismeretekkel rendelkeznek. Az olyan események gyakoriságát, melyek műszaki hibák vagy vis maior esetek által lépnek fel, statisztikák és saját tapasztalatok összegzésével lehet megbecsülni. Ugyanez érvényes a személyek akaratlan hibás tevékenysége miatt bekövetkező károk gyakoriságának becslésére. A statisztikáknál mindenkorral figyelembe kell venni, hogy mely körülmények között készültek, miután nem lehet szolgai módon átvenni, illetve minden további nélküli alkalmazni azokat egy adott felhasználó speciális körülményeire. Ezen túlmenően figyelembe kell venni, hogy a statisztikai adatok mindig tartalmaznak bizonytalanságokat.

A kárnegyság előzetes értékelésekor mérlegelni kell, hogy az adott fenyelgető tényező hatására milyen anyagi és más természetű károk következnek be, melyek a közvetlen károk és minden későbbi következményekkel, úgynevezett következményes károkkal kell számolni.

A kockázatelemzésből biztonsági igény adódik, amennyiben minden kockázatot megvizsgálunk és megállapítjuk, hogy egy vagy több kockázat nem elviselhető. A biztonsági követelmények egyenként abból adódnak, hogy kiválasztjuk a túl magas kockázatokat. Ezen biztonsági követelményekből kiindulva kell elkészíteni az informatikai biztonsági konцепciót, ennek keretében kiválasztani a megfelelő intézkedéseket, amelyek ezeket a kockázatokat elfogadható szintre csökkentik, és a költségek, illetve a haszon szempontjából is igazolhatók.

#### 2.4.2.3. [Az informatikai rendszer elemeinek fenyelgető tényezői](#)

##### **Fenyelgető tényezők az adathordozók területén**

A rendelkezésre állást és működőképességet fenyegető tényezők:

- Lopás
- Szándékos megkárosítás (mechanikus, mágneses stb.)
- Károsodás külső események miatt, például tűz, víz
- Károsodás helytelen kezelés vagy tárolás miatt
- Elöregedés miatti használhatatlanság (demagnetizálódás, mechanikai változások)
- Károsodás környezeti körülmények miatt (hőmérséklet, nedvesség stb.)
- Már nem fellelhető adathordozók (nem szabályszerű tárolás)
- Hibásan legyártott adathordozók (fizikai íráshiba)
- Használhatatlanság a hiányzó kódoló, illetve dekódoló berendezések miatt
- Használhatatlanság az inkompatibilis formátum miatt (logikai és fizikai értelemben)

A sértetlenséget fenyegető tényezők:

- Hiányzó vagy nem kielégítő jelölés
- A jelölés meghamisítása
- Ismeretlen vagy kétséges eredetű adathordozók használata (szoftverimport)

A bizalmasságot fenyegető tényezők:

- Az adathordozók újrafelhasználásra vagy megsemmisítésre történő kiadása előzetes törlésük vagy átírásuk nélkül
- Lopás

Fenyegető tényezők az adathordozók kezelésével összefüggésben:

- Ellenőrizetlen hozzájutás az adathordozókhöz
- A szervezet tulajdonát képező adathordozók privát célú használata
- Privát adathordozók szolgálati használata (illetékes másolatok, vírusok behatolása)
- Ellenőrizetlen másolás

#### 2.4.2.4. [Biztonsági intézkedések](#)

##### **Biztonsági intézkedések az adathordozók védelmében**

- Adathordozó-adminisztráció kialakítása (beszerzés, gazdálkodás, készlet- és használat nyilvántartás, selejtezési eljárás, az utánpótlás megszervezése stb.)
- Külön, belépés-ellenőrzéssel ellátott adathordozó tároló helyiségek kialakítása.
- A környezeti körülmények ellenőrzése (hőmérséklet, nedvességtartalom stb.).
- Megelőző intézkedés az előregedés és a már nem preferált formátumok vonatkozásában (átmásolás).
- Törlés a felszabadítás, kiselejtezés előtt.
- Katasztrófa-megelőzés céljából a másodpéldányok kiemelten biztonságos (más telephelyen történő) raktározása.
- A beszerzett adathordozók ellenőrzése az alkalmazásra való felszabadításuk előtt.

- Előírások az adathordozók felhasználói számára (védelem rongálódás ellen, külső jelölés, védelem jogosulatlan használattól stb.).
- Az előállított adathordozók ellenőrzése (újraolvashatóság).
- Az adathordozók tartalmának védelme (kódolás, rejtjelezés, olyan jelölés, amely nem tartalmaz közvetlen utalást a tartalomra, kódoló, dekódoló eszközök használata stb.).
- Privát adathordozók szolgálati célokra vagy fordítva történő igénybevételének tilalma.
- A kölcsönzés, a regisztrálás, a visszaadás eljárása.
- Az adathordozók ellenőrzött kiselejtezése.

#### 2.4.2.5. Kárkövetkezmények és azok értékelési rendszere

Az értékskála kialakítása: a károknak a jelentéktelentől a katasztrófáig terjedő nagyságrendjét általában öt értékkategóriában, a károk típusától függő értékekkel célszerű hozzárendelni a skálabeosztáshoz, amint azt a következő példák szemléletesen mutatják.

Ft-összegekben meghatározott érték/kár (pl. üzleti veszteség):

- "-": nem jár pénzügyi veszteséggel,
- "0": 10 ezer Ft-ig,
- "1": 10 ezer Ft felett,
- "2": 100 ezer Ft felett,
- "3": 1 millió Ft felett,
- "4": 10 millió Ft felett.

Testi épség/személyi biztonság:

- "1": Egy személy könnyebb sérülése.
- "2": Egy (nem több) személy komolyabb sérülése.
- "3": Több ember sérülése.
- "4": Egy személy súlyos sérülése vagy halála, több ember súlyos sérülése.
- "4+": Több ember halála.

Személyiségi jogok megsértése:

- "1": Kisebb kényelmetlenség egy személy számára (pl. a személy által magánjellegűnek tekintett adat felfedése, további következmények nélkül).
- "2": Komoly kényelmetlenség egy személy számára (pl. pénzügyi információ kiadása).
- "3": Tartós kényelmetlenség egy személy számára (pl. egészségügyi adatok kiadása).
- "4": Polgári peres eljáráshoz vezető jogosítás.

A gyakoriságskála kialakítása: a gyakoriság mértéke alkalmazástól, rendszertől, védelmi igénytől függően más és más lehet. A következő példa illusztrálja, hogyan lehet az értékek jelentését rögzíteni a várható bekövetkezési esetek száma szerint.

- "4": Percenként egy.
- "3": Óránként egy.
- "2": Naponta egy.

- "1": Havonta egy.
- "0": Évente egy.
- "0-": A következő években nem várható.

A kockázati mátrix kialakítása: A kockázatokat egy kijelölt értékhatárhoz képest „elviselhető” (E) vagy „nem elviselhető” (N) kategóriába soroljuk. Az értékhatár természetesen káronként különböző lehet.

Példa: Az elviselhető kockázatok felső határaként azt az értékpárt jelöltük ki, amelyek esetében a két érték összege az 5-öt nem éri el.

K	4+	N	N	N	N	N	N
Á	4	E	E	N	N	N	N
R	3	E	E	E	N	N	N
É	2	E	E	E	E	N	N
R	1	E	E	E	E	E	N
T	0	E	E	E	E	E	E
É	-	E	E	E	E	E	E
K	0-	0	1	2	3	4	
	GYA	KO	RI	SÁ	GI	ÉR	TÉK

#### 2.4.3. Common Criteria

A Közös Követelményrendszer az IT Biztonság Értékeléséhez (angolul: Common Criteria for Information Technology Security Evaluation, vagy röviden: Common Criteria) egy széles körűen elfogadott szabvány. A Common Criteria előírja, hogy hogyan kell tesztelni az informatikai rendszereket, hogy azok biztonságáról meggyőződjünk.

A Common Criteria alkalmazása előtt definiálni kell, hogy milyen szoftvert vagy informatikai rendszert kívánunk tesztelni. Ezt nevezük az értékelés tárgyának. Meghatározzuk azokat a felhasználói követelményeket, más néven biztonsági funkciókat, amiket az értékelés tárgyának tudnia kell. Ezeket védelmi profilokba gyűjtjük. Ezek alapján készül a biztonsági rendszerterv. A Common Criteria azt írja le, hogy hogyan kell a biztonsági rendszertervben leírt biztonsági funkciókat tesztelni. Ehhez 7 úgynyevezett értékelési garancia szintet határoz meg. minden szint leírja, hogy milyen garanciális biztonsági követelményeket kell teljesíteni az értékelés tárgyának, hogy az adott szintnek megfeleljen. Minél magasabb értékelési garancia szintet ér el egy szoftver, annál biztonságosabbnak gondoljuk, habár a magasabb szint csak azt jelenti, hogy többféle szempontból és nagyobb szigorral tesztelték a biztonsági rendszertervben leírt funkciókat.

Nézzük az egyes fogalmakat részletesebben is. A fogalmak lefordításánál figyelembe vettük a <http://www.itb.hu/ajanlasok/a16/> oldalon található fordítást.

ÉT – Értékelés Tárgya (TOE – Target of Evaluation): Az a szoftver vagy informatikai rendszer, amely biztonságát vizsgáljuk a Common Criteria segítségével. Fontos megjegyezni, hogy a Common Criteria nem foglalkozik az ÉT környezetével, pl. az épületbiztonsággal.

**VP – Védelmi Profil (PP – Protection Profile):** A VP egy dokumentum, amelyet általában felhasználói csoportok alakítanak ki a saját elvárásainak megfelelően egy informatikai rendszerrel kapcsolatban. Létezik VP például a tűzfal vagy az intelligens kártya termékekhez. A VP biztonsági követelmények gyűjteménye. Egy szoftvercég dönthet úgy, hogy a termékét egy VP-nek megfelelően fejleszti ki és az ott meghatározott követelményeket implementálja. A VP-k alapján írhatják a szoftvercég szakemberei a biztonsági rendszertervet (lásd lent), de ez nem kötelező, figyelmen kívül is hagyhatják azokat.

**BRT – Biztonsági Rendszerterv (ST – Security Target):** A BRT egy vagy több, esetleg nulla VP alapján jön létre. Az ott megfogalmazott biztonsági követelmények alapján sorolja fel azokat a biztonsági funkcionális követelményeket (lásd lent), amiket tesztelni fogunk. Fontos kiemelni, hogy az ÉT-t csak ezek ellen teszteljük. A Common Criteria rugalmasságát az adja, hogy nem ad semmilyen előírást a BRT tartalmára, így két teljesen különböző IT terméknek (pl. egy tűzfal és egy online bolt) két teljesen eltérő BRT-je lehet. Sőt két ugyanolyan típusú IT terméknek is lehet két teljesen különböző BRT-je. A BRT általában nyilvános, így a vevők tudják, milyen biztonsági funkciói vannak az ÉT-nek.

**BFK-ek – Biztonsági Funkcionális Követelmények (SFRs – Security Functional Requirements):** A BFK-ek általánosan megfogalmazott biztonságra vonatkozó funkcionális követelmények. Például lehessen a felhasználót beléptetni. A Common Criteria sok BFK-t sorol fel. Ezek közül semmit sem tesz kötelezővé, de a köztük fennálló összefüggésekre rávilágít. Például a felhasználó beléptetéséhez jelszót kell kérni, a jelszavakat kódolva kell eltárolni.

**GBK-ek – Garanciális Biztonsági Követelmények (SARs – Security Assurance Requirements):** A GBK-ek olyan követelmények, amelyeket a fejlesztés során vagy a tesztelés során kell kielégíteni annak érdekében, hogy a vállalt biztonsági funkciókat tudja teljesíteni az ÉT. Ezeknek a követelményeknek számszerűen mérhetőnek kell lenniük. GBK lehet például, hogy a forráskódot verziókövető rendszerben kell tárolni. A szabvány felsorol sok GBK-t, sőt azokat csoportokba szervezi.

**ÉGSz – Értékelési Garancia Szint (angolul: Evaluation Assurance Level, röviden: EAL):** A Common Criteria alkalmazásának eredményeként az ÉT egy ÉGSz-t kap 1-től 7-ig. minden ÉGSz egy GBK csomag, amely az ÉT teljes fejlesztését lefedi különböző szigorúsággal. A legmegengedőbb az 1-es szint, a legszigorúbb a 7-es. minden szint eléréséhez teljesíteni kell az előzőeket is. Ugyanakkor a magasabb ÉGSz szint nem feltétlenül jelenti, hogy egy termék biztonságosabb. Csak annyit, hogy a BRT-ben leírt biztonsági funkciók szigorúbben lettek tesztelve. Ha az ÉGSz mellett látható egy pluszjel is, akkor az azt jelenti, hogy teljesítette az adott ÉGSz elvárásait és a magasabb szintekről is néhány GBK-t. Így van például ÉGSz4+ szint is, illet kapott a Windows XP operációs rendszer, ami azt jelenti, hogy teljesítette az ÉGSz4 elvárásait és a magasabb szintekről is néhány GBK-t.

Minél magasabb ÉGSz-t szeretne elérni egy szoftverfejlesztő cég, annál több és részletesebb dokumentációt kell elkészítenie a szigorú tesztek alapján, ami egyre drágább. Ugyanakkor a cégeket motiválja, hogy a magas ÉGSz magas vásárlói bizalmat is jelent. Általában állami / katonasági megrendelések feltétele a Common Criteria alkalmazása. Általában az ÉGSz4 szint felett már nem éri meg a befektetés, kivéve néhány nagyon érzékeny alkalmazási területet, pl. atomerőművek, ahol elvárt a magasabb ÉGSz szint.

Az ÉGSz szintek a következők:

- ÉGSz1: Funktionálisan tesztelt (EAL1: Functionally Tested)
- ÉGSz2: Strukturálisan tesztelt (EAL2: Structurally Tested)
- ÉGSz3: Módszeresen tesztelt és ellenőrzött (EAL3: Methodically Tested and Checked)
- ÉGSz4: Módszeresen tervezett, tesztelt és áttekintett (EAL4: Methodically Designed, Tested, and Reviewed)
- ÉGSz5: Félförmálisan tervezett és tesztelt (EAL5: Semiformally Designed and Tested)
- ÉGSz6: Félförmálisan igazolt terv és tesztelt (EAL6: Semiformally Verified Design and Tested)
- ÉGSz7: Formálisan igazolt terv és tesztelt (EAL7: Formally Verified Design and Tested)

### **3. Programozási technológiák – Tervezési alapelvek**

Ez a rész a „Programozás technológiák” című tárgy tudásanyagát öleli fel. Két nagy programozási paradigmát mutatunk be, az objektumorientált programozást (OOP) és az aspektusorientált programozást (AOP). Az elsőt azért, mert manapság ez a legnépszerűbb, leginkább kiforrott, legjobban támogatott, de ami ezeknél is fontosabb, támogatja a rugalmas forráskód fejlesztését. Erre azért van szükség, mert a programozástechnológiák alapelve kimondja, hogy a program kódja állandóan változik. Az aspektusorientált programozással (AOP) azért foglalkozunk, mert van egy terület, amit OOP segítségével csak csúnyán lehet megoldani. Ha minden objektumnak kell naplóznia, jogosultságot ellenőriznie, akkor hova tegyük ezeket a metódusokat? Az AOP erre ad megoldást.

Az OOP bemutatására ebben a fejezetben kerül sor, az AOP bemutatására a Programozási technológiák – Technológiák című fejezetben.

A fejezet két legfontosabb része az objektumorientált tervezés és a tervezési minták. Mindkét technológia azt mutatja meg, hogyan kell könnyen bővíthető, a változásokhoz könnyen alkalmazkodó, újrahasznosítható, egyszóval rugalmas kódot fejleszteni.

Az ebben a jegyzetben tárgyalt programozási technológiák akkor hasznosak, ha rugalmas szoftvert szeretnénk fejleszteni, amit könnyű módosítani és bővíteni. Erre azért van szükség, mert a programozástechnológia alaplevéből tudjuk, hogy a program kódja állandóan változik. Azaz érdemes felkészülni előre az elkerülhetetlen változásokra. Persze mondhatjuk azt is, hogy ez ránk nem érvényes, mert kicsiben programozunk (programming in small). Ugyanakkor, ha egynél több programozó dolgozik a program fejlesztésén, ami hosszabb, mint néhány ezer sor, azaz nagyban programozunk (programming in large), akkor a változásokat aligha tudjuk elkerülni.

#### **3.1. Objektumorientált programozás – OOP**

##### **3.1.1. Bevezetés**

A szoftverkritizise a programozási nyelvek azt a választ adták, hogy megjelentek a modulok, illetve a moduláris programozás. A modul a forráskód olyan kis része, amelyet egy programozó képes átlátni. A modulok gyakran fordítási alegységek is, azaz külön állományban találhatók. Az objektumorientált programozás (OOP) esetén a modul az osztály, ami egyben fordítási alegység is.

Az osztály első megközelítésben a valóság egy (megfogható vagy megfoghatatlan) darabkájának absztraktiója. Hogy ez a darabka kicsi vagy nagy, az az osztály felbontását (más szavakkal: granularitását, szemcsézettségét, durvaságát) adjá meg. Ugyanakkor az osztály lehet teljesen technikai is, a valóságban semmihez sem kapcsolható. Tervezési minták sok ilyen osztályt tartalmaznak.

Az osztály második megközelítésben egy összetett, inhomogén adattípus. Sokban hasonlít a rekordhoz, ami szintén egy összetett, inhomogén adattípus. Ugyanúgy vannak mezői, a mezői bármilyen típusúak lehetnek, a mezőit minősítő jellel (sok nyelvben ez a pont (.)) érjük el. Egy különbség, hogy az osztály tartalmazhat metódusokat (eljárásokat, függvényeket), a rekord nem.

A rekord az eljárásorientált nyelvek (vagy más néven az imperatív nyelvek) kedvenc típusa. Az eljárások rekordokon dolgoznak. Az OOP is ebbe a családba tartozik, csak itt már a rekordokat és a rekordokon

dolgozó eljárásokat egybeolvastjuk, méghozzá osztályokba. Azt mondjuk, hogy az adatokat és a rajtuk végrehajtott műveleteket egységekbe zárjuk. Ezeket az egységeket hívjuk osztályoknak.

Az osztály mezőkből (más néven adattagokból) és metódusokból áll. A metódusok az adattagokon értelmezett műveletek.

```
public class Kutya {  
    private String név;  
    public Kutya(String név) { this.név = név; }  
    public String GetNév() { return név; }  
}
```

#### Osztály példa – Kutya osztály

Az osztálynak lehetnek példányai. A példányokat objektumoknak hívjuk. Ha maradunk annál a megközelítésnél, hogy az osztály a valóság absztrakciója, akkor a Kutya osztály a világ összes lehetséges kutyájának az absztrakciója. Ennek az osztálynak egy példánya, azaz egy Kutya típusú objektum, pedig a valóság egy konkrét kutyájának az absztrakciója. A konkrét kutya nevét az osztály konstruktőrben adhatjuk meg, amikor példányosítjuk.

Az eddig leírtak valószínűleg mindenki ismerte voltak. Ugyanakkor van egy másik, programozástechnikai megközelítés is. E szerint az osztálynak két jellemzője van:

- felülete és
- viselkedése (vagy implementációja).

Az objektumnak három jellemzője van:

- felülete (vagy típusa),
- viselkedése és
- belső állapota.

Az osztály felületét a publikus részei adják. Mezőt ritkán teszünk publikussá, jellemzően metódusokon és property-ken keresztül használjuk őket (kivéve a statikus konstans mezőket), ezért az osztály felületét a publikus metódusainak feje adja. Az osztály felülete adja meg, hogy milyen szolgáltatásokat nyújt az osztály. A Kutya osztály például vissza tudja adni a kutya nevét a GetNév metódussal.

Az osztály viselkedését a metódusainak (nem csak a publikus, hanem az összes metódusának) implementációja határozza meg. Például a GetNév metódus viselkedése az, hogy visszaadja a név mező értékét. Habár ez a szokásos viselkedése a GetNév-nek, más viselkedést is megadhatnánk.

```
Kutya kutyám = new Kutya("Bodri");
```

#### Példányosítás példa – a kutyám nevű objektum a Kutya osztály példánya

A fenti példában létrehoztuk a kutyám nevű objektumot, ami a Kutya osztály egy példánya. A konkrét kutyánkat Bodrinak hívják. Fontos megjegyezni, hogy a kutyám nevű változó Kutya osztály referencia típusú. Tehát a kutyám csak egy referencia a példányra, amit a new utasítással hoztunk létre. A példány típusa és a referencia típusa nem feltétlenül egyezik meg, mint azt látni fogjuk.

Az objektum felülete megegyezik az osztályának a felületével, azaz a kutya objektum és a Kutya osztály felülete megegyezik. Még pontosabb azt mondani, hogy kutyám objektum Kutya típusú, vagy rövidebben, a kutyám Kutya típusú. Látni fogjuk, hogy egy objektumnak több típusa is lehet.

**Érdekesség:** Az erősen típusos nyelveken egy objektumot csak akkor használhatok egy osztály példányaként, ha olyan típusú. Ilyen nyelv pl. a Java és a C#. A gyengén típusos nyelveknél elegendő, ha az objektum felülete bővebb az osztálynál. Ilyen nyelv pl. a Smalltalk.

Az objektum viselkedését a metódusainak implementációja adja. Ez megegyezik annak az osztálynak a viselkedésével, aminek a példány az objektuma. Fontos megjegyezni, hogy az objektum viselkedése a program futása közben változhat, mint azt látni fogjuk.

Az objektum belső állapotát mezőinek pillanatnyi értéke határozza meg. Mivel az osztály metódusai megváltoztathatók a mezők értékeit, ezért a metódusokat tekinthetjük állapotátmeneti operátoroknak is. Az objektum kezdő állapotát a mezőinek kezdő értéke és az őt létrehozó konstruktorhívás határozza meg.

Fontos megjegyezni, hogy az interfésznek csak felülete van, az absztrakt osztálynak felülete és részleges viselkedése. Az absztrakt osztálynak lehet, hogy egyáltalán nincs viselkedése, ha minden metódusa absztrakt.

A fenti fogalmakkal fogalmazzuk meg az objektum orientáltság jól ismert alapelveit. Látni fogjuk, hogy az eddigi kedvencünkönkről, az öröklődésről kiderül, hogy veszélyes. Az új kedvencünk a többalakúság lesz.

### 3.1.2. Egységbázárás – Encapsulation

Az egységbázárás (angolul: encapsulation) klasszikus megfogalmazása valahogy így hangzik: Az adattagokat és a rajtuk műveleteket végrehajtó metódusokat egységekbe zárjuk, ezt az egységet osztálynak nevezzük. Új fogalmainkkal az egységbázárás azt jelenti, hogy az objektum belső állapotát meg kell védeni, azt csak a saját metódusai változtathatók meg. Ezt szokás információ rejtésnek (angolul: information hiding) is nevezni. Ez a két megfogalmazás kiegészíti egymást, mindenkorban.

### 3.1.3. Öröklődés – Inheritance

Az öröklődés (angolul: inheritance) a kód-újrahasznosítás kényelmes formája. A gyermek osztály az ős osztály minden nem privát mezőjét és metódusát megörökli. Azaz a gyermek osztály öröklíti az ős osztály felületét és megvalósítását. Mint látni fogjuk, az öröklődés a gyermek és az ős osztály között implementációs függőséget okoz, ami kerülendő. Öröklődés helyett, ha csak lehet, objektum-összetételek ajánlott használni.

Az örökölt, absztrakt vagy virtuális metódusokat felülírhatjuk (overriding). Ezt a lehetőséget sokan a többalakúsághoz sorolják.

### 3.1.4. Többalakúság – Polymorphism

A jegyzetben ismertetett tervezési alapelvek és tervezési minták majd mindegyike a többalakúságon (angolul: polymorphism) alapszik. Tehát ez egy nagyon fontos alapelvek. Maga a többalakúság az öröklődés következménye. Mivel a gyermek osztály öröklíti az ős felületét, ezért a gyermek osztály példányai megkapják az ős típusát is. Így egy objektum több típusként, azaz több alakban is használható.

```
public class Vizsla : Kutya { }
Kutya kutya = new Vizsla("Frakk");
```

Többalakúságra példa – a „Frakk” nevű vizsla példány használható Kutyaként

A fenti példában a Vizsla osztály a Kutya osztály gyermeke. A Vizsla konstruktora segítségével készítünk egy „Frakk” nevű Vizsla példányt. Ennek a példánynak három típusa van: Vizsla, Kutya és Object. Mindhárom típus példányaként használható. Erre rögtön látunk is egy példát, hiszen egy Kutya típusú változónak adjuk át értékül az új példányt.

Egy osztály példányai az öröklődési hierarchián felfelé haladva rendelkeznek az összes típussal. Ennek megfelelően minden objektum Object típusú is, hiszen ha nem adom meg egy osztály ősét, akkor az az Object osztályból származik.

Sok szerző a metódus túlterhelést (overloading) is a többalakúsághoz sorolja, hiszen ezáltal egy metódusnak több alakja lesz. Ebben a jegyzetben mi többalakúságon csak azt értjük, hogy egy objektum több osztály példányaként is használható.

### 3.2. Az OOP hasznos megoldásai

Azt gondolnánk, hogy a fenti három alapvető közül az öröklődés a legerősebb, hiszen ez teszi lehetővé, hogy nagyon egyszerűen újrahasznosítsuk az ős kódját. Lehetséges, hogy az OOP ettől lett népszerű, de az OOP igazi ereje ezekben a technikákban rejlik:

- Automatikus szemetgyűjtés (angolul: garbage collection),
- Mező, mint lokális-globális változó,
- Többalakúság használata osztály behelyettesítésre,
- Csatoltság csökkentése objektum-összetételel.

#### 3.2.1. Automatikus szemetgyűjtés

Az automatikus szemetgyűjtés nem csak az OOP nyelvekre jellemző, de az OOP nyelvekkel egyszerre lett népszerű. Leveszi a programozó válláról azt a terhet, hogy az általa lefoglalt memória ( minden new utasítás memóriát foglal ) felszabadításáról gondoskodjon. Ezt a programozó

- elfelejtheti,
- rosszul oldhatja meg (pl. túl korán szabadítja fel).

Tudjuk, hogy amit lehet rosszul csinálni, azt a programozók a nagy hajtásban általában rosszul is csinálják. Ha ezt az automatikusan is megoldható feladatot a keretrendszer végzi el, az nagyban csökkenti a fejlesztési és a tesztelési időt is. Ugyanakkor ez nem OOP specifikus tulajdonság.

#### 3.2.2. A mező mint lokális-globális változó

A mező mint lokális-globális változó egy nagyon hasznos újítás. Tudjuk, hogy sok imperatív nyelvben van globális változó. Ezek gyorsabb és kisebb kód fejlesztését teszik lehetővé, hiszen egy globális változót nem kell paraméterként átadni. Ugyanakkor a globális változók használata mellékhatást okoz.

Mellékhatásnak nevezzük, ha egy alprogram (függvény, eljárás, vagy metódus) megváltoztatja a környezetét, azaz:

- globális változóba ír,
- kimenetre (képernyőre / nyomtatóra / kimeneti portra) ír,
- fájlba ír.

Mellékhatás használatával gyorsíthatjuk a program futását, de használata nehezen megtalálható hibákat eredményez, mivel a hiba a változás helyétől távol lévő programsor hibás működését eredményezheti. Az ilyen hibák megtalálásához nem elég az új funkció részletes nyomkövetése. Gyakran az egész forráskódot muszáj átvizsgálni, ami időrabló feladat. Ezért nem tanácsos mellékhatáshoz folyamodni, azaz globális változót használni.

Mégis, a globális változók használata gyorsítja a programot és kisebb, elegánsabb a forráskód is. Tehát jó lenne, ha lenne globális változó, illetve mégse lenne jó. A mező pont ilyen, hiszen az osztályon belül globális, kívülről elérhetetlen. A mezők használatával tudunk mellékhatást előidézni, de ez az osztályon belül lokális, így az esetleges mellékhatásokból fakadó hibák könnyebben megtalálhatók.

Igazság szerint csinálhatunk teljesen globális változót is. Egy publikus osztályszintű mezőt bárhonnan írhatunk és olvashatunk, tehát az ilyen mező globális. Szerencsére az egységbezáras, illetve azon belül az információ rejtés miatt a publikus mezőket természetellenesnek érezzük, így senki sem használ már globális változókat OOP nyelveken.

### 3.2.3. [Többalakúság használata osztály behelyettesítésre](#)

A többalakúság biztosítja, hogy a kódunk rugalmas legyen. Míg az öröklődés nagyon merev struktúrákat hoz létre, addig a többalakúság a rugalmasságot szolgálja. Ennek alapja, hogy egy gyermek osztályú példány használható mindenütt, ahol ős osztály típusú paramétert várunk. Ez a többalakúság lényege.

Például könnyen készíthetünk egy pipagyár osztályt. Hogy konkrétan milyen pipát gyártunk, az csak attól függ, hogy a fapiпа vagy a tajtékpipa gyermekét példányosítjuk.

Hol van itt a többalakúság, hiszen eddig szinte csak öröklődésről beszélünk? Helyes megfigyelés, hiszen többalakúság nincs öröklődés nélkül. A gyermek osztály helyettesíthető az ős helyére. A lényeg a helyettesítésen van. Attól függ a programok működése, hogy mely gyermeket helyettesítjük be. Ezt a behelyettesítést viszont a többalakúságnak köszönhetjük, ami nem feltétlenül öröklődés útján érhető el, hanem egy interfész implementációjával is. Mikor helyettesíthetünk be egy osztályt a másik helyére? Ha ez az osztály:

- a másik osztály gyermeke,
- ha megvalósítja a várt interfészt,
- vagy megvan minden metódus, amit hívni akarok (csak a gyengén típusos nyelvek esetén).

Hol van lehetőség behelyettesítésre:

- Paraméterátadás (ős osztályú példányt várunk, de gyermeket kapunk),
- Példányosítás (a referencia ős osztály típusú, de egy gyermek példányra mutat),
- Felelősséginjektálás (kívülről kapunk egy objektumot, aminek csak a felületét ismerjük).

Láttni fogjuk, hogy minden tervezési minta a behelyettesíthetőség lehetőségén alapszik.

### 3.2.4. Csatoltság csökkentése objektum-összetételel

A csatoltság (angolul: coupling) alatt annak fokát értjük, hogy egy osztály (vagy más modul) milyen mértékben alapszik a többi osztályon. A csatoltságot szokás a kohézió (angolul: cohesion) ellentéteként értelmezni. Alacsony fokú csatolás általában magas fokú kohéziót eredményez, illetve ez a másik irányban is igaz. A csatoltság mértékét Larry Constantine csoportjának munkája alapján a következő módon számoljuk.

Definíció: OOP-ben a csatoltság annak mértéke, hogy milyen erős kapcsolatban áll egy osztály a többi osztállyal. A csatolás mértéke két osztály, mondjuk A és B között növekszik, ha:

- A-nak van B típusú mezője.
- A meghívja B valamelyik metódusát.
- A-nak van olyan metódusa, amelynek visszatérési típusa B.
- A B-nek leszármazottja, vagy implementálja B-t.

A csatoltság szintjei (legerősebbtől a leggyengébbig):

- erősen csatolt (tightly coupled)
- gyengén / lazán csatolt (loosly coupled)
- réteg (layer)

Az erős csatoltság erős függőséget is jelent. A következő fajta függőségeket szoktuk megkülönböztetni:

- Függőség a hardver és szoftver környezettől: Ha a programunk függ egy adott hardvertől vagy szoftvertől (leggyakrabban operációs rendszertől), akkor ez azt jelenti, hogy ezek speciális tulajdonságait kihasználjuk és így a programunk nem vagy csak nehezen portolható át egy másik környezetbe. Ennek egyik nagyszerű megoldása a virtuális gép használata. A forráskódunkat egy virtuális gép utasításaira fordítjuk le. Ha egy adott operációs rendszer felett egy adott hardveren fut a virtuális gép, akkor a mi programunk is futni fog.
- Implementációs függőség: Egy osztály függ egy másik implementációjától, azaz ha az egyik osztály megváltoztatása esetén meg kell változtatni a másik osztályt is, akkor implementációs függőségről beszélünk. Ez is egyfajta környezeti függés, egy osztály függ a környezetében lévő egy vagy több másik osztálytól, de itt a környezete a program forráskódja. Ha csak a másik osztály felületétől függünk, azaz teljesen mindegy, hogy hogyan implementáltuk a másik osztály metódusait, csak azok helyes eredményt adjanak, akkor nem beszélünk implementációs függőségről. Ezzel a függőséggel még részletesen fogunk foglalkozni.
- Algoritmikus függőség: Akkor beszélünk algoritmikus függőségről, ha nehézkes az algoritmusok finomhangolása. Gyakran előfordul, hogy a program egy-egy részét gyorsabbá kell tenni, mondjuk buborékos rendezés helyett gyors rendezést kell alkalmazni. Például ha a rendezés közben szemléltetjük a rendezés folyamatát, akkor nehéz lesz áttérni egyik rendezésről a másikra.

A három függőség közül csak az implementációs függőséggel foglalkozunk, de azzal nagyon részletesen. Már megemlítettük, hogy az öröklődés implementációs függőséget okoz. Nézzünk erre egy példát Java nyelven. A feladat a beépített HashSet osztály bővítése a betett elemek számolásával.

```

import java.util.*;
public class MyHashSet extends HashSet{
    private int addCount = 0;
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}

```

Ebben a példában létrehoztuk örökléssel a MyHashSet osztályt. Annyival egészítettük ki az őst, hogy számoljuk, hány elemet adunk hozzá a hasító halmazhoz. Ehhez az addCount mezőt használjuk, ami kezdetben nulla. Két metódussal lehet elemet hozzáadni a halmazhoz, az add és az addAll metódussal, ezért ezeket felülírjuk. Az add megnöveli egygyel az addCount-ot és meghívja az ős add metódusát, hiszen az tudja hogyan kell ezt a feladatot megoldani, mi csak ráültünk a megoldásra. Az addAll hasonlóan működik, de ott több elemet adunk egyszerre hozzá a listához, ezért az addCount értékét az elemek számával növeli meg.

Ezt a feladatot mindenki hasonlóan készítette volna el, hiszen a kód-újrahasznosítás legegyszerűbb formája az öröklés. Egy bökkenő van. Ez így nem megfelelően működik!

```

import java.util.*;
public class Main {
    public static void main(String[] args) {
        HashSet s = new MyHashSet();
        String[] abc = {"a", "b", "c"};
        s.addAll(Arrays.asList(abc));
        System.out.println(s.getAddCount());
    }
}

```

Ebben a példában létrehoztunk egy 3 elemű tömböt, azt addAll metódussal hozzáadtuk a MyHashSet egyik példányához. Ezután kiíratjuk, hány elemet adtunk hozzá a halmazhoz. Azt várnánk, hogy a program azt írja ki, hogy 3, de e helyett az írja, hogy 6.

Mi történt? Nem tudtuk, hogy az ősbén, azaz a HashSet osztályban, úgy van megvalósítva az addAll metódus, hogy az egy ciklusban hívogatja az add metódust, így veszi fel az elemeket. Amikor a gyermek addAll metódusát hívta, az hozzáadtott 3-at az addCount-hoz és meghívta az ős addAll metódusát. Az háromszor meghívta az add metódust. A késői kötés miatt nem az ős add metódusát, hanem a gyermek add metódusát hívta, ami szépen minden 1-gyel növelte az addCount értékét. Így jött ki a 6. Azaz az történt, hogy csúnyán ráfáztunk az öröklődés okozta implementációs függőségre.

A fenti kódot úgy lehet kijavítani, hogy csak az add metódusban növeljük az addCount értékét:

```
import java.util.*;
```

```

public class MyHashSet extends HashSet{
    private int addCount = 0;
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public int getAddCount () { return addCount; }
}

```

Amikor írjuk a gyerek osztályt, tudnunk kell, hogyan van az ős implementálva, különben hasonló nehezen megérthető problémákkal találhatjuk magunkat szembe. Ugyanakkor, ha kihasználjuk, hogy hogyan van implementálva az ős, akkor az ős változása eredményezheti, hogy a gyermeknek is változnia kell. Ez pedig implementációs függés!

Hogyan lehet ezt elkerülni? A megoldás, hogy ne öröklődést, hanem objektum-összetételt használjunk. Mondjuk, ha az A osztálynak van egy B osztály típusú mezője, akkor azt mondjuk, hogy objektum-összetételt használtunk.

Az öröklődés minden kiváltható objektum-összetételel, hiszen az alábbi két, végletekig leegyszerűsített program ugyanazt csinálja:

<pre> class A {     public void M1() {         Console.WriteLine("hello");     } }  class B : A {     public void M2() { M1(); } }  class Program {     static void Main(string[] a)     {         B b = new B();         b.M2();         Console.ReadLine();     } } </pre>	<pre> class A {     public void M1() {         Console.WriteLine("hello");     } }  class B {     A a = new A();     public void M2() { a.M1(); } }  class Program {     static void Main(string[] a)     {         B b = new B();         b.M2();         Console.ReadLine();     } } </pre>
--	---

}	
Itt a B osztály az A osztály gyermeke, így öröklí az A osztályból az M1 metódust, amit M2 metódusban hív meg. A főprogramban meghívjuk az M2 metódust, amely meghívja az ősbén lévő M1 metódust, ami kiírja, hogy hello.	Itt a B osztálynak van egy A típusú mezője. Ezt példányosítanunk kell. Az M2 metódus meghívja ezen a mezőn keresztül az M1 metódust. A főprogramban meghívjuk az M2 metódust, amely az objektum-összetétel, azaz az „a” referencián keresztül hívja az M1 metódust, ami kiírja, hogy hello.

Az objektum-összetétel nagyon rugalmas, hiszen az futási időben történik, szemben az öröklődéssel, ami már fordítási időben ismert. Ugyanakkor az öröklőést sokkal egyszerűbb felfogni, megérteni és elmagyarázni. Ezért objektum-összetételel, ami kisebb csatoltságot, kisebb implementációs függőséget és rugalmasabb kódot biztosít, csak akkor használunk, ha már sok programozói tapasztalattal bírunk.

Amikor objektum-összetételel egy metódust úgy valósítok meg, hogy annak lényegi része csak az, hogy az összetételt megvalósító referencián keresztül meghívom annak egy metódusát, akkor azt mondjuk, hogy átdelegálom a felelősséget a beágyazott objektumnak. A fenti példában ilyen az M2 metódus, ami csak meghívja az M1 metódust. A felelősségelegálás fogalmának egyik formája a .NET keretrendszer callback mechanizmusa.

Az objektum-összetételel kérdés, hogy hogyan kapjuk meg az összetételben szereplő objektumot. A fenti példában saját példányt készítettünk. Ezzel a kérdéssel részletesen foglalkozunk a felelősséginjektálás témaörén belül.

A későbbiekbén látni fogjuk, hogy habár az öröklőést mindenki lehet váltani objektum-összetételel, nem mindenki ez a célravezető, hiszen öröklődés nélkül nincs többletlakáság. Többletlakáság nélkül pedig nem lehet rugalmas kódot írni.

### 3.3. Az objektumorientált tervezés alapelvei

Az objektumorientált tervezés alapelvei (object-oriented design principles) a tervezési mintáknál magasabb absztrakciós szinten írják le, milyen a „jó” program. A tervezési minták ezeket az alapelveket valósítják meg szintén még egy elég magas absztrakciós szinten. Végül a tervezési mintákat realizáló programok az alapelvek megvalósulásai. Az alapelveket természetesen úgy is fel lehet használni, hogy nem követjük a tervezési mintákat.

A tervezési alapelvek abban segítenek, hogy több, általában egyenértékű programozói eszköz, pl. öröklődés- és objektum-összetétel közül kiválasszuk azt, amely jobb kódot eredményez. Általában jó a kód, ha rugalmasan bővíthető, újrafelhasználható komponensekből áll és könnyen érhető más programozók számára is.

A tervezési alapelvek segítenek, hogy ne essünk például abba a hibába, hogy egy osztályba kódolunk minden, hogy élvezzük a mezők mint globális változók programozást gyorsító hatását. A tapasztalat az, hogy lehet programozni az alapelvek ismerete nélkül, vagy akár tudatos megszegésével, csak nem érdemes. Gondolunk vissza a programozási technológiák alapelvére: „A program kódja állandóan változik!”. Azaz, ha rugalmatlan programot írunk, akkor a jövőben keserítjük meg az életünket, amikor

egy változást kell belehegeszteni a programunkba. Inkább érdemes a jelenben több időt rászárni a fejlesztésre és biztosítani, hogy a jövőben könnyű legyen a változások kezelése. Ezt biztosítja számunkra az alapelvek betartása.

### 3.3.1. A GOF könyv 1. alapelve – GOF1

A GOF1 alapelt a Gang of Four (GOF) könyvben jelent meg 1995-ben. A könyv magyar címe: „Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz.” A könyv angol címe: „Design Patterns: Elements of Reusable Object-Oriented Software”. Az alapelt eredeti angol megfogalmazása: „Program to an interface, not an implementation”, azaz „Programoz felületre implementáció helyett”.

Mit jelent ez a gyakorlatban? Egyáltalán, hogy lehet implementációra programozni? Miért rossz implementációra programozni? Miért jó felületre?

Akkor programozunk implementációra, ha kihasználjuk, hogy egy osztály hogyan lett implementálva. Egy példát a MyHashSet osztályon keresztül már láttunk, amikor tudnunk kellett, hogyan lett az ō implementálva. Egy másik példa:

```
class NagySzám {  
    //maximum ennyi számjegyű nagy szám  
    private const int maxHossz = 20;  
    //használt számlrendszer alapja  
    private const int alap = 10;  
    //a számjegyek fordított sorrendben vannak  
    //pl. 64 esetén: számjegyek[0]=4, számjegyek[1]=6  
    private int[] számjegyek = new int[maxHossz];  
    public NagySzám(int[] szám) {  
        Array.Copy(szám, számjegyek, szám.Length);  
    }  
    public static NagySzám Összead(NagySzám S1, NagySzám S2) {  
        int[] A = S1.számjegyek;  
        int[] B = S2.számjegyek;  
        int[] C = new int[maxHossz];  
        int átvitel = 0;  
        for(int i=0; i<maxHossz; i++) {  
            C[i] = A[i] + B[i] + átvitel;  
            átvitel = C[i] / alap; C[i] %= alap;  
        }  
        return new NagySzám(C);  
    }  
    public long ToLong() {  
        int i = maxHossz - 1; long szám = 0;  
        while (számjegyek[i] == 0 && i>0) i--;  
        for (; i >= 0; i--) {  
            szám *= alap; szám += számjegyek[i];  
        }  
        return szám;  
    }  
}
```

```
}
```

A fenti példában készítettünk egy NagySzám osztályt, amely a nagy szám számjegyeit a számjegyek tömbben tárolja. A legkisebb helyiértékű szám van a legkisebb indexen. A konstruktur ezt a tömböt tölti fel. Ezen túl két metódust látunk, az egyik az összeadás, a másik long típusú számmá alakítja vissza a számjegyek tömbben tárolt számot. A tömbben tárolt szám számrendszerének alapja az alap konstansban van eltárolva. Most 10-es számrendszer az alapértelmezett. De mi van, ha az alap megváltozik? Sajnos akkor minden kód, ami feltételezi, hogy 10-es számrendszeret használunk, az elromlik. Például az alábbi is:

```
class Program {
    static void Main(string[] args) {
        int[] a = { 3, 5 }; //53
        int[] b = { 1, 2, 3 }; //321
        NagySzám A = new NagySzám(a);
        NagySzám B = new NagySzám(b);
        NagySzám C = NagySzám.Összead(A, B);
        Console.WriteLine(C.ToInt64());
        Console.ReadLine();
    }
}
```

A fenti kód 374-et ír ki, ha az alap 10-es, 252-öt, ha az alapot átírjuk 8-ra, és így tovább. Tehát a NagySzám belső implementációja befolyásolja az őt használó osztályok működését. A problémát az okozza, hogy a bemeneti szám átalakítását lusták voltunk elvégezni, habár az a NagySzám felelőssége lenne. Az átalakítást a hívóra hagytuk, de ez rossz megoldás, mint ahogy láttuk.

A megoldás, ha egy olyan konstruktort csinálunk, ami egy long típusú számot vár. A másik konstruktort priváttá kell tenni. Ebben az esetben akármilyen belső alapot használunk, az nem fogja zavarni a többi osztályt. Tehát a jó megoldás (csak a megváltozott és az új kódot mutatjuk):

```
class NagySzám {
    ...
    private NagySzám(int[] szám) { // ez mostmár privát
        Array.Copy(szám, számjegyek, szám.Length);
    }
    public NagySzám(long szám) { //új konstruktur
        int i = 0;
        while (szám > 0) {
            számjegyek[i] = (int)(szám % alap);
            szám /= alap;
            i++;
        }
    }
    ...
}
class Program {
    static void Main(string[] args) {
        NagySzám A = new NagySzám(53);
```

```

NagySzám B = new NagySzám(321);
NagySzám C = NagySzám.Összead(A, B);
Console.WriteLine(C.ToString()); //374
Console.ReadLine();
}
}

```

Itt már akármilyen számrendszert használ a NagySzám, minden 374 lesz az eredmény.

Látható, hogy általában akkor kényszerülünk implementációra programozni, ha az osztály felelősségi körét rosszul határoztuk meg és egy osztály több felelősségi kört is lefed, vagy egy felelősséget nem teljesen fed le, mint a NagySzám. Tehát, ha a kódunkban találunk olyan részt, amely egy másik osztály implementációjától függ, akkor az hibás tervre utal.

Ha implementációra programozunk, és ha megváltozik az osztály, akkor a vele kapcsolatban álló osztályoknak is változniuk kell. Ezzel szemben, ha felületre programozunk, és megváltozik az implementáció, de a felület nem, akkor nem kell megváltoztatni a többi osztályt.

### 3.3.2. A GOF könyv 2. alapelve – GOF2

A GOF2 alapelve, hasonlóan a GOF1-hez, a GOF könyvben jelent meg 1995-ben. Az alapelve eredeti angol megfogalmazása: „Favor object composition over class inheritance”, azaz „Használj objektum-összetételel öröklés helyett, ha csak lehet”.

Mit jelent ez a gyakorlatban? Egyáltalán mit jelent az objektum-összetétel? Miért jobb az öröklődésnél? Mi a baj az öröklődéssel? Ha jobb az objektum-összetétel, akkor miért nem minden azt használjuk?

Már láttuk, hogy objektum-összetételel mindenki lehet váltani az öröklődést. Az öröklés azért jó, mert megörököljük az ōs összes szolgáltatását (metódusait), amit használni tudunk. Objektum-összetételelnél ezen osztály egy példányára szerzik egy referenciát és azon keresztül használjuk a szolgáltatásait. Ez utóbbi futási időben dinamikusan változhat, hiszen az, hogy melyik objektumra mutat a referencia, futási időben változtatható.

#### 3.3.2.1. Az IS-A és a HAS-A kapcsolat

Az öröklődést IS-A kapcsolatnak hívjuk. Ha a Kutya osztály a Gerinces osztály gyermeke, akkor azt mondjuk, hogy „a kutya egy gerinces”, azaz angolul „the dog 'is a' vertebrate”. Innen jön az IS-A elnevezés.

IS-A kapcsolatnak nevezzük azt is, amikor egy osztály megvalósít egy interfész, hiszen ekkor az osztály példányai megkapják az interfész típusát is.

Az objektum-összetételel HAS-A kapcsolatnak hívjuk. Ha a Kutya osztályban van egy gerinc nevű mező, ami Gerinces osztály típusú, akkor azt mondjuk, hogy „a kutynak van egy gerince”, azaz angolul „the dog 'has a' backbone”. Innen jön a HAS-A elnevezés.

A következő példában a Kutya és a Gerinces osztály között IS-A kapcsolat, a Kutya2 és a Gerinces osztály között pedig HAS-A kapcsolat van.

```
class Gerinces {
```

```

        public void LábVezérlés() {
            Console.WriteLine("mozog a lába.");
        }
    }
    class Kutyा : Gerinces {
        public void Fut() {
            Console.Write("Gyorsan ");
            LábVezérlés();
        }
    }
    class Kutyа2 {
        Gerinces gerinc;
        public Kutyа2(Gerinces gerinc) { this.gerinc = gerinc; }
        public void Fut() {
            Console.Write("Gyorsan ");
            gerinc.LábVezérlés();
        }
    }
    class Program {
        static void Main(string[] args) {
            Kutyа bodri = new Kutyа();
            bodri.Fut();
            Kutyа2 rex = new Kutyа2(new Gerinces());
            rex.Fut();
            Console.ReadLine();
        }
    }
}

```

Figyeljük meg, hogy minden megoldás esetén a `Fut` metódus ugyanúgy működik. Ez a példa is mutatja, hogy az öröklődés kiváltható objektum-összetétellel, azaz az IS-A kapcsolat kiválható HAS-A kapcsolattal.

### 3.3.2.2. Átlátszó és átlátszatlan újrahasznosítás

Az öröklődést néha átlátszó újrahasznosításnak (angolul: white box reuse) is hívjuk. Ezzel arra utalunk, hogy az örökölt metódusokat használhatjuk és azokról sok információ van, gyakran ismerjük a forráskódjukat is.

Az objektum-összetételt átlátszatlan újrahasznosításnak (angolul: black box reuse) is hívjuk. Ezzel arra utalunk, hogy az összetételt megvalósító mezőn keresztül hívhatunk metódusokat, de azok megvalósításáról általában nincs információink.

### 3.3.2.3. Aggregáció és kompozíció

Az objektum-összetételt, vagy más néven a HAS-A kapcsolatot több szempontból is lehet osztályozni. Az első szempont a birtoklás módja, a második a becsomagolás (angolul: wrapping) módja.

Az első szempont azt helyezi a középpontba, hogy az objektum-összetétel minden birtoklást fejez ki. Pl. a kutyának van farka. A második szempont pedig azt hangsúlyozza ki, hogy az objektum-összetétnél a birtokolt objektumot a birtokló becsomagolja. Pl. a tűzfal becsomagol egy szervet, amely csak rajta keresztül érhető el.

A birtoklás módja szerint az objektum-összetétel lehet aggregáció és a kompozíció:

- Aggregáció (angolul: aggregation): A birtokolt példány nem csak az enyém, azt más is használhatja. Például a kutyának van gazdija, de a gazdi nem csak a kutyáé. Ennek a példának megfelelő kódrészlet:

```

class Ember { }
class Kutyta
{
    Ember gazdi; // HAS-A kapcsolat, amit a gazdi nevű mező valósít meg
    public Kutyta(Ember gazdi) { this.gazdi = gazdi; }
}
class Program
{
    public static void Main()
    {
        Ember gabi = new Ember();
        Kutyta buksi = new Kutyta(gabi); // gabi nem csak buksi gazdija
        Kutyta cézár = new Kutyta(gabi); // nem kizárolagos tulajdonlás
    }
}

```

- Kompozíció (angolul: composition): A birtokolt példány csak az enyém, azt más nem is ismerheti. Például a kutyának van farka, azt csak ő csóválhatja. Ennek a példának megfelelő kódrészlet:

```

class KutytaFarok { public void Csóvál() { } }
class Kutyta
{
    KutytaFarok farok; // HAS-A kapcsolat, amit a farok nevű mező valósít meg
    public Kutyta() { farok = new KutytaFarok(); } // kizárolagos tulajdonlás
    public void Csóvál() { farok.Csóvál(); }
}

```

Vizsgáljuk meg az aggregációt és a kompozíciót. Vegyük a következő esetet: „A gitárosnak van egy gitárja.” Ugyebár ez egy objektum-összetétel, hiszen HAS-A kapcsolat van a gitáros és a gitár között. Hogy melyik fajta összetételekkel kell választani, azt egy egyszerű kérdéssel lehet eldönten: Ha a gitáros meghal, vele temetik a gitárját is? Ha igen, akkor kompozícióról beszélünk, ha nem, aggregációról. Azaz, ha senki másnak nincs rá referencia, és ezért a szemétygyűjtő felszabadítja, akkor kompozíció. Aggregációra szép példa többek közt a stratégia tervezési minta. Kompozícióra szép példa az állapot tervezési minta.

### 3.3.2.4. Átlátszó és átlátszatlan becsomagolás

Az objektum-összetétel soha nem öncélú, hanem van rá valamilyen jó okunk. Ez az ok nagyon gyakran az, hogy a saját metódusaink, más néven szolgáltatásaink, megírásához szükségünk van olyan metódusok hívására, amit a birtokolt objektumból lehet hívni. Más szóhasznállattal azt mondjuk: A saját szolgáltatásainkhoz szükségünk van a birtokolt objektum szolgáltatásaira. Ha azért birtoklunk egy objektumot, hogy a saját szolgáltatásaink felelősséget részben vagy egészben átadjuk neki (idegen szóval delegáljuk), akkor birtoklás mellett becsomagolásról is beszélünk.

A becsomagolás módja szerint a becsomagolás lehet átlátszó vagy átlátszatlan:

- Átlátszó becsomagolás: A becsomagolt példány ugyanolyan felületű, mint a becsomagoló, azaz a becsomagolt objektum szolgáltatásai elérhetők a becsomagolón keresztül. Amikor hívjuk az A metódust, akkor az meghívja a becsomagolt objektum A metódusát. Más szóval, az A

metódus átadja (idegen szóval delegálja) a felelősséget a becsomagolt objektum A metódusának. Ezt szemlélteti a lenti példa. A megvalósításához kell egy IS-A és egy HAS-A kapcsolat is. A HAS-A kapcsolat lehet aggregáció és kompozíció is. Például a karácsonya karácsonya marad, akárhány díszt is teszünk rá. Ezt úgy érjük el, hogy a dísszel becsomagoljuk a karácsonyát, úgy hogy az objektum majd minden, díszítéstől nem függő, metódusa egyszerűen meghívja a becsomagolt karácsonya ugyanolyan nevű metódusát. Azért van értelme, hogy a díszítés után a karácsonya karácsonya maradjon, mert így további díszeket lehet rátenni, lásd a díszítő tervezési mintát.

Példa kódrészlet, ami követi a fenti magyarázatot:

```
class GömbDísz : Karácsonya // IS-A kapcsolat a Karácsonya osztállyal
{
    Karácsonya kf; // HAS-A kapcsolat a Karácsonya osztállyal
    public GömbDísz(Karácsonya kf) { this.kf = kf; }
    public override string GetTípus()
    {
        return kf.GetTípus(); // felelősség átadás
    }
    public void A() { kf.A(); } // felelősség átadás általánosítva
    public override string ToString()
    {
        return "Díszes " + kf.ToString(); // részleges felelősség átadás
    }
}
```

- Átlátszatlan becsomagolás: A becsomagolt példány nem ugyanolyan felületű, mint a becsomagoló, így a becsomagolt objektum szolgáltatásai rejtve maradnak, kívülről nem érhetők el. Ugyanakkor az elérhető szolgáltatások elvégzéséhez felhasználhatók a becsomagolt objektum szolgáltatásai. Ilyenkor nem elvárás, hogy ugyanaz legyen a metódus neve, mint aminek delegáljuk a felelősséget. Azaz A metódusból nem biztos, hogy A metódust kell hívnom, hanem akár hívhatom a B metódust, vagy a C metódust, attól függően, hogy melyik valósítja meg azt a működést, amire szükségünk van. A megvalósításához csak egy HAS-A kapcsolat kell. A HAS-A kapcsolat lehet aggregáció és kompozíció is. A fenti példa itt is használható, de ebben az esetben, a becsomagolt karácsonya nem marad karácsonya, azaz nincs IS-A kapcsolat, és ezért nem lehet újabb díszeket rátenni.

Példa kódrészlet, ami követi a fenti magyarázatot:

```
class GömbDísz // nincs IS-A kapcsolat a Karácsonya osztállyal
{
    Karácsonya kf; // csak HAS-A kapcsolat van
    public GömbDísz(Karácsonya kf) { this.kf = kf; }
    public string GetTípus() { return kf.GetTípus(); } // felelősség átadás
    public void A() { kf.B(); } // felelősség átadás általánosítva
    public override string ToString()
    {
        return "Díszes " + kf.ToString(); // részleges felelősség átadás
    }
}
```

Vizsgáljuk meg az átlátszó és az átlátszatlan becsomagolást is. Az átlátszó becsomagolás általában aggregáció, de lehet kompozíció is. Ilyenkor a becsomagolt osztállyal gyermek (IS-A) és összetétel (HAS-A) viszonyban is állnak: „Az ős gyermeké vagyok, hogy ős típusként használható legyek. Illetve

becsomagolom az ōsöm egy példányát, hogy azon keresztül használhassam a szolgáltatásait.” Erre szép példa a dekorátor tervezési minta. Átlátszatlan becsomagolás esetén: „Szintén használom a becsomagolt objektum szolgáltatásait, de azokat nem publikálom külvilág felé.” Erre szép példa az illesztő tervezési minta.

### 3.3.2.5. Rész – egész viszony

A kompozíció megfogalmazható úgy is, hogy a rész nem létezhet az egész nélkül. A kutya farka csak a kutyával együtt létezhet, ha nincs kutya, akkor nincs farka sem. Ezzel szemben aggregáció esetén a rész létezhet az egész nélkül is. Az autóból kiszedhető az ülés, az ülésen lehet ülni anélkül is, hogy az egy autó része lenne.

Fordítva is érdemes megvizsgálni a kérdést. Az egész létezhet-e a részei nélkül. Objektum-összetétel esetén az egész nem létezhet a részei nélkül, ugyanakkor a részei lecserélhetők menet közben a program futása alatt. Ez adja az objektum-összetétel rugalmasságát.

A következő példa azt mutatja, hogy egy autó objektum sima ülés részét lecseréljük sport ülésre:

```
abstract class Ülés { /*...*/ }
class SimaÜlés : Ülés { /*...*/ }
class SportÜlés : Ülés { /*...*/ }
class Autó
{
    Ülés ülés;
    public Auto(Ulés ülés) { this.ulés = ülés; }
    public void SetÜlés(Ulés ülés) { this.ulés = ülés; }
}
class Program
{
    public static void Main()
    {
        Autó autó1 = new Autó(new SimaÜlés()); // az autóban sima ülés van
        autó1.SetÜlés(new SportÜlés()); // az ülést lecseréljük sport ülésre
    }
}
```

### 3.3.2.6. Összegző példa objektum-összetételre

Nézzünk egy szép példát objektum-összetételre.

```
class Alváz { /*...*/ }
class Kaszni { /*...*/ }
class Motor { /*...*/ }
class Autó
{
    Alváz alváz;
    Kaszni kaszni;
    Motor motor;
    public Auto(Alváz alváz, Kaszni kaszni, Motor motor)
    {
        this.alváz = alváz;
        this.kaszni = kaszni;
```

```

        this.motor = motor;
    }
}

```

A fenti példában hármas objektum-összetételt látunk. Az autó 3 fő alkatrészről áll, ezek aggregációja adja az autót. Az is látszik a fenti példából, hogy ez egy átlátszatlan becsomagolás, hiszen az Autó és a többi három osztály között csak HAS-A kapcsolat van, IS-A kapcsolat nincs.

### 3.3.2.7. Az objektum-összetétel és a csatoltság

Csatoltság szempontjából az öröklődés a legerősebb, majd jön a kompozíció és az aggregáció. Éppen ez az oka, hogy a GOF2 kimondja, hogy használunk inkább objektum-összetételt öröklődés helyett, hiszen az kisebb csatoltságot eredményez és így rugalmasabb kódot kapunk. Ugyanakkor ki kell emelni, hogy az ilyen kód nehezebben átlátható, ezért nem szabad túlzásba vinni az objektum-összetételt.

Ha egy osztálynak van olyan mezője, amely lehet null értékű, akkor az a mező általában nem objektum-összetételt valósít meg, hanem az objektum-összetételtől lazább kapcsolatot, a barátságot (angolul: I: dependency). A barátság egy átmeneti kapcsolat, gyakran nem is kell hozzá mező, elegendő annyi, hogy az objektum egy metódusában kap egy referenciát egy másik metódusra, kér tőle valamit és el is felejti.

### 3.3.2.8. GOF2 a gyakorlatban,

Egy másik ok, ami miatt nem váltunk ki minden öröklődést objektum-összetételel, az az, hogy öröklődés nélkül nincs többlelképesség (legalábbis erősen típusos nyelvek esetén). Jól tudjuk, hogy egy osztályhierarchia tetején lévő osztály példánya helyett bármelyik gyermek osztály példányát használhatjuk. Erre gyakran van szükségünk, hiszen így tudunk a változásokhoz könnyen alkalmazkodni. Például van egy gyermek osztályunk, ami Windows speciális, egy másik Unix speciális, az egyik környezetben az egyiket, a másikban a másikat használom.

### 3.3.3. Egy felelősség - egy osztály alapelve – SRP (Single Responsibility Principle)

Az Egy felelősség - egy osztály alapelve (angolul: Single Responsibility Principle – SRP) azt mondja ki, hogy minden osztálynak egyetlen felelősséget kell lefednie, de azt teljes egészében. Eredeti angol megfogalmazása: „A class should have only one reason to change”, azaz „Az osztályoknak csak egy oka legyen a változásra”.

Már a GOF1 elvnél is láttuk, hogy ha egy osztály nem fedi le teljesen a saját felelősségi körét, akkor muszáj implementációra programozni, hogy egy másik osztály megvalósítsa azokat a szolgáltatásokat, amik kimaradtak az osztályból.

Ha egy osztály több felelősségi kört is ellát, például a MacsKuty eszik, alszik, ugat, egerészik, akkor sokkal jobban ki van téve a változásoknak, mintha csak egy felelősséget látna el. A MacsKuty osztályt meg kell változtatni, ha kiderül, hogy a kutyák nem csak a postást ugatják meg, hanem a bicikliseket is, illetve akkor is, ha a macskák viselkedése változik vagy bővül.

Már láttuk, hogy minden módosítás magában hordozza a veszélyt, hogy egy forráskódszörnyet kapjunk, amihez már senki se mer hozzájárni. Az ilyen kód fejlesztése nagyon drága.

Gyakran szembesülünk azzal, hogy mi szeretnénk, hogy minden osztálynak csak egy oka legyen a változásra, azaz egy felelősségi kört lásson el, de minden osztálynak kell naplóznia vagy a

jogosultságokat ellenőriznie. Erre ad megoldást az aspektusorientált programozás. Ezeket a felelősségeket, mint például a naplózás, kiemeljük egy úgynevezett aspektusba, amit bármely osztályhoz hozzákapcsolhatunk.

Az egy felelősség - egy osztály elvére szép példa a felelősséglánc tervezési minta.

### 3.3.4. [Nyitva-zárt alapelvek – OCP \(Open-Closed Principle\)](#)

Az Open-Closed Principle (OCP), magyarul a nyitva-zárt elv, kimondja, hogy a program forráskódja legyen nyitott a bővítésre, de zárt a módosításra. Eredeti angol megfogalmazása: „Classes should be open for extension, but closed for modification”.

Egy kicsit szűkebb értelmezésben, az osztályhierarchiánk legyen nyitott a bővítésre, de zárt a módosításra. Ez azt jelenti, hogy új alosztályt vagy egy új metódust nyugodtan felvehetünk, de meglévőt nem írhatunk felül. Ennek azért van értelme, mert ha már van egy működő, letesztelt, kiforrott metódusom és azt megváltoztatjuk, akkor több negatív dolog is történhet:

- a változás miatt az eddig működő ágak hibásak lesznek,
- a változás miatt a vele implementációs függőségen lévő kódrészleteket is változtatni kell,
- a változás általában azt jelenti, hogy olyan esetet kezelek le, amit eddig nem, azaz bejön egy új if vagy else, esetleg egy switch, ami csökkenti a kód átláthatóságát, és egy idő után már senki se mer hozzányúlni.

Az OCP elvet meg lehet fogalmazni a szintaxis szintjén is C# nyelv esetén: Ne használd az override kulcsszót, kivéve ha

- absztrakt vagy
- horog (angolul: hook)

metódust akarsz felülírni.

Ugyebár az absztrakt metódusokat muszáj felülírni, de ez nem az OCP megszegése, hiszen az absztrakt metódusnak nincs törzse, így lényegében a törzzsel bővítjük a kódot, nem módosítunk semmit. A másik eset, amikor horog (angolul: hook) metódust írunk felül. Akkor beszélek horog metódusokról, ha a metódusnak van ugyan törzse, de az teljesen üres. Ezek felülírása nem kötelező, csak opcionális, így arra használják őket, hogy a gyermek osztályok opcionálisan bővíthessék viselkedésüket. Ezek felülírásával lényegében megint csak bővítjük a kódot, nem módosítjuk, azaz nem szegjük meg az OCP elvet.

A horog metódus törzse üres, vagy csak egy return utasítás van benne egy alapértelmezett értékkel. Ez utóbbira példa C# nyelven a `ToString` vagy Java nyelven a `toString` metódus. Tehát ha a `ToString` metódust írjuk felül, azzal még nem szegjük meg az OCP elvet.

A következő rövid példában nem tarjuk be az OCP elvet:

```
class Alakzat
{
    public const int TEGLALAP = 1;
    public const int KOR = 2;
```

```

        int tipus;
    public Alakzat(int tipus) { this.tipus = tipus; }
    public int GetTipus() { return tipus; }
}
class Teglalap : Alakzat{ Teglalap():base(Alakzat.TEGLALAP) {} }
class Kor : Alakzat{ Kor():base(Alakzat.KOR) {} }
class GrafikusSzerkeszto
{
    public void RajzolAlakzat(Alakzat a)
    {
        if (a.GetTipus() == Alakzat.TEGLALAP) RajzolTeglalap(a);
        else if (a.GetTipus() == Alakzat.KOR) RajzolKor(a);
    }
    public void RajzolKor(Kor k) { /* ... */ }
    public void RajzolTeglalap(Teglalap t) { /* ... */ }
}

```

Ha egy kódban if – else if szerkezetet látunk, akkor az valószínűleg azt mutatja, hogy nem tartottuk be az OCP elvet. Nem tartottuk be, hiszen, ha új alakzatot akarunk hozzáadni a kódhoz, akkor az if – else if szerkezetet tovább kell bővítenünk. Lássuk, hogy lehet ezt kivédeni:

```

abstract class Alakzat{ public abstract void Rajzol(); }
class Teglalap : Alakzat
{
    public override void Rajzol() { /* téglalapot rajzol */ }
}
class Kor : Alakzat
{
    public override void Rajzol() { /* kört rajzol */ }
}
class GrafikusSzerkeszto
{
    public void RajzolAlakzat(Alakzat a) { a.Rajzol(); }
}

```

A fenti példában bevezettünk egy közös ōst, az absztrakt Alakzatot. A konkrét alakzatok csak felülírják az ōs absztrakt Rajzol metódusát és kész is az új gyermek. Ebből akárhányat hozzáadhatunk, a meglévő kódot nem kell változtatni. Tehát itt betartjuk az OCP elvet.

A közös absztrakt ōs másik előnye az, hogy ha a kódban a gyermek osztály példányait csak az közös ōs felületén keresztül használjuk, akkor ezzel betartjuk a GOF1 ajánlást is.

Az OCP elv alkalmazására nagyon szép példa a stratégia és a sablonmetódus tervezési minta. Az utóbbi hook metódusokra is ad példát.

### 3.3.5. Liskov-féle behelyettesítési alapelve – LSP (Liskov Substitutional Principle)

A Liskov-féle behelyettesítési elv, rövid nevén LSP, kimondja, hogy a program viselkedése nem változhah meg attól, hogy az ōs osztály egy példánya helyett a jövőben valamelyik gyermek osztályának példányát használom. Azaz a program által visszaadott érték nem függ attól, hogy egy Kutya vagy egy

Vizsla vagy egy Komondor példány lábainak számát adom vissza. Eredeti angol megfogalmazása: „If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T”.

Nézzünk egy példát, amely nem felel meg az LSP elvnek. A klasszikus ellenpélda az ellipszis – kör illetve a téglalap – négyzet példa. A kör olyan speciális ellipszis, ahol a két sugár egyenlő. A négyzet olyan speciális téglalap, ahol az oldalak egyenlő hosszúak. Szinte adja magát, hogy a kör az ellipszis alosztálya, illetve a négyzet a téglalap alosztálya legyen. Lássuk a téglalap – négyzet példát:

```
class Téglalap
{
    protected int a, b;
    // @ utófeltétel: a == x és b == \régi(b)
    public virtual void SetA(int x) { a = x; }
    public virtual void SetB(int x) { b = x; }
    public int Terület() { return a * b; }
}
class Négyzet : Téglalap
{
    // invariáns: a == b;
    // utófeltétel: a == x && b == x;
    public override void SetA(int x) { a = x; b = x; }
    public override void SetB(int x) { a = x; b = x; }
}
```

A fenti példában az a és b mezőt használjuk a téglalap oldalhosszainak tárolására. Mindkét mezőhöz tartozik egy szetter metódus. A Négyzet osztályban a két szetter metódust felül kellett írni, mert a négyzet két oldala egyenlő. Azt mondjuk, hogy ez a Négyzet osztály invariánsa, mert minden metódushívás előtt és után igaznak kell lennie, hogy a két oldal egyenlő. A SetA metódusnak megadtuk az utófeltételét is. A gond az, hogy a Négyzet osztályban a SetA utófeltétele gyengébb, mint a Téglalap osztályban. Pedig, mint látni fogjuk, a gyermek osztályban az utófeltételeknek erősebbeknek, az előfeltételeknek gyengébbeknek kellene lennie, hogy betartsuk az LSP elvet.

```
class Program
{
    public static void Main(string[] args)
    {
        Random rnd = new Random();
        for (int i = 0; i < 10; i++)
        {
            Téglalap rect;
            if (rnd.Next(2) == 0) rect = new Téglalap();
            else                  rect = new Négyzet();
            rect.SetA(10);
            rect.SetB(5);
            Console.WriteLine(rect.Terület());
        }
        Console.ReadLine();
    }
}
```

```
}
```

A fenti főprogram 50%-os valószínűséggel a Téglalap osztályt, 50%-os valószínűséggel ennek gyermek osztályát a Négyzetet példányosítja. Ha az LSP igaz lenne, akkor minden osztály példányán keresztül hívjuk a Terület metódust, de ez nem igaz, mert a SetA és a SetB teljesen más képp viselkedik a két osztályban. Ennek megfelelően egyszer 50, egyszer 25 lesz a kiírt érték. Azaz a program viselkedése függ attól, melyik példányt használjuk, azaz az LSP elvet megszegtük.

Mi is volt a tényleges probléma a fenti példában? A probléma az, hogy a Négyzet alosztálya a Téglalapnak, de nem altípusa.

### 3.3.6. [Szerződésalapú programozás – Design by Contract](#)

Az altípus fogalmának megadásához be kell vezetnünk a szerződésalapú programozás (angolul: Design by Contract) fogalmait:

- előfeltétel, amely a bemenetre és a mezőkre ad megkötést,
- utófeltétel, amely a kimentre és a mezőkre ad megkötést,
- invariáns, amely csak a mezőkre ad megkötést.

Megjegyezzük, hogy a szerződésalapú programozás magyar nyelvű irodalmában használatos a szerződésalapú tervezés is.

A metódus előfeltétele írja le, hogy milyen bementre működik helyesen a metódus. Az előfeltétel általában a metódus paraméterei és az osztály mezői segítségével írja le ezt a feltételt. Például az Osztás(int osztandó, int osztó) metódus előfeltétele, hogy az osztó ne legyen nulla.

A metódus utófeltétele írja le, hogy milyen feltételnek felel meg a visszaadott érték, illetve milyen állapotátmenet történt, azaz az osztály mezői hogyan változnak a metódushívás hatására. Például a Maximum(int X, int Y) utófeltétele, hogy a visszatérési érték X, ha  $X > Y$ , egyébként Y.

A metódus szerződése az, hogy ha a hívó úgy hívja meg a metódust, hogy igaz az előfeltétele, akkor igaz lesz az utófeltétele is a metódus lefutása után. Az előfeltétel és az utófeltétel így két állapotot kötöti átmenetet ír le, a metódus futása előtti és utáni állapotot. Az elő- és utófeltétel párok megadása helyett adhatunk egy úgynévezett állapotátmeneti megszorítást (ez ugyanazt a feladatot látja el, mint a Turing-gépek delta függvénye, csak predikátumként megadva), ami leírja az összes lehetséges állapotátmenetet. E helyett a szakirodalom ajánlja még a történeti megszorítás (angolul: history constraint) használatát, de erre nem térünk ki részletesen.

Ezen túl még beszélünk osztályinvariánsról is. Az osztályinvariáns az osztály lehetséges állapotait írja le, azaz az osztály mezőire ad feltételt. Az invariánsnak minden metódushívás előtt és után igaznak kell lennie.

Tegyük fel, hogy az N(égyzet) osztály gyermeke a T(églalap) osztálynak. Azt mondjuk, hogy az N egyben altípusa is a T osztálynak akkor és csak akkor, ha

- a T mezői felett az N invariánsából következik a T invariánsa,
- T minden metódusára igaz, hogy

- a metódus T-ben megadott előfeltételéből következik az N megadott előfeltétele,
- a metódus N-ben megadott utófeltételéből következik a T megadott utófeltétele,
- a metódus N-ben csak olyan kivételeket válthat ki, amelyek megegyeznek vagy gyermekei a T megadott kivételeknek. Megjegyzés: Ezt a Java esetén a fordító helyettünk ellenőrzi, de C# esetén a kiváltott kivételek nem részei a metódus fejének, így a fordító nem tudja ezt helyettünk ellenőrizni.
- a T mezői felett az N állapot átmeneti megszorításából következik a T állapot átmeneti megszorítása.

Az utolsó feltételre azért van szükség, mert a gyermek osztályban lehetnek új metódusok is, és ezeknek is be kell tartaniuk az ős állapot átmeneti megszorítását. Ha az ősben „egyes” állapotból nem lehet közvetlenül elérni a „hármas” állapotot, akkor ezt a gyermekben sem szabad.

A Téglalap – Négyzet példában az invariánsra vonatkozó feltétel igaz, hiszen a Téglalap invariánsa IGAZ, a Négyzeté pedig a  $a == b$ , és könnyen belátható, hogy  $a == b$  formulából következik az IGAZ. Az előfeltételekre vonatkozó feltétel is igaz. Az utófeltételek feltétele viszont hamis, mert a SetA metódus esetén az  $a == x \text{ ÉS } b == x ==> a == x \text{ ÉS } b == \text{\regi}(b)$  állítás nem igaz. Ezért a Négyzet nem altípusa a Téglalapnak.

Az altípus definícióját informálisan gyakran így adjuk meg:

- az ős mezői felett az altípus invariánsa nem gyengébb, mint az ősé,
- az altípusban az előfeltételek nem erősebbek, mint az ősben,
- az altípusban az utófeltételek nem gyengébbek, mint az ősben,
- az altípus betartja ősének történeti megszorítást (history constraint).

Erősebb feltételt úgy kapok, ha az eredeti feltételhez ÉS-sel veszek hozzá egy plusz feltételt. Gyengébb feltételt úgy kapok, ha az eredeti feltételhez VAGY-gyal veszek hozzá egy plusz feltételt. Egy kicsit könnyebb ezt megérteni, ha halmazokkal fogalmazzuk meg. Mivel a gyengébb feltétel nagyobb halmazt, az erősebb feltétel pedig kisebb halmazt jelent, a fenti definíció így is megadható:

- az ős mezői felett a belső állapotok halmaza kisebb vagy egyenlő az altípusban, mint az ősben,
- minden metódus értelmezési tartománya nagyobb vagy egyenlő az altípusban, mint az ősben,
- minden metódusra a metódus hívása előtti lehetséges belső állapotok halmaza nagyobb vagy egyenlő az altípusban, mint az ősben,
- minden metódus értékkészlete kisebb vagy egyenlő az altípusban, mint az ősben,
- minden metódusra a metódus hívása utáni lehetséges belső állapotok halmaza kisebb vagy egyenlő az altípusban, mint az ősben,
- az ős mezői felett a lehetséges állapotátmenetek halmaza kisebb vagy egyenlő az altípusban, mint az ősben.

Ha a Téglalap – Négyzet példában betartottuk volna az OCP elvet, akkor az LSP elvet se sértettük volna meg. Hogy lehet betartani az OCP elvet ebben a példában? Úgy, hogy egyáltalán nem készítünk SetA és SetB metódust, mert akkor azokat mindenképpen felül kellene írni. Csak konstruktort készítünk és a területmetódust. Az OCP és az LSP általában egymást erősítik.

### 3.3.7. Interfészszegregációs-alapelv – ISP (Interface Segregation Principle)

Az interfészszegregációs-alapelv (angolul: Interface Segregation Principle – ISP) azt mondja ki, hogy egy sok szolgáltatást nyújtó osztály fölé el kell helyezni interféseket, hogy minden kliens, amely használja az osztály szolgáltatásait, csak azokat a metódusokat lássa, amelyeket ténylegesen használ. Eredeti angol megfogalmazása: „No client should be forced to depend on methods it does not use”, azaz „Egy kliens se legyen rászorítva, hogy olyan metódusoktól függön, amiket nem is használ”.

Ez az elv segít a fordítási függőség visszaszorításában. Képzeljük csak el, hogy minden szolgáltatást, például egy fénymásoló esetén a fénymásolást, nyomtatást, faxküldést, a példányok szétválogatását egy nagy Feladat osztály látna el. Ekkor, ha a fénymásolás rész megváltozik, akkor újra kell fordítani a Feladat osztályt és lényegében az egész alkalmazást, mert mindenki innen hívja a szolgáltatásokat. Ez egy néhány 100 ezer soros forráskód esetén bizony már egy kávészünetnyi idő. Nyilván így nem lehet programot fejleszteni.

A megoldás, hogy minden klienshez (kliensnek nevezzük a forráskód azon részét, ami használja a szóban forgó osztály szolgáltatásait) készítünk egy interfészt, amely csak azokat a metódusokat tartalmazza, amelyeket a kliens ténylegesen használ. Tehát lesz egy fénymásoló, egy nyomtató, egy fax és egy szétválogatás interfész. A Feladat ezen interfések mindegyikét implementálja. Az egyes kliensek a Feladat osztályt a nekik megfelelő interfészen keresztül fogják csak látni, mert ilyen típusú példányként kapják meg. Ezáltal ha megváltozik a Feladat osztály, akkor az alkalmazásnak csak azt a részét kell újrafordítani, amit érint a változás.

Az ilyen monumentális osztályokat, mint a fenti példában a Feladat, kövér osztályoknak nevezzük. Gyakran előfordul, hogy egy sovány kis néhány száz soros osztály elkezd hízni, egyre több felelősséget lát el, és a végén egy kövér, sok ezer soros osztályt kapunk. A kövér osztályokat az egy felelősség - egy osztály elv (SRP) kizárra, de ha már van egy ilyen osztályunk, akkor egyszerűbb fölé tenni néhány interfészt, mint a kövér osztályt szétszedni kisebbekre. Egy egyszerű példa:

```
interface IWorkable { void Work(); }
interface IFeedable { void Eat(); }
interface IWorker : IFeedable, IWorkable {}
class Worker : IWorker
{
    public void Work() { /*.dolgozik */ }
    public void Eat() { /*.eszik */ }
}
class Program
{
    public static void Main(String[] args)
    {
        IWorkable workable = new Worker();
        IFeedable feedable = new Worker();
        IWorker worker = new Worker();
    }
}
```

Ha betartjuk az interfész szegregációs elvet, akkor a forráskód kevésbé csatolt lesz és így egyszerűbben változtatható. Erre az elvre szép példa az illesztő tervezési minta.

### 3.3.8. Függőség megfordításának alapelve – DIP (Dependency Inversion Principle)

A függőség megfordításának elve (angolul: Dependency Inversion Principle – DIP) azt mondja ki, hogy a magas szintű komponensek ne függjenek alacsony szintű implementációs részleteket kidolgozó osztályuktól, hanem épp fordítva, a magas absztrakciós szinten álló komponensektől függjenek az alacsony absztrakciós szinten álló modulok. Eredeti angol megfogalmazása: „High-level modules should not depend on low-level modules. Both should depend on abstractions.” Azaz: „A magas szintű modulok ne függjenek az alacsony szintű moduluktól. Mindkettő függön az absztrakciótól.” Ezt ennél frappánsabban így szoktuk mondani: „Absztrakciótól függi, ne függi konkrét osztályuktól”.

Az alacsony szintű komponensek újrafelhasználása jól megoldott az úgynevezett osztálykönyvtárak (library) segítségével. Ezekbe gyűjtjük össze azokat a metódusokat, amikre gyakran szükségünk van. A magas szintű komponensek, amik a rendszer logikáját írják le, általában nehezen újrafelhasználhatók. Ezen segít a függőség megfordítása. Vegyük a következő egyszerű leíró nyelven íródott kódot:

```
public void Copy() { while( (char c = Console.ReadKey()) != EOF) Printer.printChar(c); }
```

Itt a Copy metódus függ a Console.ReadKey és a Printer.printChar metódustól. A Copy metódus fontos logikát ír le, a forrásból a célra kell másolni karaktereket file vége jelig. Ezt a logikát sok helyen fel lehet használni, hiszen a forrás bármilyen lehet és a cél is, ami karaktereket tud beolvasni, illetve kiírni. Ha most ezt a kódot újra akarom hasznosítani, akkor két lehetőségem van. Az első, hogy if – else – if szerkezet segítségével megállapítom, hogy most melyik forrásra, illetve célra van szükségem. Ez nagyon csúnya, nehezen átlátható, módosítható kódot eredményez. A másik lehetőség, hogy a forrás és a cél referenciáját kívülről adja meg a hívó felelőssége injektálásával (angolul: dependency injection).

A felelősség injektálásának több típusa is létezik:

- Felelősség injektálása konstruktőrral: Ebben az esetben az osztály a konstruktőrén keresztül kapja meg azokat a referenciaikat, amiket keresztül a neki hasznos szolgáltatásokat meg tudja hívni. Ezt más néven objektum-összetételnek is nevezzük és a leggyakrabban épp így programozzuk le. (Javascript keretrendszerknél gyakran alkalmazott módszer, Angular JS)
- Felelősség injektálása szetter metódusokkal: Ebben az esetben az osztály szetter metódusokon keresztül kapja meg azokat a referenciaikat, amikre szüksége van a működéséhez. Általában ezt csak akkor használjuk, ha opcionális működés megvalósításához kell objektum-összetételt alkalmaznunk. (Java esetén gyakran alkalmazott módszer, JEE7)
- Felelősség injektálása interfész megvalósításával. Ha a példányt a magas szintű komponens is elkészítheti, akkor elegendő megadni a példány interfészét, amit általában maga a magas szintű komponens valósít meg, de paraméteresztály paramétereiként is jöhet az interfész.
- Felelősség injektálása elnevezési konvenció, konfigurációs állomány, vagy annotáció alapján. Ez általában keretrendszerkre jellemző. Ezeket csak tapasztalt programozóknak ajánljuk, mert nyomkövetéssel nem lehet megtalálni, hogy honnan jön a példány és ez nagyon zavaró lehet.

A fenti egyszerű Copy metódus a függőségmegfordítás elvének megfelelő változata látható a következő példában. A fenti lehetőségek közül a felelősség beinjektálás konstruktőrrel megoldást választottuk.

```

class Source2Sink
{
    private System.IO.Stream source;
    private System.IO.Stream sink;
    public Source2Sink(Stream source, Stream sink)
    {
        this.source = source;
        this.sink = sink;
    }
    public void Copy()
    {
        byte b = source.ReadByte();
        while (b != 26)
        {
            sink.WriteByte(b);
            b = source.ReadByte();
        }
    }
}

```

Sokan kritizálják a függőség megfordításának elvét, miszerint az csak az objektum-összetétel használatának, azaz a GOF2 elvnek egy következménye. Mások szerint ez egy önálló tervezési minta. mindenestre a haszna vitathatatlan, ha rugalmas kód fejlesztésére törekszünk.

### 3.3.9. További tervezési alapelvek

Itt emlíjtük meg azokat a tervezési alapelveket, amelyek a szakirodalomban kevésbé elfogadottak, ugyanakkor mégis érdemes megismerkedni velük.

#### 3.3.9.1. Hollywood alapelv – HP (Hollywood Principle)

A Hollywood alapelv eredeti angol megfogalmazása: „Don’t call us, we’ll call you”, azaz „Ne hívj, majd mi hívunk”. A Hollywood alapelvet a következő példával szemléltethetjük: Rómeó szerepére szereplőválogatást hirdetnek. Több száz jelentkező van. A válogatás után mindenki szeretné megtudni, Ő kapta-e a hőn áhitott szerepet. Két megoldás van:

- mindenki kisebb-nagyobb időközönként érdeklődik, Ő kapta-e a szerepet. Ilyenkor a titkár egyre idegesebben válaszol, hogy még nincs döntés, hívjon később. Ez a „busy waiting”.
- következő alkalommal a titkár már jó előre közli minden színéssel, ne hívj, majd mi hívunk. Azaz senki se érdeklődjön, ha majd megvan a döntés, mindenkit értesítünk, hogy megkapta-e a szerepet. Ez a Hollywood elv alkalmazása.

A busy waiting nagyon káros, mert foglalja a processzoridőt, lassítja a többi szálat. Tipikus megoldása, hogy egy végtelen ciklusban egy sleep hívással várunk, majd hívjuk a metódust, ami megmondja, hogy várni kell-e még. Ha már nem kell várni, akkor egy break utasítással kilépünk a ciklusból.

A busy waiting megoldásnak van létfogalma, de csak nagyon kevés helyzetben. A legismertebb a megfigyelő kutya, angolul watch dog, amikor egy távoli objektumot kérdezgetünk (ping-elgetünk) megadott időközönként, hogy él-e még. Ezt máshogyan nem tudjuk megoldani, hiszen, ha elmegy az

áram, a távoli gép nem tud még egy üzenetet sem küldeni, hogy mostantól elérhetetlen lesz. Ha a figyelő kutya észreveszi, hogy lehalt a távoli objektum, akkor annak feladatát másra osztják.

A Hollywood elv azt mondja ki, hogy ne az kérdezgessen, aki az eseményre vár, hanem az esemény értesítse a várakozókat. Ezt a megoldást használja például a Java eseménykezelése. Ha lenyomok egy gombot, akkor keletkezik egy esemény, de ezen túl semmi se történik. Ha azt akarom, hogy történjen is valami, akkor fel kell íratnom az eseményre egy figyelőt (Java nyelvhasználattal listener). Ha kiváltódik az esemény, akkor az összes feliratkozott figyelő értesítést kap. Pontosan ezt valósítja meg a figyelő tervezési minta.

A Hollywood elv akkor ad nagy segítséget, ha egy-több kapcsolatban vannak az objektumok és a több oldal dinamikusan változik, azaz fel is lehet iratkozni, meg le is. Egyik alternatívája az üzenetsugárzás (angolul: broadcasting), amikor egy üzenet mindenki máshoz eljut. Ekkor az üzenet küldője nem feltétlenül ismeri az üzenet fogadóját, ami előny lehet. Hátránya, hogy olyan objektum is megkaphatja az üzenetet, akit nem érdekel.

### 3.3.9.2. [Demeter törvénye / a legkisebb tudás elve](#)

Demeter törvénye, vagy más néven a legkisebb tudás elve (angolul: Law of Demeter / Principle of Least Knowledge) kimondja, hogy egy osztály csak a közvetlen ismerőseit hívhatja. Eredeti angol megfogalmazása: „Talk only to your immediate friends”. Azaz: „Csak a közvetlen ismerőseiddel beszélj”.

Praktikusan úgy is megfogalmazhatjuk ezt az elvet, hogy csak annak a példánynak a metódusát hívhatjuk, akire van referencia, azaz az A.getB().C() alakú hívások tilosak. Ez az elv azért hasznos, mert ha betartjuk, akkor a változások minden csatlakoztatott objektumhoz lesznek.

## 4. Programozási technológiák – Tervezési minták

A tervezési minták gyakori programozói feladatokat oldanak meg. Gyakorlott programozók, miután már sokszor megoldottak egy-egy problémát, desztillálták a bevált megoldások lényegét. Így születtek a tervezési minták, ezek gyűjteményei. Ezek közül az első a GOF könyv volt, majd ezt több is követte. Ezek közül a legjelentősebbek:

- Eric Freeman, Elisabeth Robson, Kathy Sierra, Bert Bates: Head First Design Patterns, O'Reilly Media, 2004.
- Cay S. Horstmann: Object-Oriented Design and Patterns, Wiley, 2006.
- Robert C. Martin: Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2002.

Ebben a jegyzetben elsősorban a GOF könyvben ismertetett mintákat nézzük át.

A minták alkalmazásával könnyen bővíthető, módosítható osztály szerkezetet kapunk, tehát rugalmas kódot. Az ár, amit ezért fizetünk, a bonyolultabb, nehezebb átlátható kód és a nagyobb fejlesztési idő. Sokan azt mondják, hogy ez nem fizetődik ki. Törekedjünk a legegyszerűbb megoldásra (lásd extrém programozás) és ha kell, kódszépítéssel általánosítsuk a kódot. Így érjük el a rugalmas kódot.

A tervezési minták viszonylagos bonyolultsága abból adódik, hogy olyan osztályokat tartalmaznak, amiknek semmi köze valóságos objektumokhoz, habár azt tanultuk, hogy egy OOP osztály a valóság absztraktív modellje. Ugyanakkor ezekre a technikai osztályokra szükség van a rugalmassághoz. Ezek azok az osztályok, amiket józan paraszti ésszel nehéz kitalálni, de nem is kell, mert a legjobb megoldások tervezési minták formájában rendelkezésre állnak.

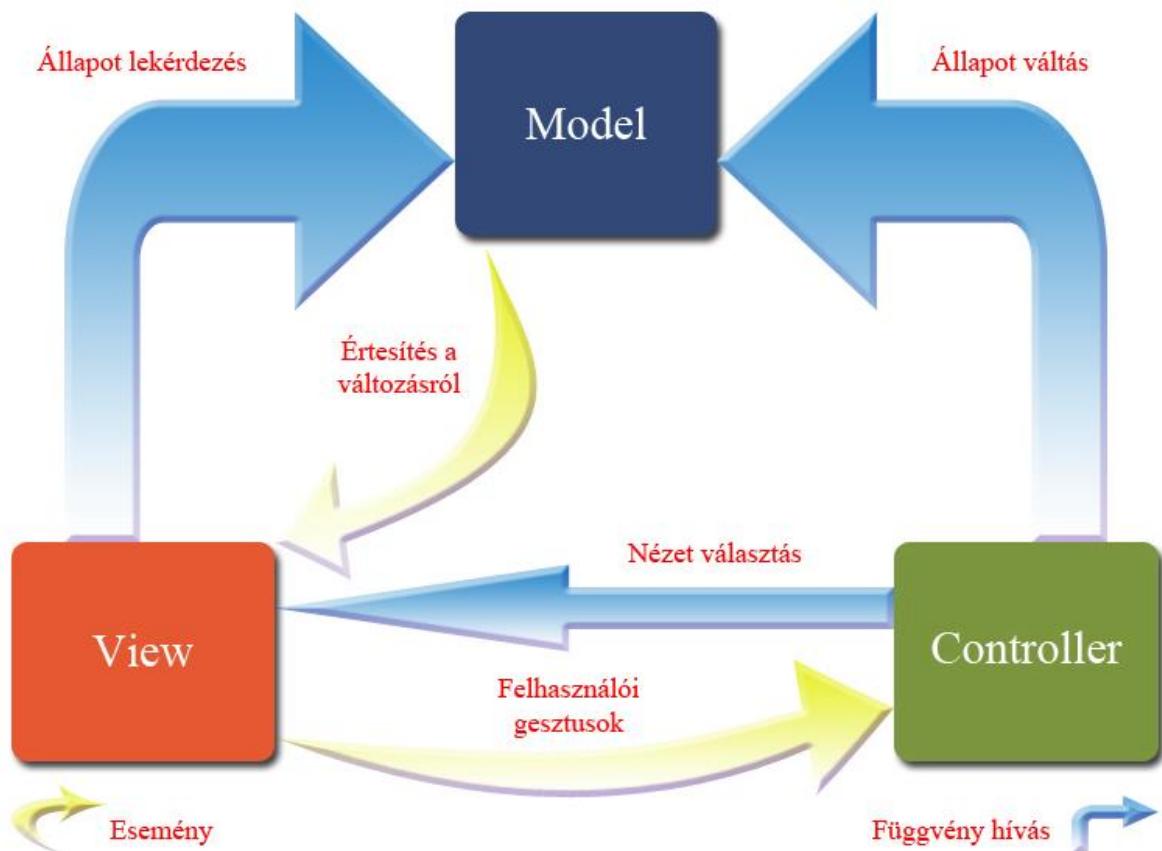
### 4.1. Architekturális minták

Az architektúra a program azon része, ami nem változik az idő során, vagy ha változik, akkor az nagyon nehezen kivitelezhető. Talán egy szívatültetéshez vagy agyműtéthez hasonlítható.

#### 4.1.1. MVC – Model-View-Controller

Az MVC minta talán az első tervezési minta. A nevét a három fő komponensének nevéből kapta:

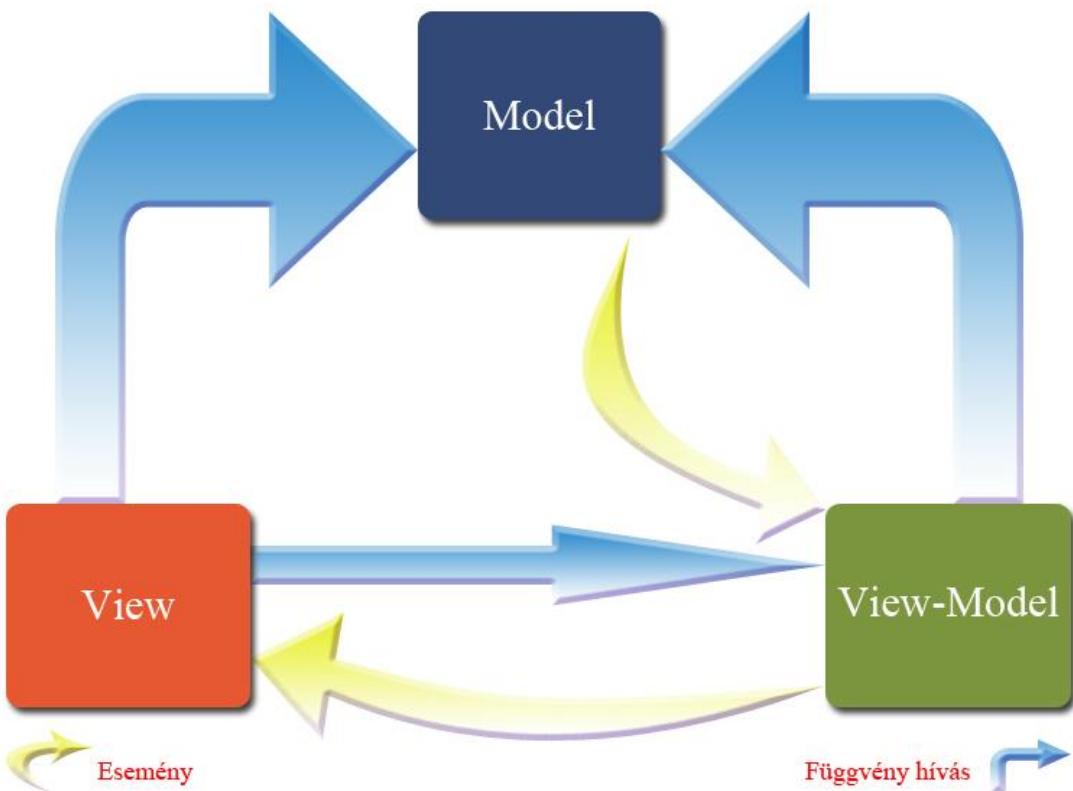
- Model (magyarul modell): Az adatokat kezelő réteg. Ez felel az adatok tárolásáért, visszaolvasásáért. Itt foglalnak helyet azok a függvények is, amik műveleteket végeznek az adatokon. Része az adatbázis is.
- View (magyarul nézet): A felhasználói felület megjelenítéséért, a felhasználó különféle nyűjjeinek a Vezérlő felé továbbításáért felelős réteg. Itt jelennek meg a felhasználó számára a vezérlőelemek, a felhasználónak szánt adatok megfelelő formában való megjelenítése is itt történik.
- Controller (magyarul vezérlő): Ez a réteg a vezérlőelemek eseményeinek alapján meghívja a modell megfelelő függvényeit, illetve ha a megjelenítésben érintett adatok változnak, akkor erről értesíti a Nézetet. Általában itt kap helyet az üzleti logika.



14. ábra: Az MVC modell

Az alkalmazást három egységre bontjuk. Külön egység felelős a megjelenítésért, az adatok kezeléséért valamint a felhasználói cselekedetek megfelelő kezeléséért. Ez több okból is jó nekünk, legelőször is, ha lecseréljük valamelyik részt, akkor a többi még maradhat, nem kell hozzányúlni, több időnk marad (munkaidőben játszani:). Könnyebben módosíthatjuk az egyes részeket.

Az MVC egyik fő újítása az volt, hogy lehetővé tette, hogy egy modellhez több nézet is tartozzon. minden nézet ugyanannak a modellnek a belső állapotát jeleníti meg. Bármielyik nézeten lenyomnak egy gombot, az az esemény eljut a kontrollernek. A kontroller meghívja a modell megfelelő metódusát. Ha e miatt a modell belső állapota megváltozik, akkor a modell a megfigyelő tervezési mintának megfelelően értesíti a nézeteket, hogy változás történt, nekik is meg kell változni.



15. ábra: Az MVVM modell

Az MVC mintának több továbbfejlesztése is létezik. Ezek közül a két legismertebb:

- MVP – Model View Presenter, magyarul Modell – Nézet – Megjelenítő: Ebben a változatban a modell nem a nézetet, hanem a megjelenítőt értesíti, ha változás történik. A megjelenítő lekéri az adatokat a modellből, feldolgozza, és megformázza a nézet számára.
- MVVM – Model View View-Model, magyarul Modell – Nézet – Nézetmodell: Ez az MVP továbbfejlesztése, ahol a nézetben a lehető legkevesebb logika van. A nézetmodell elvégez minden feladatot a nézet helyett, csak a megjelenítés marad a nézetre.

#### 4.1.2. ASP.NET MVC Framework

Az ASP.NET MVC Framework az ASP.NET Web Forms alapú fejlesztésnek nyújt alternatívát MVC alapú webalkalmazások fejlesztéséhez. ASP.NET MVC Framework egy olyan könnyű és jól tesztelhető megjelenítő keretrendszer, amely (csakúgy, mint az ASP.NET Web Forms) integrálja a már meglévő ASP.NET lehetőségeit, mint például a master page-eket és a beépített felhasználó kezelést, azaz membership provider alapú azonosítást. Az MVC alapjait a System.Web.Mvc névtér definiálja, amely a System.Web névtér támogatott része.

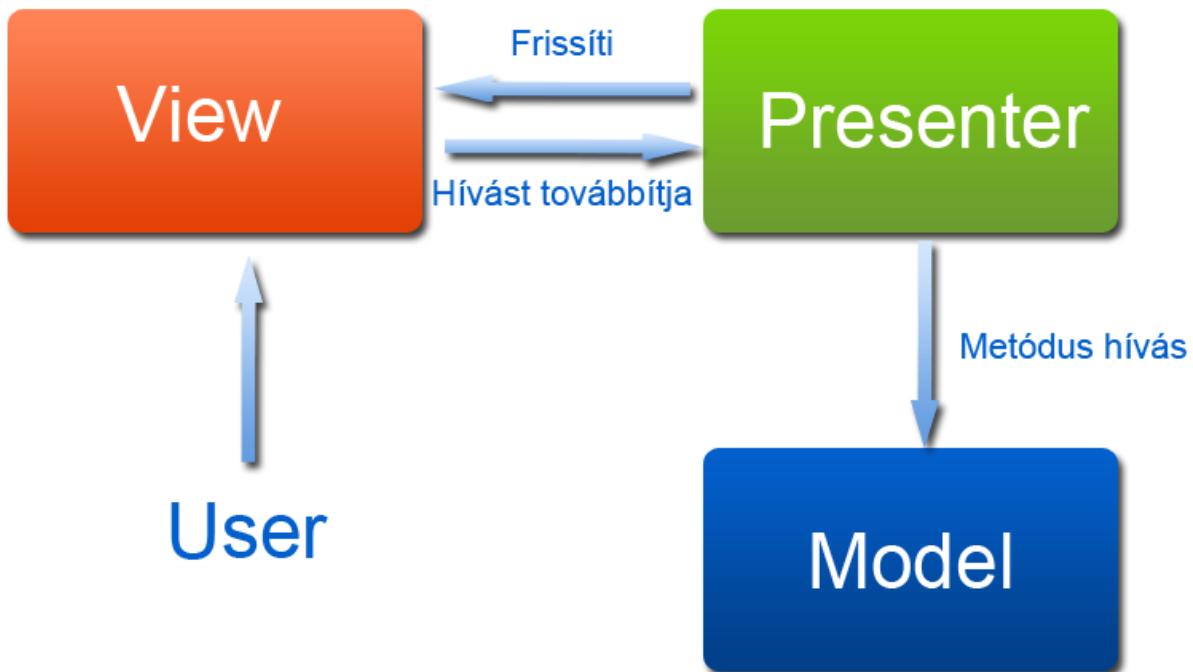
Az MVC egy alapvető programtervezési minta, amely számos fejlesztőnek már ismerős lehet. Néhány webalkalmazás már régóta használja az MVC keretrendszer előnyeit, míg mások továbbra is az ASP.NET hagyományos Web Forms-os postback alapú rendszert használják. Egyesek pedig ötvözik a két rendszer előnyeit. Azt később tárgyaljuk, hogy az MVC fejlesztési mód mikor előnyös.

Az MVC framework három fő komponenst foglal magában:

- Modell (angolul: Model): A modellobjektumok az alkalmazás azon részei, amelyek az adatokat "szállító" logikát implementálják. A modellobjektumok gyakran fogadnak adatokat az adatbázisból és tárolják azokat magukban. Például egy Termékobjektum lekérhet adatokat adatbázisból, dolgozhat vele, majd a módosított adatokat visszaírhatja a Terméktáblába az SQL Szerveren.

Kisebb alkalmazások esetében a modellek inkább koncepcionálisak, mint fizikailag megvalósítottak. Például ha az alkalmazás kizárolag olvassa és megjeleníti az adatokat, akkor nincs konkréten megvalósítva a modellréteg és a hozzá tartozó osztályszerkezet. Ebben az esetben a modellréteget csak az adattoló objektumok reprezentálják.

- Nézet (angolul: View): A nézetek a felhasználói felület (angolul: User Interface, röviden: UI) megjelenítő komponensei. A UI általában azokból az adatokból készül, amelyek a modellrétegből jönnek. Ilyen lehet például egy szerkesztő nézete a Terméktáblának, amely állhat szövegdobozokból, gombokból, lenyíló menükből stb., melyek a Termékobjektum aktuális állapotát mutatják.
- Vezérlő (angolul: Controller): A vezérlők azok a komponensek, melyek a felhasználói interakciót kezelik, dolgoznak a modellobjektumokkal és kiválasztják a megfelelő nézetet a megjelenítéshez. Egy MVC-alkalmazásban a nézet csak információt jelenít meg; a vezérlő kezeli és reagálja le a felhasználói interakciót. Például a vezérlő kezeli a query sztring értékeket, továbbítja a modell felé, melyek a megfelelő adatbázis-lekérdezést állítják össze az átadott értékek alapján.



16. ábra: Az MVP modell

Az MVC minta olyan alkalmazások elkészítésében nyújt segítséget, melyek szétválasztják az alkalmazás különböző részeit (input logika, üzleti logika, megjelenítési logika), miközben egy laza csatolófelületet biztosít a szétválasztott részek között. A minta meghatározza azt is, hogy melyik logikai rétegnek hol kell elhelyezkednie az alkalmazásban. A megjelenítési vagy UI réteg a nézetekhez kötődik, az üzleti logika a vezérlőkhöz, az input logika pedig a modellekhez tartozik. Ez a szeparáció segít kezelní a komplexitást az alkalmazás fejlesztésekor, mivel lehetővé teszi, hogy az implementáció során egy adott időben adott szemszögből vizsgáljuk a dolgokat. Például a megjelenítési réteg fejlesztésekor nem kell foglalkoznunk azzal, hogy az üzleti logikai rétegen milyen műveleteket kell végezni az adattal, hiszen a nézeteken keresztül csak megjelenítjük őket.

Ráadásul a komplexitás kezelésében az MVC minta könnyebbé teszi az alkalmazás tesztelését, mint egy Web Forms alapú fejlesztési modellben. Például Web Forms alapú webalkalmazásban egyetlen osztály felelhet a megjelenítésért és a felhasználói interakcióért is. Automata teszteket írni Web Forms alapú alkalmazásokhoz bonyolult lehet, mert egyedülálló oldal teszteléséhez példányosítani kell az oldal osztályát, az összes gyerekvezérlőt és további függő osztályokat is. Mivel az oldal futtatásához ennyi osztály példányosítására van szükség, nehéz olyan tesztet írni, amely az oldal egyes részeivel kizárolagosan foglalkozik. Kijelenthetjük tehát, hogy Web Forms alapú környezetbe sokkal nehezebb a tesztelést integrálni, mint egy MVC-t használó alkalmazásba. Továbbá Web Forms-os környezetben a teszteléshez szükségeltetik egy webszerver is. Mivel az MVC keretrendszer szétválasztja a komponenseket és ezek között interfészket használ, könnyebb különálló komponensekhez teszteket gyártani az izoláció miatt.

A laza kötés az MVC-alkalmazás három fő komponense között párhuzamos fejlesztést is lehetővé tesz. Ez azt jelenti, hogy egy fejlesztő dolgozhat a kinézeten, egy második a vezérlő logikán, egy harmadik pedig az üzleti logikára fókuszálhat egy időben.

#### 4.1.2.1. [Mikor készítsünk MVC-alkalmazást](#)

Körültekintően kell megválasztanunk, mikor használunk ASP.NET MVC keretrendszeret a fejlesztéshez az ASP.NET Web Forms helyett, ugyanis az ASP.NET MVC nem helyettesíti a Web Forms modellt; használhatjuk mindenkor egyszerre egy alkalmazáson belül is akár.

Mielőtt az MVC keretrendszer használata mellett döntünk a Web Forms modell helyett, mérlegeljük mindenkor előnyeit.

#### 4.1.2.2. [Az MVC alapú webalkalmazás előnyei](#)

Az ASP.NET MVC keretrendszer a következő előnyöket nyújtja:

- Könnyebbé teszi komplex alkalmazások fejlesztését azzal, hogy három részre osztja az alkalmazást: modellre, nézetre és vezérlőre.
- Nem használ állapottárolást (view state) és szerveroldali form-okat sem. Ez ideálissá teszi az MVC keretrendszeret azok számára, akik teljes hatalmat szeretnének az alkalmazás viselkedése felett.
- Egy fő vezérlőn mintán keresztül dolgozza fel a webalkalmazáshoz érkező kéréseket, innen továbbítja a megfelelő vezérlőknek tovább. A fő vezérlőről az MSDN weboldalán lehet több információhoz hozzájutni a Front Controller szekció alatt.
- Segítséget nyújt a teszt-vezérelt fejlesztéshez (test-driven development - TDD)

- Jól működik olyan webalkalmazások esetében, amelyet fejlesztők csapata fejleszt és támogat, ahol a kinézettervezőknek magas fokú ellenőrzésre van szüksége az alkalmazás viselkedése felett.

#### 4.1.2.3. A Web Forms alapú alkalmazás előnyei

A Web Forms alapú keretrendszer előnyei:

- Támogatja az eseménykezelés modellt és megőrzi az állapotokat HTTP protokoll felett, mely előnyös az ún. "line-of-business" webalkalmazás fejlesztésnél. A Web Forms alapú alkalmazás tucatnyi eseménykezelőt biztosít, amit több száz szerverkontrollból elérhetünk.
- Page Controller mintát használ, melyek különálló tulajdonságokkal ruházzák fel az egyes oldalakat. További információ a Page Controller-ről az MSDN weboldalán található.
- Állapottárolást (view state) és szerveroldali form-okat használ, melyek megkönnyítik az állapotkezelési információk menedzselését.
- Jól működik kisebb fejlesztői csoportban, számos komponens felhasználható segítve a gyors alkalmazásfejlesztést.
- Összességében kevésbé összetett alkalmazás fejlesztés szempontjából, mert a komponensek (a Page osztály, vezérlők stb.) szorosan integráltak így általában kevesebb kódolást igényel, mint az MVC modell.

#### 4.1.2.4. Az ASP.NET Framework tulajdonságai

Az ASP.NET MVC Framework a következő funkciókat nyújtja:

- Az alkalmazás feladatainak szeparálása (input logika, üzleti logika, megjelenítési logika), tesztelhetőség és teszt-vezérelt fejlesztés támogatása alapból. Az MVC összes mag eleme interfész alapú, mely lehetővé teszi az úgynevezett "mock" objektumokkal való tesztelést, amelyek olyan objektumok, amik imitálják az aktuális objektum viselkedését az alkalmazásban. Lehetővé teszi az egységeszt alapú tesztelést anélkül, hogy a vezérlőket egy ASP.NET folyamaton keresztül futtatnunk kellene, így flexibilissé és gyorrá téve az egységesztelést.
- Egy kiterjeszthető és bővíthető keretrendszer. Az ASP.NET MVC keretrendszer komponensei úgy lettek lefejlesztve, hogy azok könnyen testre szabhatóak, ill. lecserélhetőek legyenek.
- Az URL-mapping (útvonal-feltérképezés) komponens lehetővé teszi olyan alkalmazások fejlesztését, amelyek érhető és kereshető URL-ekkel rendelkeznek. Az URL-ek nem tartalmaznak fájlnév kiterjesztésekét, kereső- és felhasználóbarát.
- Támogatja a meglévő ASP.NET oldalak (.aspx fájlok), felhasználói vezérlők (.ascx fájlok) és master page-ek (.master fájlok) használatát. Használhatjuk a meglévő ASP.NET lehetőségeit, mint a beágyazható (angolul: nested) master page-ek használatát, valamint az ASP.NET jelölőnyelvén belüli szerveroldali kód (pl. C#) használatát a <%= %> kifejezés segítségével.
- Meglévő ASP.NET funkciók támogatása. Az ASP.NET MVC lehetőséget ad a beépített lehetőségek használatára, mint a form autentikáció, Windows autentikáció, felhasználó kezelés (membership és roles), session kezelés stb.

#### 4.1.3. Többrétegű architektúra

Réteg alatt a program olyan jól elszeparált részét értjük, amely akár külön számítógépen futhat. A rétegek jól definiált interfészeken keresztül kommunikálnak, mindenkor csak a felettük és alattuk lévő

réteggel kommunikálhatnak. A rétegek annyira lazán csatoltak a többihez, hogy egy réteg a többi réteg számára észrevétlenül lecserélhető, feltéve, hogy ugyanazt az interfészt használja, mint az elődje.

A többrétegű architektúra akárhány rétegből állhat. Minél több a rétegek száma, annál rugalmasabb a rendszer, de ezzel szembeható, hogy annál nehezebb a karbantartása. A legismertebb többrétegű architektúra a 3 rétegű (angolul: 3-tier programming). Itt a három réteg:

- Felhasználói felület
- Üzleti logika
- Adatbázis

A felhasználói felület gyakran grafikus, így csak a GUI (angolul: Graphical User Interface) rövidítéssel hivatkozunk rá. Az üzleti logikát (angolul: business logic) angol neve után gyakran BL-nek rövidítjük. Az adatbázis (angolul: database) réteget gyakran perzisztencia rétegnek hívjuk és általában DB-nek rövidítjük az angol neve után.

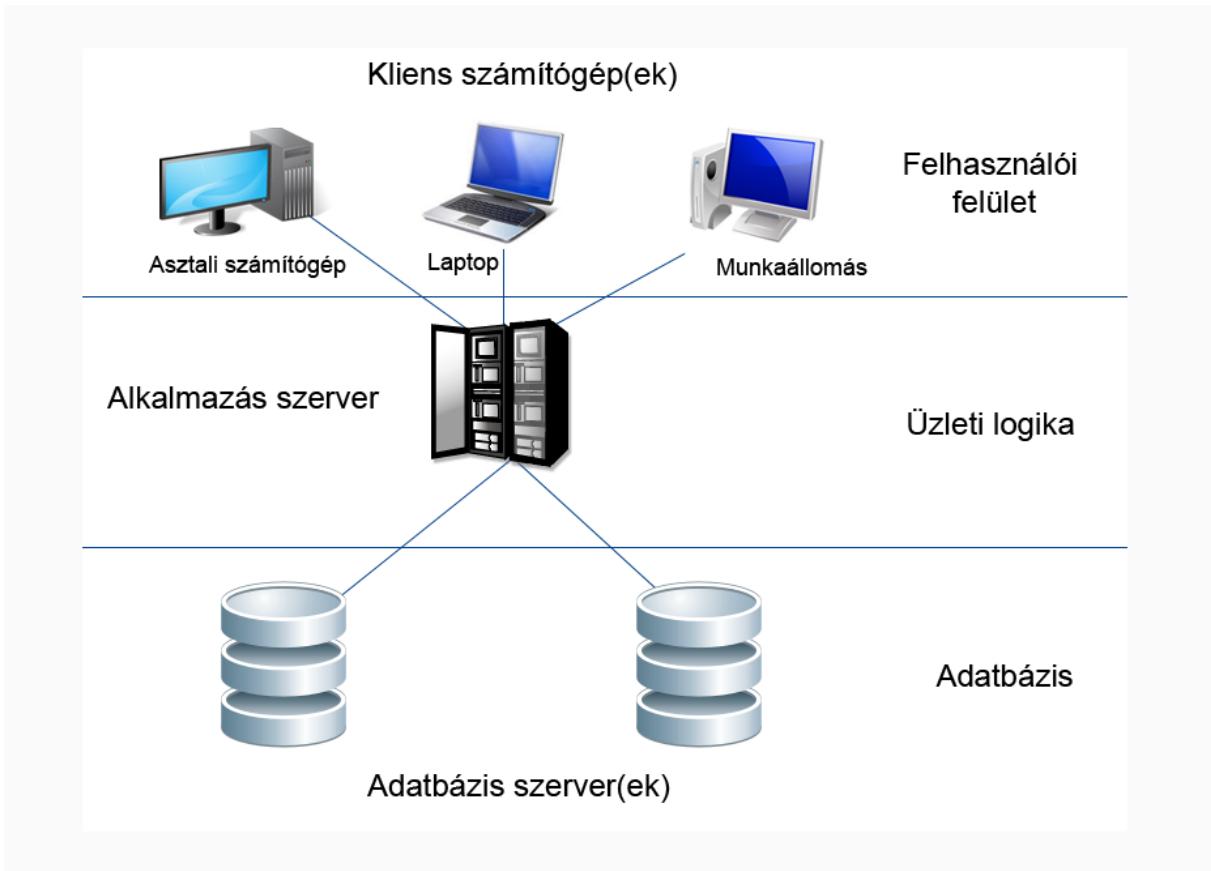
Hiba azt gondolni, hogy a háromrétegű architektúra csak az MVC minta másik neve. Az első esetén a felhasználói felület nem kommunikálhat az adatbázisréteggel, tehát ez egy lineáris rendszer a kommunikáció útját tekintve. Ezzel szemben az MVC háromszög alakú, hiszen a modell közvetlenül értesíti a nézeteket, ha megváltozik.

A 3 rétegű architektúra általában három számítógépet használ:

- kliens
- alkalmazásszerver
- adatbázisszerver

A kliens lehet vastag vagy vékony kliens. A vékony kliens csak egy böngésző, illetve a benne futó weboldal. A vastag kliens egy általában C# vagy Java nyelven megírt önálló alkalmazás. Mindkettőnek van előnye és hátránya:

Vékony kliens	Vastag kliens
Szegényes felhasználói élmény.	Gazdag felhasználói élmény.
Nem kell frissíteni. Nem kell a frissítéseket eljuttatni a felhasználóhoz.	Hibajavítás, új verzió kiadása csak frissítéssel lehetséges.
Kicsi hardverigény.	Magas hardverigény.
A kliens számítógép erőforrásai csak részben állnak rendelkezésre.	A kliens számítógép erőforrásaihoz hozzáférhet, pl. állományt írhat, olvashat.
Fő hátránya a szegényes felhasználó élmény, de ez AJAX technológiával gazdagabbá tehető.	Fő hátránya a nehézkes frissítés, de ez történhet automatikusan is, ha van internetkapcsolat.
Látható, hogy a két technológia előnyei és hátrányai kezdenek kiegyenlítődni, ezért a vékony kliens használata egyre terjed.	



17. ábra: A háromrétegű szoftver architektúra

Az alkalmazásszerveren (hardver értelemben) JavaEE platform esetén alkalmazásszerver (szoftver értelemben) fut. Ez megkönnyíti az alkalmazás fejlesztését, mert néhány szerverfunkciót, pl. a terheléselosztást (load balancing) megold helyettünk az alkalmazásszerver.

#### 4.2. Létrehozási tervezési minták

A létrehozási tervezési minták olyan tervezési minták, amelyek objektumok gyártásának bevált módszereit mutatják be.

A létrehozási minták feladata, hogy megszüntessék a sok, new kulcsszóval ránk szakadó függőséget. Ha úgy írjuk meg a programunkat, hogy mindenhol a new Kutya() hívást írjuk, amikor Kutya példányra van szükségünk, akkor nehéz lesz ezt lecserélni egy későbbi new SzuperKutya() hívásra. Jobban járunk, ha a „gyártást” a létrehozási mintákra hagyjuk és például így készítjük a kutyáinkat: kutyaGyár.createKutya(). Ilyenkor, ha változnak a követelmények, akkor csak egy helyen kell változtatni a létrehozás módját, ott, ahol létrehozzuk a kutyaGyár példányt.

##### 4.2.1. Egyke – Singleton

Gyakori feladat, hogy egy osztályt úgy kell megírnunk, hogy csak egy példány lehet belőle. Ez nem okoz gondot, ha jól ismerjük az objektumorientált programozás alapelveit. Tudjuk, hogy az osztályból példányt a konstruktőrrel készíthetünk. Ha van publikus konstruktur az osztályban, akkor akárhány példány készíthető belőle. Tehát publikus konstruktora nem lehet az egykének. De ha nincs konstruktur, akkor nincs példány, amin keresztül hívhatnánk a metódusait. A megoldás az osztályszintű

metódusok. Ezeket akkor is lehet hívni, ha nincs példány. Az egykének van egy osztályszintű szerezPéldány (angolul: getInstance) metódusa, ami mindenkinél ugyanazt a példányt adja vissza. Természetesen ezt a példányt is létre kell hozni, de a privát konstruktort a szerezPéldány hívhatja, hiszen ő is az egyke osztály része.

#### 4.2.1.1. Forráskód

```
using System;

namespace Singleton
{
    public class Singleton
    {
        // statikus mező az egyetlen példány számára
        private static Singleton uniqueInstance=null;
        // privát konstruktur, hogy ne lehessen 'new' kulcsszóval példányosítani
        private Singleton() { }
        // biztosítja számunkra a példányosítást és egyben visszaadja a példányt
        // mindenkinél ugyanazt
        public static Singleton GetInstance()
        {
            if (uniqueInstance==null) // megvizsgálja, hogy létezik-e már egy példány
            {
                uniqueInstance = new Singleton(); // ha nem, akkor létrehozza azt
            }
            // visszaadja a példányt
            return uniqueInstance;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // a konstruktur private, nem lehet new kulcsszóval példányosítani
            Singleton s1 = Singleton.GetInstance();
            Singleton s2 = Singleton.GetInstance();
            // Teszt: ugyanaz a példány-e a kettő?
            if (s1 == s2)
            {
                Console.WriteLine("Ugyanaz! Tehát csak egy példány van.");
            }
            Console.ReadKey();
        }
    }
}
```

#### 4.2.1.2. Szálbiztos megoldás

```
using System;

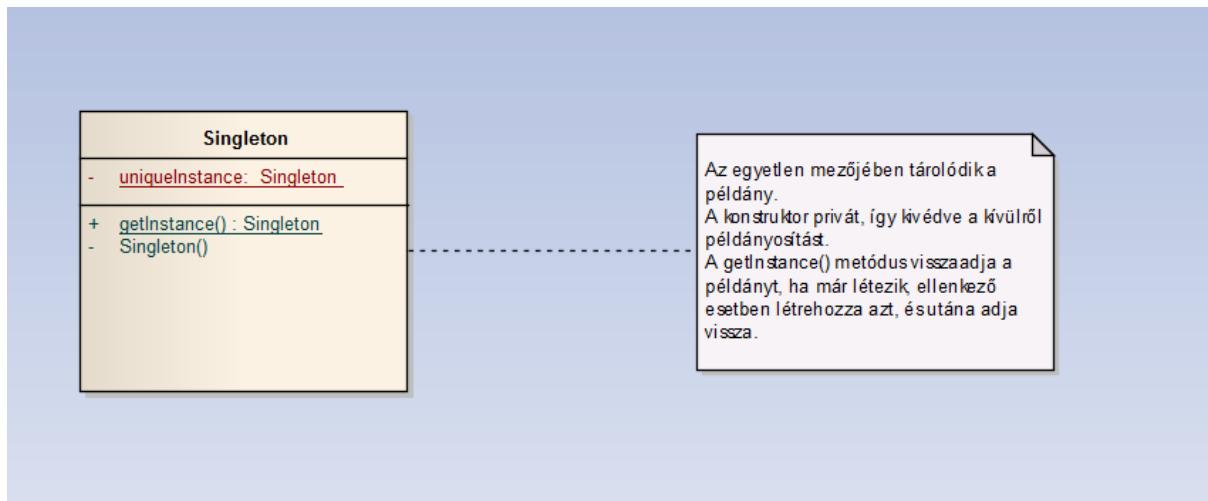
namespace SingletonThreadSafe
{
    public sealed class Singleton
    {
        // A statikus konstruktur akkor fut le, amikor az osztályt példányosítjuk,
```

```

// vagy statikus tagra hivatkozunk ÉS egy Application Domain alatt
// (értsd: adott program futásakor) maximum egyszer futhat le.
private static readonly Singleton instance = new Singleton();
// privát konstruktur külső 'new' példányosítás ellen
private Singleton() { }
// statikus konstruktur
// Azon osztályok, melyek nem rendelkeznek statikus
// konstruktorral beforefieldinit attribútumot
// kapnak az IL kódban. A statikus tagok inicializációja
// a program kezdetén azonnal megtörténik.
// Az olyan osztályok, amelyeknek van statikus konstruktora
// ezt nem kapják meg,
// ezért a statikus tagok akkor példányosulnak,
// amikor először hivatkozunk az osztályra,
// vagyis jelen esetben amikor elkérjük a példányt.
static Singleton() { }
public static Singleton Instance { get{ return instance; } }
}
class Program
{
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;
        if (s1 == s2) { Console.WriteLine("OK"); }
        Console.ReadKey();
    }
}
}

```

#### 4.2.1.3. UML ábra



18. ábra: Példa UML ábra az Egyke tervezési mintá

#### 4.2.1.4. Gyakorlófeladat

Készítsünk forráskódot az alábbi leírás szerint. A megoldáshoz használjuk az egyke tervezési mintát!

#### Feladat

Hozzon létre egy olyan Híváslista osztályt, amelyből csak egy lehet. Gondoljon bele, hogy van-e értelme több Híváslista példányt tárolni a telefonján? Ha igen, miért, ha nem miért?

Mivel nincs értelme több Híváslista példányt tárolni, ezért eleve úgy kell létrehozni az osztályt, hogy abból csak egy példány lehessen és mindenki ezt az egy példányt érhesse csak el, egy úgynévezett globális hozzáférési ponton keresztül.

Gondolja végig, hogy egy ilyen osztály hogyan nézhet ki és válaszoljon az alábbi kérdésekre:

1. Lehet publikus konstruktora ennek az osztálynak?
2. Ha nem lehet, akkor hogyan hozzuk létre belőle példányt, hiszen azt konstrukturhívással lehet csak?
3. Ha kívülről nem hívható, attól még belülről hívható?
4. Ha belülről meghívhatja egy metódus, akkor azt a metódust kellene meghívni?
5. Csakhogy egy példányszintű metódus meghívásához kellene példány, de ahoz pedig példányosítani kellene, amit nem lehet. Hogyan lehet ezt feloldani?
6. Ha ez osztály szintű metódussal megoldható, akkor hogyan kell ennek a metódusnak kinéznie?
7. Hívhatjuk ezt globális hozzáférési pontnak?

Az egyes kérdésekre a válaszok:

1. Nem lehet, mert akkor akárhány példányt lehetne belőle készíteni. Figyelem, ha nem írunk konstruktur, akkor az osztálynak van egy automatikus konstruktora, amely nulla paraméteres és publikus, ezért kell írni egy nem publikus, mondjuk privát konstruktort, ami akár lehet üres is.
2. Attól hogy nincs publikus konstruktör, attól van konstruktora, tehát példányosítani is lehet, igaz ez a konstruktör csak belülről hívható.
3. Igen, belülről minden metódus hívható, akármilyen is a láthatósági szintje.
4. Igen, ez jó megoldás. Arra kell figyelni, hogy a konstruktör által adott új példányt vissza kell adni.
5. Amíg nem tudunk példányt csinálni, addig nem lehet példányszintű metódust se hívni. Ez úgy oldható fel, hogy osztályszintű metódusra bízzuk a példányosítást, hiszen osztályszintű metódus hívásához nem kell példány.
6. Ez már egyszerű, megnézi, hogy létezik-e már a példány, ha nem, akkor létrehozza, egyébként pedig mindenkinél ugyanazt a példányt adja vissza. Lásd az osztály megvalósítását.
7. Igen, az ilyen metódust globális hozzáférési pontnak hívjuk, illetve magát az osztályt Egykének (angolul: Singleton).

A fenti gondolatmenetből jól látható, hogy az egyke tervezési mintát kell használni, hiszen:

- Van egy privte vagy protected konstruktora, nincs public konstruktora.
- Van egy Egyke típusú osztály szintű metódusa, aminek szokásos neve „instance”.
- Van egy publikus Egyke típusú metódusa, amely mindenkor ugyanazt a példányt (az instance nevű mezőt) adja vissza. Ha ez a példány még null, akkor előtte példányosítja. Ezt a metódust hagyományosan „GetInstance()” metódusnak hívjuk.
- Ezen kívül kellene feladat specifikus példány szintű mezők és metódusok. A fenti példa esetén egy Híváslista típusú mező és az ezt visszaadó get metódus kell.

A feladat megoldásához készítsen főprogramot, amely lekéri kétszer is az Egykéből a híváslistát. Adjunk hozzá ez egyikhez egy hívását, listázzuk ki a másikat. Vegye észre, hogy ha egyiket szerkesztjük, a másik

is változik, hiszen a két lista ugyanoda mutat, hiszen csak egy Híváslista példány van, mindenki azt használja.

Vegye észre továbbá, ez a minta általában minden ilyen feladatra használható. Például minden nyomtatóhoz csak egy nyomtatási sor tartozik, tehát ez is egy egyke.

Vegye észre továbbá, hogy az Egyke.GetInstance() hívások new kulcsszóval történő példányosításokat helyettesítének.

#### 4.2.2. Prototípus – Prototype

A prototípus (angolul: prototype) egy létrehozási tervezési minta, amely egy prototípus klónozásával gyárt objektumokat. A klónok a prototípus pontos másolatai, de saját memória címük van, így a klón megváltoztatása nem változtatja meg a prototípust. Ez a tervezési minta mély klónozást (angolul: deep copy) használ.

A klónozás (angolul: clone) az a programozási technika, amikor a klónozandó objektummal teljesen megegyező új objektumot hozunk létre, azaz a két objektum belső állapota ugyanaz lesz. Az új objektum részben vagy teljesen független az a klónozott objektumtól.

A prototípus tervezési minta fő technikája, mint láttuk, a klónozás. A klónozás feladata, hogy az eredeti objektummal megegyező objektumot hozzon létre. Erre az egyszerű értékadás nem alkalmas, mert azok csak az objektum referenciaját másolják, így a két referencia ugyanoda mutat. A klónozásnak két fajtája van:

- sekély klónozás (angolul: shallow copy): A klónozásnak az a fajtája, amikor az eredeti objektum referencia típusú mezőit csak másoljuk (a két referencia ugyanoda fog mutatni), így az új klón csak részben lesz független az eredeti objektumtól.
- mély klónozás (angolul: deep copy): A klónozásnak az a fajtája, amikor az eredeti objektum referencia típusú mezőit is klónozzuk (a két referencia nem ugyanoda mutat), így az új klón teljesen független lesz az eredeti objektumtól. A megváltoztathatatlan (angolul: immutable) mezőket, mint például a string típusúakat, nem érdemes klónozni.

A különbség a sejély és a mély klónozás közt az, hogy sekély esetben az osztály referenciáit ugyanúgy másoljuk, mint az elemi típusait. Mély klónozásnál az osztály referenciái által mutatott objektumokat is klónozzuk. Nézzük ezt meg egy konkrét példán:

```
class Ember
{
    private String név;
    private Ember[] barátok;
    public Ember DeepCopy()
    {
        Ember clone = new Ember();
        clone.név = név;
        clone.barátok = (Ember[])barátok.Clone();
        return clone;
    }
    public Ember ShallowCopy()
```

```

    {
        Ember clone = new Ember();
        clone.név = név;
        clone.barátok = barátok;
        return clone;
    }
    public Ember ShallowCopy2()
    {
        return (Ember)MemberwiseClone();
    }
}

```

A sekély klónozást a C# nyelv a MemberwiseClone() metódussal segíti, ami az Object osztály része, így minden osztály örökli. Ezért tudtunk a sekély klónozásra két verziót adni a fenti példában.

#### 4.2.2.1. Példa

A prototípus mintát egy példán keresztül mutatjuk be: Hurrá, a magyar gépkocsigyártás újra feléledt, legalábbis a példánk kedvéért. Megjött az utasítás a tehergépkocsi gyártására, kis csapatunk összedugja a fejét és úgy dönt, mer nagyban gondolkodni. Amennyiben sikeres lesz a teherautó üzletág, akkor megvehetjük a méltán híres Porsche és Aston Martin márkákat a profitból. Ezért, gondolva a jövőre, első körben egy általános gépkocsi osztályt hoznak létre, mely azokat a tulajdonságokat tartalmazza, amik minden négy vagy több kerekű gépesített járműre jellemzőek. Ebből az osztályból öröklődik a nagy és erős tehergépkocsi, melynek csak pár speciális tulajdonságát kell beállítanunk. Majd ha a zsebünk tele lesz a teherautó export-import bevételeiből, és végre megvettük a fent említett márkákat, könnyű dolgunk lesz az implementáció során, hiszen egy új osztályban beállítjuk a sportkocsi végsebességét, a tankmérget kisebbre vesszük és indulhat a sorozatgyártás a gyáron keresztül, és a határ a csillagos ég vagy a Forma 1. A gyártósor egy prototípushoz. Mindegy, hogy milyen Gépkocsit kap, minden tud gyártani, mert csak klónozza a prototípushoz. A klónozáson túl csak festeni tud. Szóval a gyár buta, de hatékony.

A lenti forráskódban figyeljük meg, hogy sekély klónozást használunk. Ezt kétféleképen is megírtuk. Ha a MemberwiseClone() segítségével oldjuk meg, akkor elegendő az ősbe megírni a Clone() metódust. Egyébként minden alosztályban meg kell írni. Ezt a megoldást a lenti megoldásban megjegyzések formájában látjuk.

#### 4.2.2.2. Forráskód

```

using System;

namespace Prototípus
{
    public abstract class Gépkocsi : ICloneable
    {
        private string tipus;
        public string Tipus
        {
            get { return tipus; }
            set { tipus = value; }
        }
        private int utasokSzama;
    }
}

```

```

public int UtasokSzama
{
    get { return utasokSzama; }
    set { utasokSzama = value; }
}
private double tankMeret;
public double TankMeret
{
    get { return tankMeret; }
    set { tankMeret = value; }
}
private string szin;
public string Szin
{
    get { return szin; }
    set { szin = value; }
}
public Gépkocsi(string tipus, int utasokSzama, double tankMeret)
{
    this.tipus = tipus;
    this.utasokSzama = utasokSzama;
    this.tankMeret = tankMeret;
}
public object Clone() { return this.MemberwiseClone(); }
/*
public virtual object Clone()
{
    Gépkocsi uj = new Gépkocsi(Tipus, UtasokSzama, TankMeret);
    uj.Szin = Szin;
    return uj;
}/*
public override string ToString()
{
    return tipus + " " + utasokSzama + " " + tankMeret + " " + szin;
}
}
public class Versenyautó : Gépkocsi
{
    private int vegsebesseg;
    public int Vegsebesseg
    {
        get { return vegsebesseg; }
        set { vegsebesseg = value; }
    }
    public Versenyautó(string t, int u, double tm, int vegsebesseg) :
        base(t, u, tm) { this.vegsebesseg = vegsebesseg; }
    /*
    public override object Clone()
    {
        Versenyautó uj =
            new Versenyautó(Tipus, UtasokSzama, TankMeret, Vegsebesseg);
        uj.Szin = Szin;
        return uj;
    }
}

```

```

    }*/
    public override string ToString()
    {
        return base.ToString() + " " + vegsebesseg;
    }
}
public class Teherautó : Gépkocsi
{
    private double teherbiras;
    public double Teherbiras
    {
        get { return teherbiras; }
        set { teherbiras = value; }
    }
    public Teherautó(string t, int u, double tm, double teherbiras)
        : base(t, u, tm) { this.teherbiras = teherbiras; }
    /*
    public override object Clone()
    {
        Teherautó uj =
            new Teherautó(Tipus, UtasokSzama, TankMeret, Teherbiras);
        uj.Szin = Szin;
        return uj;
    }*/
    public override string ToString()
    {
        return base.ToString() + " " + Teherbiras;
    }
}
public class Gyar
{
    public Gépkocsi[] sorozatgyartas(Gépkocsi g, string sz, int db)
    {
        Gépkocsi[] temp = new Gépkocsi[db];
        for (int i = 0; i < db; i++)
        {
            temp[i] = (Gépkocsi)g.Clone();
            temp[i].Szin = sz;
        }
        return temp;
    }
}
class Program
{
    static void Main(string[] args)
    {
        //a versenyautó és a teherautó prototípus létrehozása
        Gépkocsi prototipus1 = new Versenyautó("Aston Martin", 4, 180, 220);
        Gépkocsi prototipus2 = new Teherautó("Csepel", 3, 200, 1000);
        Gyar gyartosor = new Gyar();
        // legyárt 10 piros versenyautót
        Gépkocsi[] vk = gyartosor.sorozatgyartas(prototipus1, "Piros", 10);
        foreach (Versenyautó v in vk) { Console.WriteLine(v); }
    }
}

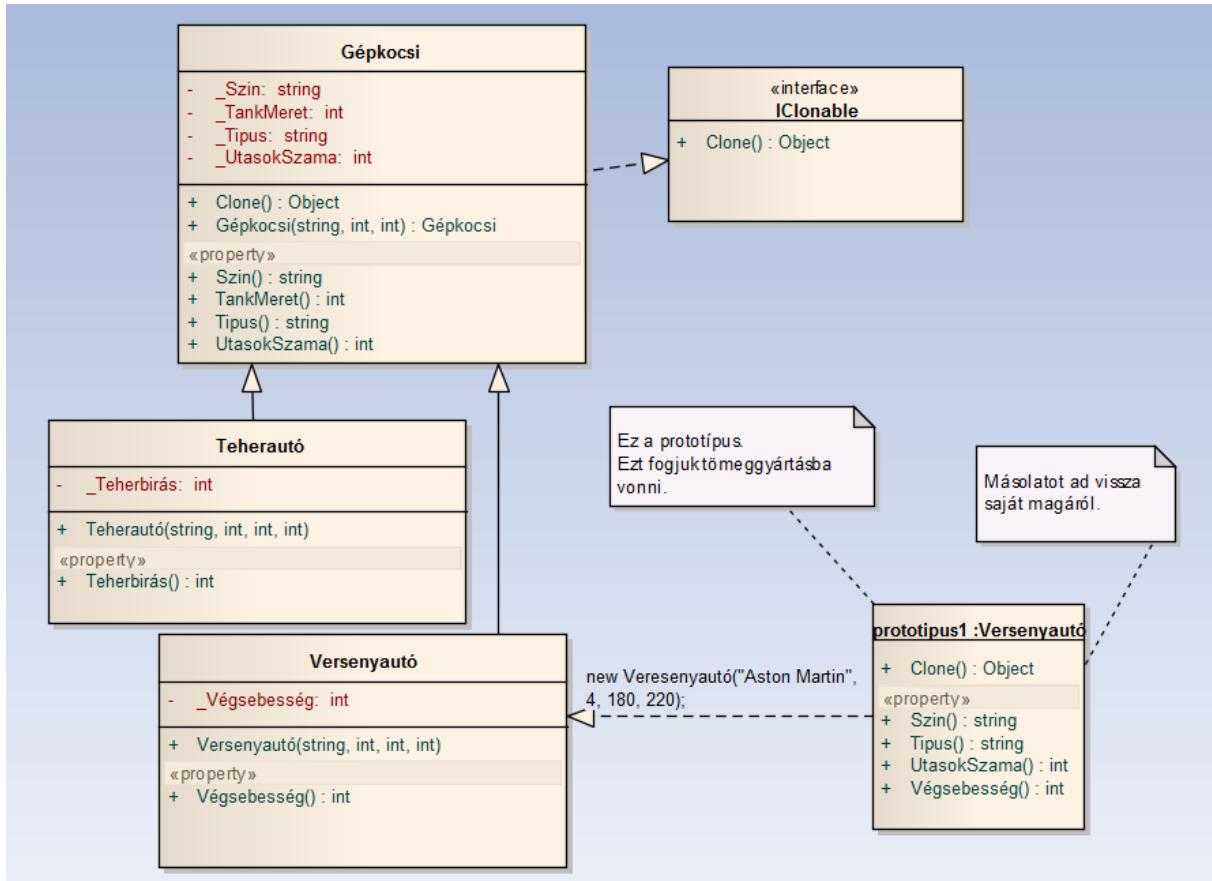
```

```

    // legyárt 20 szürke teherautót
    Gépkosci[] tk = gyartosor.sorozatgyartas(prototipus2, "Szürke", 20);
    foreach (Teherautó t in tk) { Console.WriteLine(t); }
    Console.ReadLine();
}
}
}

```

#### 4.2.2.3. UML ábra



19. ábra: Példa UML ábra a prototípus tervezési mintára

#### 4.2.2.4. Gyakorlófeladat

Készítsünk forráskódot az alábbi leírás szerint. A megoldáshoz használjuk a prototípus tervezési mintát!

#### Feladat

Készítsünk telefon gyárat. A telefongyárnak átadunk egy minta telefont, amit egy az egyben lemásol. A másolat megy tovább a szalagon, ahol egyedi tulajdonságokat kap, mint pl. a szín.

Ennek a feladatnak a megoldásához azt kell végig gondolni, hogy hogyan lehet tökéletesen lemásolni egy bonyolult objektumot. Szerencsére erre van megoldás objektumorientált programozásban, a klónozás, illetve a klónozást alkalmazó prototípus tervezési minta.

Telefon osztály:

- A telefon funkcionálitását megvalósító mezők és metódusok.
- Egyedi tulajdonságok mezői, pl. szín, típus.
- SetSzín(szín): Beállítja a szín mezőt.
- Clone(): mélységi klónozás (angolul: deep copy) segítségével lemásolja az telefont és visszaadja tökéletes másolatát.

TelefonGyár osztály:

- prototípus: a gyártáshoz szükséges prototípus.
- SetPrototípus(prototípus): beállítja a prototípus mezőt.
- Gyárt(szín): lemásolja a prototípust a prototípus.Clone() hívással, majd a másolatnak beállítja a színét és visszaadja a másolatot.

Főprogram: Hozzunk létre egy telefont. Hozzunk létre egy telefongyárat gyár néven. A gyárnak adjuk át a telefont. Gyártsunk a prototípus alapján 10 kék és 5 piros telefont.

Vegye észre, hogy minden gyár.Gyárt(szín) metódushívás kivált egy new kulcsszóval történő példányosítást. Ha a példányosítás menete változik, pl. minden telefonnak egyedi széria-számot kell adnia, akkor ezt elég a Gyárt metódusba lekódolni, míg a másik esetben minden new után bele kellene illeszteni ezt a kódba.

#### 4.2.3. Gyártómetódus – Factory Method

A gyártómetódus (angolul: factory method) egy létrehozási tervezési minta. Ezzel a mintával lehet szépen kiváltani a programunkban lévő rengeteg hasonló new utasítást. A minta leírja, hogyan készítsünk gyártómetódust. Ezt magyarul gyakran készít, angolul a create szóval kezdjük. A gyártómetódus a nevében magadott terméket adja vissza, tehát a készítKutya (angolul: createDog) egy kutyát, a készítMacska (angolul: createCat) egy macskát. Ez azért jobb, mint a new Kutya() vagy a new Macska() konstruktur hívás, mert itt az elkészítés algoritmusát egységebe tudjuk zárni. Ez azért előnyös, mert ha a gyártás folyamata változik, akkor azt csak egy helyen kell módosítani. Általában a gyártás folyamata ritkán változik, inkább az a kérdés mit kell gyártani, azaz ez gyakran változik, ezért ezt az OCP elvnek megfelelően a gyermek osztály dönti el.

Tehát az ősben lévő gyártómetódus leírja a gyártás algoritmusát, a gyermek osztály eldönti, hogy mit kell pontosan gyártani. Ezt úgy érjük el, hogy az algoritmus háromféle lépést tartalmazhat:

- A gyártás közös lépései: Ezek az ősben konkrét metódusok, általában nem virtuálisak, illetve Java nyelven final metódusok.
- A gyártás kötelező változó lépései. Ezek az ősben absztrakt metódusok, amiket a gyermek felülír, hogy eldönthesse, hogy mit kell gyártani. A gyermek osztályok itt hívják meg a termék konstruktőrét.
- A gyártás opcionális lépései: Ezek az ősben hook metódusok, azaz a metódusnak van törzse, de az üres. Ezeket az OCP elv megszegése nélkül lehet felülírni az opcionális lépések kifejtéséhez.

Jó példa a gyártómetódusra az Office csomag alkalmazásaiban lévő Új menüpont. Ez minden alkalmazásban létrehoz egy új dokumentumot és megnyitja. A megnyitás közös, de a létrehozás más

és más. A szövegszerkesztő esetén egy üres szöveges dokumentumot, táblázatkezelő esetén egy üres táblázatot kell létrehozni.

Érdekes megfigyelni, hogy az absztrakt ōs és a gyermek osztályai IoC (angolul: Inversion of Control) viszonyban állnak. Azaz nem a gyermek hívja az ōs metódusait, hanem az ōs a gyermekét. Ezt úgy érjük el, hogy a gyártómetódus absztrakt-, illetve virtuális metódusokat hív. Amikor a gyermek osztály példányán keresztül hívjuk majd a gyártómetódust, akkor a késői kötés miatt ezen metódusok helyett az ōket felülíró gyermekbeli metódusok fognak lefutni.

#### 4.2.3.1. Forráskód

```
using System;

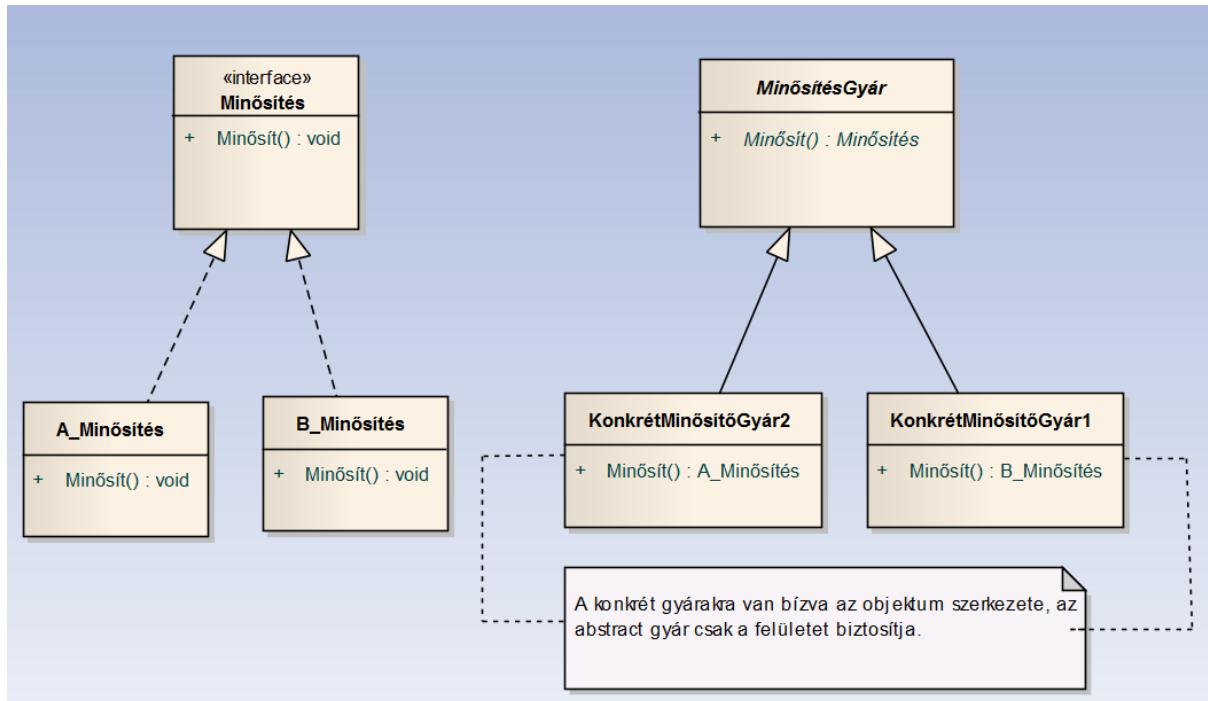
namespace FactoryMethod
{
    abstract class MinositesGyar
    {
        public Minosites CreateMinosites()
        {
            // itt a gyártás előtt lehet ezt-azt csinálni, pl. logolni
            return Minosit();
        }
        public abstract Minosites Minosit();
    }
    class KonkretMinositesGyar1 : MinositesGyar
    {
        public override Minosites Minosit() { return new A_Minosites(); }
    }
    class KonkretMinositesGyar2 : MinositesGyar
    {
        public override Minosites Minosit() { return new B_Minosites(); }
    }
    interface Minosites { void Minosit(); }
    class A_Minosites : Minosites
    {
        public void Minosit() { Console.WriteLine("A-minősítésben részesül!"); }
    }
    class B_Minosites : Minosites
    {
        public void Minosit() { Console.WriteLine("B-minősítésben részesül!"); }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MinositesGyar[] minosito = new MinositesGyar[2];
            minosito[0] = new KonkretMinositesGyar1();
            minosito[1] = new KonkretMinositesGyar2();
            foreach (MinositesGyar m in minosito)
            {
                Minosites minosites = m.CreateMinosites();
                minosites.Minosit();
            }
        }
    }
}
```

```

        Console.ReadLine();
    }
}
}
}

```

#### 4.2.3.2. UML ábra



20. ábra: Példa UML ábra a gyártómetódus tervezési mintára

#### 4.2.3.3. Gyakorlófeladat

Készítsünk forráskódot az alábbi leírás szerint. A megoldáshoz használjuk a gyártómetódus tervezési mintát!

#### Feladat

Ma reggel az egyik programozónk, aki egyébként teljesen normális, azzal állt elő, hogy gazoljuk ki a „new” szócskát a kódjainkból. Először azt gondoltuk, hogy nem itta meg a reggeli kávéját és majd rendbe jön, de azután újragondoltunk a dolgokat. Mit is jelent a „new”? Azt, hogy hozzákötöttük magunkat egy konkrét osztályhoz, amivel nincs is semmi baj, amíg nem gondolunk a: „programozz interfészre” illetve a „nyílt a változásra, de zárt a módosításokra” varázsszavakra. Kapóra jött az új ötlet kipróbálásához a nemrég érkezett megrendelés, melyben egy zsíros kenyér franchise hálózatot kellett kidolgozni. A csomagolásnak országos szinten egyformának kell lennie, de a kenyeret vidékenként más-más vastagságúra vágják, még Pest és Buda között is különbség van, és akkor még nem szoltunk a különböző típusokról, kacsaszír, libazsír stb. Két absztrakt osztályt hoztunk létre, egyet a boltnak, egyet a terméknek. Esetünkben ez a ZsírosDeszka. A boltban kidolgoztuk a csomagol függvényt az egyencsomagolásért, de az elkészítés függvényét absztrakt típusúra vettük. Egyszóval rábíztuk a konkrét boltra, milyen vastagra vágja a kenyeret, mennyi zsírt ken rá. A másik osztályt a termékért hoztuk létre. Mert mi kell a zsíros deszkához? Kenyér, zsír, só, hagyma, de csak a kenyérből legalább nyolc félét tudunk egy ültő helyünkben felsorolni. Így a nyíregyházi bolt tud gyártani nyíregyházi stílusú

zsíros kenyерet. Szóval, ha zsíros kenyeret kell gyártanunk Kecskeméten a kódunk kb. így fog kinézni:  
ZsírosDeszka zsírosKenyér = KecskemétiBolt.készítZsírosDeszka();

#### 4.2.4. Absztrakt gyár – Absztrakt Factory

Az Absztrakt gyár (angolul: Absztrakt Factory) egy létrehozási tervezési minta, amely olyan objektumok gyártására jó, amelyek képesek egymással együttműködni, ennek megfelelően több létrehozásra alkalmas metódust (angolul: create method) tartalmaz.

Ez azt jelenti, hogy akkor érdemes basztrakt gyárat használni, ha egyszerre több dolgot gyárt a gyárunk, és azoknak egymással kompatibilisnek kell lenniük. Azaz, ha gyártunk alvázat és motort, akkor azoknak összeépíthetőnek kell lenniük.

##### 4.2.4.1. Forráskód

```
using System;

namespace AbstractFactory
{
    abstract class Alváz { }
    class OpelAlváz : Alváz { }
    class MerciAlváz : Alváz { }
    abstract class Motor { }
    class OpelMotor : Motor { }
    class MerciMotor : Motor { }
    abstract class AutóGyár
    {
        public abstract Alváz CreateAlváz();
        public abstract Motor CreateMotor();
    }
    class OpelGyár : AutóGyár
    {
        public override Alváz CreateAlváz() { return new OpelAlváz(); }
        public override Alváz CreateMotor() { return new OpelMotor(); }
    }
    class MerciGyár : AutóGyár
    {
        public override Alváz CreateAlváz() { return new MerciAlváz(); }
        public override Alváz CreateMotor() { return new MerciMotor(); }
    }
    class Program
    {
        static void Main(string[] args)
        {
            AutóGyár gyár = new OpelGyár();
            Alváz alváz1 = gyár.CreateAlváz();
            Motor motor1 = gyár.CreateMotor();
        }
    }
}
```

#### 4.2.4.2. Gyakorlófeladat

Készítsünk forráskódot az alábbi leírás szerint. A megoldáshoz használjuk az absztrakt gyár tervezési mintát!

#### Feladat

A telefon részegységekből áll: kijelző, akkumulátor, panel. Egy konkrét telefon csak a hozzá való részegységekből építhető fel. Készítsen X123 és Y987 telefon gyártására alkalmas gyárakat.

Mielőtt nekilát a feladatnak, vegye észre, hogy minden gyárnak gyártania kell minden részegységet, tehát minden gyárnak lesz GyártAkkumulátor() metódusa. Ezért ezek kiemelhetőek egy közös ősbe, az absztrakt gyárba.

Absztrakt alkatrész osztályok:

- AbsztraktKijelző: A kijelző alkatrészek közös őse.
- AbsztraktAkkumulátor: Az akkumulátorok közös őse.
- AbsztraktPanel: A panelek közös őse.

AbsztraktGyár osztály:

- GyártKijelző(): absztrakt metódus, visszatérési típusa AbsztraktKijelző.
- GyártAkkumulátor(): absztrakt metódus, visszatérési típusa AbsztraktAkkumulátor.
- GyártPanel(): absztrakt metódus, visszatérési típusa AbsztraktPanel.

Konkrét alkatrész osztályok:

- X123Kijelző: X123 típusú készülékhez való kijelző, őse az AbsztraktKijelző.
- X123Akkumulátor: X123 típusú készülékhez való akkumulátor, őse az AbsztraktAkkumulátor.
- X123Panel: X123 típusú készülékhez való panel, őse az AbsztraktPanel.
- Y987Kijelző: Y987 típusú készülékhez való kijelző, őse az AbsztraktKijelző.
- Y987Akkumulátor: Y987 típusú készülékhez való akkumulátor, őse az AbsztraktAkkumulátor.
- Y987Panel: Y987 típusú készülékhez való panel, őse az AbsztraktPanel.

X123Gyár osztály:

- GyártKijelző(): X123Kijelző típusú objektumot gyárt.
- GyártAkkumulátor(): X123Akkumulátor típusú objektumot gyárt..
- GyártPanel(): X123Panel típusú objektumot gyárt.

Y987Gyár osztály:

- GyártKijelző(): Y987Kijelző típusú objektumot gyárt.
- GyártAkkumulátor(): Y987Akkumulátor típusú objektumot gyárt..
- GyártPanel(): Y987Panel típusú objektumot gyárt.

Főprogram: Hozzon létre egy X123 és egy Y987 gyárat. Gyártson a segítségükkel különböző alkatrészeket.

Megjegyzés: Az alkatrészek összeszerelésére az Építő (angolul: Builder) tervezési minta a legalkalmasabb, de ez nem téma jelen jegyzetnek.

Vegye észre, hogy minden Gyárt szóval kezdődő metódus kivált egy-egy new kulcsszóval történő objektum létrehozást.

### 4.3. Szerkezeti tervezési minták

A szerkezeti tervezési minták arra adnak módszert, hogy hogyan használjuk az objektum-összetételt új objektum szerkezetek létrehozására. Ebben a fejezetben ezt a technikát gyakran becsomagolásnak (angolul: wrapping) nevezzük. Más megfogalmazásban a szerkezeti minták azt mutatják meg, hogy hogyan használjuk a gyakorlatban az objektum-összetételt, hogy az igényeinknek megfelelő objektumszerkezetek létrejöhessenek futási időben.

Ismétlésképp leírjuk az objektum-összetételnek, vagy más néven a HAS-A kapcsolatnak a fajtait, amelyek a birtoklás módja szerint az aggregáció és a kompozíció, illetve a becsomagolás módja szerint az átlátszó és az átlátszatlan becsomagolás:

- aggregáció: amikor az összetételben szereplő objektum nem kizárolagos tulajdona az őt tartalmazó objektumnak,
- kompozíció: amikor kizárolagos tulajdona,
- átlátszó csomagolás: amikor a tulajdonos ugyanolyan típusú, mint az összetételben szereplő objektum.
- átlátszatlan csomagolás: amikor a tulajdonos nem ugyanolyan típusú, mint az összetételben szereplő objektum.

#### 4.3.1. Illesztő – Adapter

Az illesztő (angolul: adapter) egy szerkezeti tervezési minta, amely átalakítja a becsomagolt objektum felületét a kívánt felületre. Ehhez nem átlátszó becsomagolást használ. Arra szolgál, hogy egy meglévő osztály felületét hozzá igazítsuk saját elvárásainkhoz. Leggyakoribb példa, hogy egy régebben megírt osztályt akarunk újrahasznosítani úgy, hogy beillesztjük egy osztályhierarchiába. Mivel ehhez hozzá kell igazítani az ős által előírt felületeket, ezért illesztőmintát kell használnunk.

A régi osztályt ilyen estben gyakran illesztendőnek (angolul: adaptee) hívjuk. Az illesztő és az illesztendő között általában kompozíció van, azaz az illesztő kizárolagosan birtokolja az illesztendőt. Ezt gyakran úgy is mondjuk, hogy az illesztő becsomagolja az illesztendőt. Ennek megfelelő az illesztőminta másik angol neve: Wrapper. Ugyanakkor ez a becsomagolás átlátszatlan, hiszen az illesztő nem nyújtja az illesztendő felületét.

##### 4.3.1.1. Példa

Az alábbi példában az Ember osztályhierarchiába illesztjük bele a Robot osztályt a Robot2Ember osztály segítségével. Tehet a Robot az illesztendő (angolul: adaptee), a Robot2Ember az illesztő (angolul: adapter). Úgy is mondhatnánk, hogy a robotunkat emberként szeretnénk használni. A főprogramban ehhez az R2D2 nevű robotunkat becsomagoljuk egy Robot2Ember példányba.

Mivel az illesztő átkonvertálja az egyik felületet egy másikká, ezért gyakran Régi2Új nevet adunk az osztálynak. Példánkban Robot2Ember. Itt a 2 az angol „Two” szóra utal, amit ugyanúgy kell kiejteni, mint az angol „To” szót. Ez egy gyakori elnevezési konvenció a konverziót végző metódusokra, osztályokra.

#### 4.3.1.2. Forráskód

```
using System;
abstract class Ember
{
    public abstract string GetNév();
    public abstract int GetIQ();
}
class Robot
{
    string ID;
    int memory; //memória MB-ban megadva
    public Robot(string ID, int memory)
    {
        this.ID = ID;
        this.memory = memory;
    }
    public string GetID() { return ID; }
    public int GetMemory() { return memory; }
}
class Robot2Ember : Ember
{
    Robot robi;
    public Robot2Ember(Robot robi) { this.robi = robi; }
    public override string GetNév()
    {
        return robi.GetID();
    }
    public override int GetIQ()
    {
        return robi.GetMemory() / 1024; // 1GB memória = 1 IQ
    }
}
class Program
{
    static void Main(string[] args)
    {
        Robot R2D2 = new Robot("R2D2", 80000);
        Ember R2D2wrapper = new Robot2Ember(R2D2);
        Console.WriteLine("Neve: {0}", R2D2wrapper.GetNév());
        Console.WriteLine("IQ-ja: {0}", R2D2wrapper.GetIQ());
        Console.ReadLine();
    }
}
```

#### 4.3.2. Díszítő – Decorator

A díszítőminta az átlátszó csomagolás klasszikus példája. Klasszikus példája a karácsonyfa. Attól, hogy a karácsonyfára felteszik egy gömböt, az még karácsonyfa marad, azaz a díszítés átlátszó. Ezt úgy érjük el, hogy az objektum-összetételben szereplő minden osztály ugyanazon őstől származik, azaz ugyanolyan típusúak. Ez azért hasznos, mert a díszítőelemek gyakran változnak, könnyen elképzelhető, hogy új díszt kell felvenni. Ha díszítő egy külön típus lenne, akkor a karácsonyfa-feldolgozó algoritmusok esetleg bonyolultak lehetnek.

A díszítőmintánál egy absztrakt űsből indulunk ki. Ennek kétfajta gyermeke van, alaposztályok, amiket díszíteni lehet és díszítőosztályok. A karácsonya példa esetén az alaposztályok a különböző fenyőfák. A díszítőosztályokat általában egy absztrakt díszítőosztály alá szervezzük, de ez nem kötelező.

A díszítés során az űs minden metódusát implementálni kell, úgy, hogy a becsomagolt példány metódusát meghívjuk, illetve ahol ez szükséges, ott hozzáadjuk a plusz funkcionálitást. Kétféle díszítésről beszélhetünk:

- Amikor a meglévő metódusok felelősséggörét bővítjük. Ilyen a karácsonyfás példa.
- Amikor új metódusokat is hozzáadunk a meglévőkhöz. Ilyen a Java adatfolyam (angolul: stream) kezelése, illetve a lenti kölcsönözhető jármű példája.

Mindkét esetben a példányosítás tipikusan így történik:

```
ŐsOsztály példány = new DíszítőN(...new Díszítő1( new AlapOsztály())...);
```

Mivel a csomagolás átlátszó, ezért akárhányszor becsomagolhatjuk a példányunkat, akár egy díszítővel kétzer is. Ez rendkívül dinamikus, könnyen bővíthető szerkezetet eredményez, amit öröklődéssel csak nagyon sok osztállyal lehetne megvalósítani.

Érdekes megfigyelni a minta UML ábráján, hogy a díszítőosztályból visszafelé mutat egy aggregáció az űs osztályra. Ez az adatbázis-kezelés Alkalmazott – Főnök reláció megoldásához hasonlít, amikor az Alkalmazott tábla önmagával áll egy-több kapcsolatban, ahol a külső kulcs a főnök alkalmazott\_ID értékét tartalmazza.

#### 4.3.2.1. Példa

A díszítőmintát a következő példával mutatjuk be. Képzeljük el, hogy egy versenypályán üzemeltetünk egy autókölcsonzót. Az autókölcsonzóben természetesen több típusú autót, többnyire versenya utóból adunk kölcsönzésre. A lényeg, hogy előfordulhat az, hogy újabb autókkal bővítjük az állományt. Felkészülve erre, először egy alapautó-osztályt hoznak létre, amelyben a bérlehető autók információi szerepelnek, mint gyártó neve, a modell neve, a bérlet időtartama körökben számolva és a bérlet díja. A kölcsönzőben időnként akciókkal kedveskednek az ügyfeleknek, valamint változó, hogy egy bizonyos autó kölcsönözhető-e vagy sem. Ezen extrák hozzáadását a díszítőmintá implementálásával tették lehetővé. Az alapautó-osztályból származik az alapdekorátor-osztály, mely elvégzi a becsomagolást. A konkrét díszítőosztályoknak már csak a funkciók kibővítésével kell foglalkozniuk. Amint egy autót feldíszítünk, mint kölcsönözhető, az már csak a bérletjét várja, aki kiviszi a pályára. Az akciókat is díszítőosztályokkal valósíthatjuk meg. Látható, ha új autókkal bővül a parkunk, vagy újabb akciós ajánlatokat szeretnénk bevezetni, azt könnyedén megtehetjük, új konkrétautó és konkrétdíszítő osztályok hozzáadásával.

#### 4.3.2.2. Forráskód

```
using System;
```

```
namespace DecoratorDesignPattern
{
    public abstract class VehicleBase // alap osztály, adott funkcionálitásokkal
    {
        public abstract string Make { get; }
```

```

        public abstract string Model { get; }
        public abstract double HirePrice { get; }
        public abstract int HireLaps { get; }
    }
    public class Ferrari360 : VehicleBase // egy konkrét autó
    {
        public override string Make { get { return "Ferrari"; } }
        public override string Model { get { return "360"; } }
        public override double HirePrice { get { return 100; } }
        public override int HireLaps { get { return 10; } }
    }
    public abstract class VehicleDecoratorBase : VehicleBase // a dekorátor osztály
    {
        private VehicleBase vehicle; // HAS-A kapcsolat, ezt csomagoljuk be
        public VehicleDecoratorBase(VehicleBase v) { vehicle = v; }
        public override string Make { get { return vehicle.Make; } }
        public override string Model { get { return vehicle.Model; } }
        public override double HirePrice { get { return vehicle.HirePrice; } }
        public override int HireLaps { get { return vehicle.HireLaps; } }
    }
    public class SpecialOffer : VehicleDecoratorBase // konkrét dekorátor osztály
    {
        public SpecialOffer(VehicleBase v) : base(v) { }
        public int Discount { get; set; }
        public int ExtraLaps { get; set; }
        public override double HirePrice
        {
            get
            {
                double price = base.HirePrice;
                int percentage = 100 - Discount;
                return Math.Round((price * percentage) / 100, 2);
            }
        }
        public override int HireLaps { get { return (base.HireLaps + ExtraLaps); } }
    }
    public class Hireable : VehicleDecoratorBase
    {
        public Hireable(VehicleBase v) : base(v) { }
        public void Hire(string name)
        {
            Console.WriteLine("{0} {1} típust kölcsönözött {2} {3}$-ért {4}
körre.\r\n", Make, Model, name, HirePrice, HireLaps);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Ferrari360 car = new Ferrari360();
            Console.WriteLine("Alap Ferrari360:\r\n");
            Console.WriteLine("Alap ár: {0}, alap tesztkörök száma: {1}\r\n\r\n",
car.HirePrice, car.HireLaps);
        }
    }

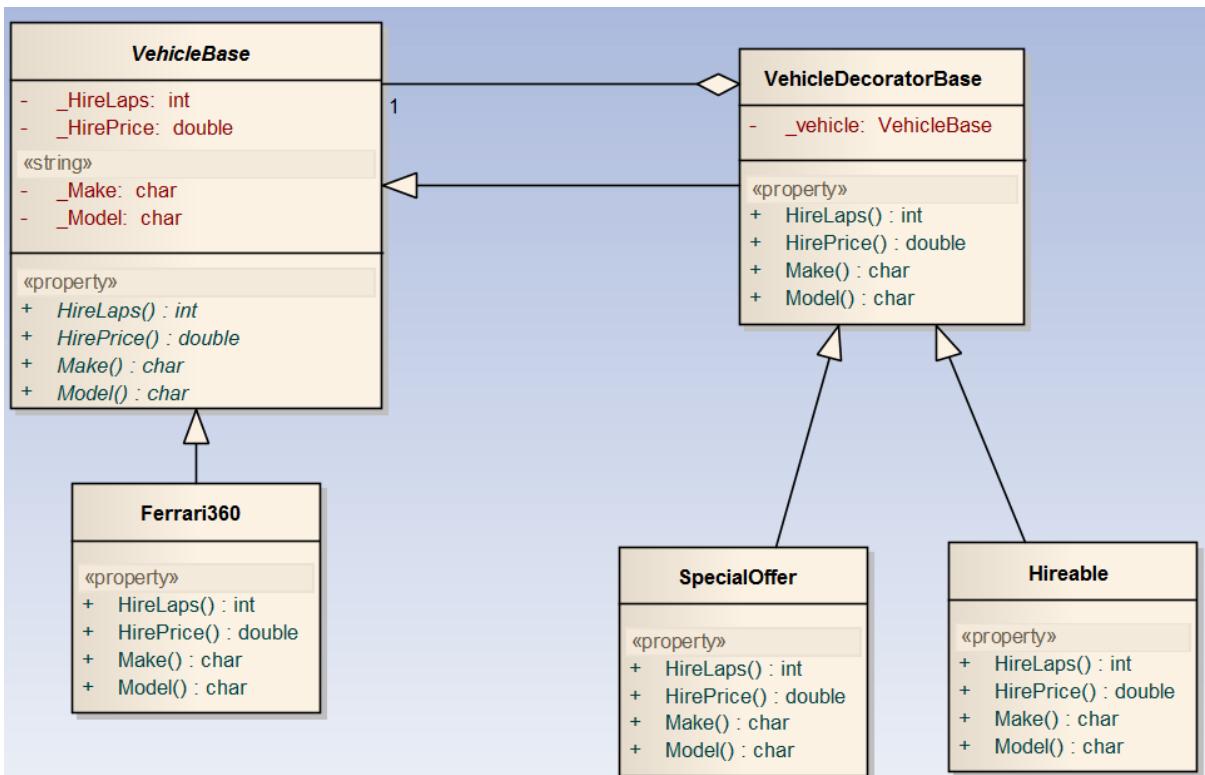
```

```

        SpecialOffer offer = new SpecialOffer(car);
        offer.Discount = 25;
        offer.ExtraLaps = 2;
        Console.WriteLine("Speciális ajánlat:\r\n");
        Console.WriteLine("Különleges ajánlat ára: {0}, {1}$-ért\r\n\r\n",
offer.HirePrice, offer.HireLaps);
        Hireable hire = new Hireable(car);
        hire.Hire("Bill");
        Hireable hire2 = new Hireable(offer);
        hire2.Hire("Jack");
        Console.ReadLine();
    }
}
}

```

#### 4.3.2.3. UML ábra



21. ábra: Példa UML ábra a díszítő tervezési mintára

#### 4.3.2.4. Gyakorló feladat

Az alábbi leírás szerint készítsünk forráskódot. A megoldáshoz használjuk a díszítő tervezési mintát!

##### Feladat 1.

Nekem senki ne mondja, hogy egy programozó élete unalmas, azon kívül, hogy szabadidejében ugyanazokat a dolgokat teheti, mint más rendes ember. Még a munkájában is kaphat érdekes megbízatásokat. A minap kaptunk is: egy bringaboltot kellett csinálnunk. Mi már láttuk is a szemünk előtt a sok csillológó-villogó bringát fel-alá gurulni a szalonban. Okosan, vagy inkább objektumorientáltan-t kéne mondani, készítettünk egy absztrakt osztályt BringaAlap néven, majd

ebből az osztályból származtattuk a Bringa21seb, BringaCsengővel, BringaNői stb. osztályokat, ezek az osztályok tudták a konkrét példány árát. Telt-múlt az idő, és a vásárlók igényeit követve már ilyen osztályneveket használtunk: BringaCsengovel21sebAluvazSarvedoveldecsakElolAkciós. Szép ugye? Éreztük rögtön, hogy ez így nem lesz jó, arról nem is beszélve, hogy osztályaink számának növekedése hasonlított egy demográfiai robbanáshoz. Hosszas tanácskozás után kénytelenek voltunk belátni, hogy majdnem az egész kódot ki kell dobni, bár az első, absztrakt BringaAlap osztályt megtartottuk. Ebben írtunk egy GetLeírás és egy Ár nevű absztrakt függvényt a hozzájuk tartozó mezőkkel. Ebből öröklődött a konkrét Bringa, de még kellettek az alkatrészek, csengő, váltó, sárvédő stb. Így létrehoztunk egy újabb absztrakt osztályt, amit BringaDíszítőnek nevezünk el és ez is a BringaAlap gyermek. A Díszítőből származnak a konkrét elemek, amelyek csak a saját áraikat ismerik, de az Ár függvényük és a konstruktörök úgy van megírva, hogy az őket hívó elem árát is hozzáadják az árhoz. Tulajdonképpen veszünk egy bringát, majd „körbecsomagoljuk” (ezért nevezik ezt a mintát wrapper-nek is) egy sárvédővel, majd ezt egy csengővel és így tovább. Amikor minden igényt kielégítettünk, meghívuk a legutolsó elem Ár függvényét, mely a saját árával meghívja a következő elem ugyanezen függvényét, és a végén visszakapjuk az összeállítás teljes árát.

### Feladat 2.

Készítsen egy kávéitalprogramot, amely szemlélteti a díszítő működését! A feladat szempontjából csak az ár és a kávé összetevői számítsanak (pl. cukor, tejszín, tej, hab, esetleg rum, öntet)! A program vegye figyelembe az árak alakulását is. A feladat az, hogy a kezdetben üres, keserű, fekete kávénkat díszítsük fel.

### Feladat 3.

Készítsen karácsonya programot, amely egy alap fenyőfát díszíthetünk különböző díszekkel. A legfontosabb megfigyelés az az, hogy egy karácsonya karácsonya marad a díszítés után is. Ez azt jelenti, hogy átlátszó becsomagolást kell használnunk.

A feladat megoldásához kell egy absztrakt űs, amely rögzíti a karácsonya szolgáltatásait. Ez jelen esetben csak egy metódus le, a kirajzol, ami az egyszerűség kedvéért csak kiír valamit a képernyőre. Ezen túl kell legalább egy fenyőfa típus, amit díszíteni fogunk és legalább egy dísz típus. Ha több dísz típus is van, akkor érdemes nekik egy közös űst létrehozni.

AbsztraktKarácsonya osztály:

- Absztrakt osztály, a fenyő és a dísz típusoknak is az őse.
- KiRajzol(): Absztrakt metódus. Kirajzolja, vagy csak kiírja, a karácsonyat.

LucFenyő osztály:

- Az AbsztraktKarácsonya osztályból származik, ezt lehet díszíteni.
- KiRajzol(): kiírja, hogy „lucfenyő”.

AbsztraktDísz osztály:

- Az AbsztraktKarácsonya osztályból származik, ezt lehet díszíteni.

- karácsonya: AbsztraktKarácsonya típusú mező. Azt a karácsonyfát tartalmazza, amire a dísz kerül. Ez egy objektumösszetétel, azon belül is átlátszó becsomagolás, mert az AbsztraktDísz és a karácsonya is AbsztraktKarácsonya típusú.
- AbsztraktDísz(karácsonya): Konstruktor, beállítja a karácsonya nevű mezőt. Mivel egy objektum összetétel kívülről kap értéket, ezért ez egy felelősség beinjektálás.
- DíszRajzolás(): Absztrakt metódus, a gyermek osztályok fogják kifejteni. Kirajzolja, vagy csak kiírja, a díszt.
- KiRajzol(): Kódja: DíszRajzol(); karácsonya.KiRajzol(); Ez egy felelősség átadás, és egy a gyermekben kifejtendő metódus hívása. Az első kirajzolja az eddigi karácsonyfát, a második az új díszt.

GömbDísz osztály:

- Az AbsztraktDísz osztályból származik, ez egy feldíszített karácsonya, amit tovább lehet díszíteni.
- GömbDísz(karácsonya): Konstruktor, egyszerűen meghívja az ős konstruktorát a karácsonya paraméterrel.
- DíszRajzol(): Kiírja, hogy „gömbös”.

Főprogram: Hozzon létre egy lucfenyőt. Csomagolja be egy gömb dísz segítségével, így egy „gömbös lucfenyő” jön létre. Készítsen „gömbös gömbös lucfenyő”-t. Csínáljon új dísz típusokat, pl. CsillagDísz és készítsen egy „csillagos gömbös gömbös lucfenyő”-t.

Vegye észre, hogy használtuk átlátszó becsomagolást, felelősség beinjektálást, felelősség átadás.

#### 4.3.3. Helyettes – Proxy

A helyettes (angolul: proxy) tervezési minta egy nagyon egyszerű kompozícióra ad példát, ami ráadásul átlátszó becsomagolás. Egy valamelyen szempontból érdekes (drága, távoli, biztonsági szempontból érzékeny stb.) példányt birtokol a helyettese. Ez az érdekes objektum nem érhető el kívülről, csak a helyettesén keresztül érhetők el a szolgáltatásai. Ugyanakkor a külvilág azt hiszi, hogy az érdekes objektumot közvetlenül éri el, mert a helyettes átlátszó módon csomagolja be az érdekes objektumot. Az átlátszóság miatt a helyettesnek és az érdekes objektumnak közös őse van.

Sokféle helyettes létezik aszerint, hogy milyen szempontból érdekes a helyettesített objektum, pl.:

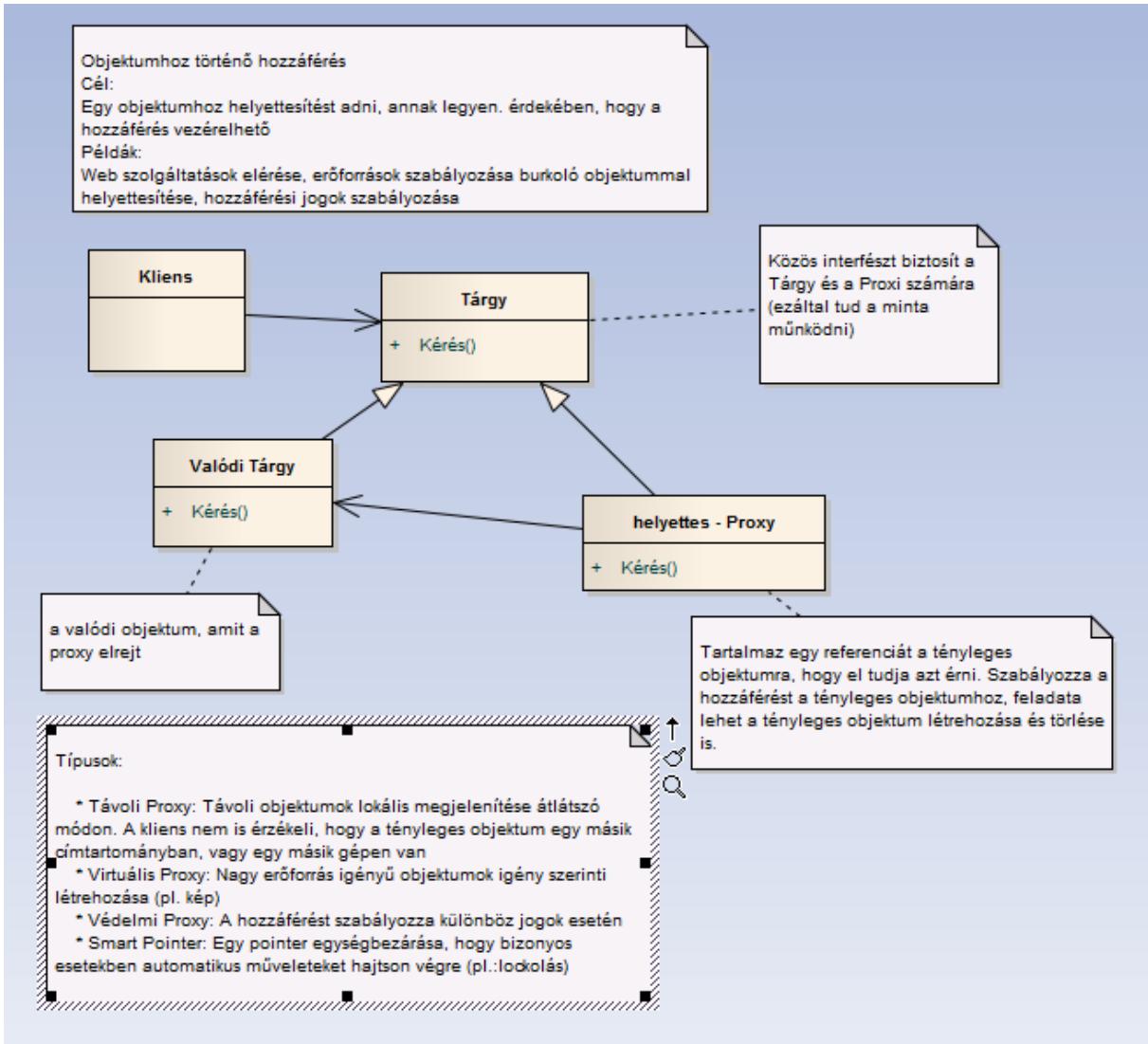
- Virtuális proxy: Nagy erőforrásigényű objektumok (pl. kép) helyettesítése a példányosítás (vagy más drága művelet) elhalasztásával, amíg ez lehetséges. A szövegszerkesztők ezt használják a képek betöltésére. Ha csak gyorsan átlapozom a dokumentumot, akkor a kép nem töltődik be (elhalasztódik a betöltés), csak a helye látszik.
- Távoli proxy: Távoli objektumok lokális megjelenítése átlátszó módon. A kliens nem is érzékeli, hogy a tényleges objektum egy másik gépen van, amíg van hálózati kapcsolat. Ezt alkalmazza a távoli metódushívás (remote method invocation – RMI).
- Védelmi proxy: A hozzáférést szabályozza különböző jogok esetén.
- Okos referencia: Az egyszerű referenciát helyettesíti olyan esetekben, amikor az objektum elérésekor további műveletek szükségesek.

- Gyorsító tár (angolul: cache): Ha van olyan számítás (ide sorolva a letöltéseket is), ami drága, akkor a számítás eredményét érdemes letárolni egy gyorsító tárban, ami szintén egyfajta proxy.

#### 4.3.3.1. Forráskód – Példa 1.

```
using System;
namespace helyettes
{
    class MainApp
    {
        static void Main()
        {
            // Készítünk egy helyetteset és kérünk egy szolgáltatást.
            Proxy proxy = new Proxy();
            proxy.Kérés();
            Console.ReadKey();
        }
    }
    // Közös interfész a Tárgy és a Proxi számára, ezáltal tud a minta működni.
    abstract class Tárgy { public abstract void Kérés(); }
    // valódi munka "tárgy"-amit tenni akarunk
    // a valódi objektum, amit a proxy elrejt
    class ValódiTárgy : Tárgy
    {
        public override void Kérés()
        {
            Console.WriteLine("Meghívom a ValódiTárgy.Kérés-et()");
        }
    }
    // The 'Proxy' osztály
    // Tartalmaz egy referenciát a tényleges objektumra, hogy el tudja azt érni.
    // Szabályozza a hozzáférést a tényleges objektumhoz, feladata lehet a tényleges
    // objektum létrehozása és törlése is.
    class Proxy : Tárgy
    {
        private ValódiTárgy valódiTárgy;
        public override void Kérés()
        {
            if (valódiTárgy == null) { valódiTárgy = new ValódiTárgy(); }
            valódiTárgy.Kérés();
        }
    }
}
```

#### 4.3.3.2. UML ábra



22. ábra: Példa UML ábra a helyettes tervezési mintára

#### 4.3.3.3. Forráskód – Példa 1.

```

using System;
using System.Collections.Generic;
abstract class Faktoriális
{
    public abstract long Fakt(int n); //n faktoriális számol
}
class FaktoriálisCache : Faktoriális
{
    class RekurzívFaktoriális : Faktoriális // beágyazott osztály
    {
        public override long Fakt(int n)
        {
            if (n == 0) return 1;
            return n * Fakt(n - 1);
        }
    }
}
  
```

```

    }
    Dictionary<int, long> t = new Dictionary<int, long>();
    RekurzívFaktoriális f = new RekurzívFaktoriális();
    public override long Fakt(int n)
    {
        if (t.ContainsKey(n)) return t[n];
        long value = f.Fakt(n);
        t.Add(n, value);
        return value;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Faktoriális f = new FaktoriálisCache();
        Console.WriteLine(f.Fakt(20));
        Console.WriteLine(f.Fakt(10));
        Console.WriteLine(f.Fakt(20));
        Console.ReadLine();
    }
}

```

#### 4.3.3.4. Gyakorlófeladat

Az alábbi leírás szerint készítsünk forráskódot. A megoldáshoz használjuk a helyettes tervezési mintát!

#### Feladat 1.

A fenti forráskódot úgy írjuk át, hogy minden részeredmény bekerüljön az átmeneti tárba (angolul: cache). A rekurzió során is figyeljük, hogy a kívánt részeredmény megvan-e az átmeneti tárban. Hasonlítsuk össze az egyes megoldások futásidejét!

#### Feladat 2.

Ki ne ismerné azokat a helyes kis automatákat, amik némi pénz bedobása után jópofa dolgokat adnak egy műanyag golyóban. A hálózat üzemeltetője jelentkezett cégnknél, hogy szertné az interneten keresztül felügyelni a gépek állapotát, mint például tudni, hogy mennyi golyó van még benne. A megvalósítási megbeszélésen egyik kollégánk megemlíttette, hogy Ó volt egy csapatépítő tréningen, ahol az esti táboriúznál egy nagyszakállú, bölcs és kellőképpen öreg programozó mesélt nekik a proxyról és hogy az pont valami illyesmirre való. Szakkönyvek, internet és valóban, az öregnek igaza volt. Innen már könnyű út vezetett a megvalósításig. Természetesen az absztrakt osztályok kidolgozásával kezdtük, először is a közös felületet kellett megírni, amin a helyettes és a mi kis automatánk megtalálja a közös nyelvet. Tehát ebbe az osztályba került a MennyiGolyo() és a MennyiPenz() absztrakt függvény. A szakirodalomból azt is megtudtuk, hogy a visszatérési értékeknek serializable-nek kell lenniük a hálózati forgalom miatt. Ezután a Proxy-t implementáltuk, feladata, hogy a kliens kérését (a főnök utasítását) eljutassa az automatának. Ami ténylegesen átmegy, az a meghívott függvény neve és az esetleges argumentumai. Az igazi kemény munkát ezután az automata (ValódiTárgy) végzi, hiszen csak ő tudja hány golyó rejti még meglepetést az arra sétálóknak. Meghívja a MennyiGolyo() függvényt, majd a kapott eredmény visszajutatja a Proxy-nak, mely büszkén mutatja azt fel a kliensnek. Persze ebben

az esetben nem szabad megfeledkezni a kivételkezeléséről sem, mert amit a hálózaton keresztül elküldünk, az nem biztos, hogy oda is ér.

#### 4.4. Viselkedési tervezési minták

A viselkedési tervezési minták arra adnak módszert, hogy hogyan írunk olyan programot, amit nagyon könnyű újfajta viselkedéssel bővíteni, ha előre látható, hogy milyen fajta viselkedésekre lehet számítani a jövőben.

Ehhez meg kell érteni a szétválasztás elvét (angolul: separation of concerns), amely kimondja, ha valamit szét lehet választani, akkor azt érdemes szétválasztani. Különösen igaz ez az úgynévezett változékony metódusokra. A változékony metódusokat mindig érdemes kiemelni az őket tartalmazó osztályból.

A kiemelt metódusok meghívására sok módszer van, ezek közül bemutatjuk a felelősség átadást, a kontrol megfordítását (angolul: Inversion of Control, vagy röviden IoC) és a műsorszórást.

A felelősség átadás vagy delegálás alatt azt értjük, hogy egy objektum valamely metódusa meghívja a birtokolt objektum egy metódusát, hogy az helyette oldja meg a feladatot részben vagy egészben.

A kontrol megfordítása (angolul: Inversion of Control, vagy röviden IoC) az a programozási módszer, amikor nem a kosztum kód hívja az előre megírt általános kódot, pl. egy könyvtár (angolul: library) függvényt, hanem az általános kód hívja a kosztum kódot. Egyik példája, amikor nem a gyermekosztály hívja az ősosztályt, hanem az ős a gyermeket.

A műsorszórás (angolul: broadcast) egy olyan programozási módszer, amikor egy metódus sok más, előre nem ismert, metódust hív meg egy lista alapján.

##### 4.4.1. Állapot – State

Az állapot (angolul: state) viselkedési tervezési mintát akkor használjuk, ha több összefüggő változékony metódust akarunk kiemelni és azokat delegációval meghívni.

Lehetővé teszi egy objektum viselkedésének megváltozását, amikor megváltozik az állapota.

Egy jól ismert példa a TCPConnection osztály, amely egy hálózati kapcsolatot reprezentál. Három állapota lehet: Listening, Established, Closed. A kéréseket az állapotától függően kezeli.

Használjuk, ha

- az objektum viselkedése függ az állapotától, és a viselkedését az aktuális állapotnak megfelelően futás közben meg kell változtatnia, illetve
- a műveleteknek nagy feltételes ágai vannak, melyek az objektum állapotától függnek.

Előnyök:

- Egységbe zárja az állapotfüggő viselkedést, így könnyű új állapotok bevezetése.
- Áttekinthetőbb kód (nincs nagy switch-case szerkezet).
- Az állapot objektumokat meg lehet osztani.

Hátrányok: Nő az osztályok száma, ezért csak indokolt esetben használjuk.

#### 4.4.1.1. Példa

Az állapot tervezési mintát a következő példán keresztül mutatjuk be: Feladatunk, hogy elkészítsünk egy rendkívül egyszerű audio lejátszót. A lejátszónknak a következőképpen kell működni: ha a lejátszó készenléti állapotban van, akkor a lejátszás gomb hatástan, az audioforrás gombbal pedig megkezdődik az mp3 fájl lejátszása. Mp3 lejátszás közben a lejátszás gomb leállítja a lejátszást, az audioforrás gomb pedig rádióhallgatást tesz lehetővé. Ha az mp3 lejátszás szünetel, akkor a lejátszás gomb hatására folytatódik a lejátszás, az audioforrás gomb pedig ebben az esetben is rádióhallgatást tesz lehetővé. Rádióhallgatás közben a lejátszás gomb adót vált, az audioforrás gomb pedig készenléti üzemmódot eredményez. A leírt összetett működés eléréséhez az állapotgépet valósítjuk meg. Létrehozzuk az audioléjátszó osztályunkat, amelynek van egy belső állapota, valamint egy lejátszás és egy audioforrás metódusa. Létrehozunk egy alapállapot osztályt is, melyből a később szükséges állapotaink származni fognak, és amelyek később a lejátszónk állapotai lehetnek. Az, hogy a lejátszónk az egyes állapotokban hogyan reagál a lejátszás és audio forrás lenyomására, az egyes állapotuktól függ, ezért ezek az egyes állapotokban vannak definiálva, csakúgy, mint az állapotátmenetek is. Módszerünk előnye, hogy könnyedén bővíthetjük a lejátszónkat újabb állapotokkal, és ezáltal újabb funkciókkal bővülhet.

#### 4.4.1.2. Forráskód

```
using System;

namespace Állapot
{
    /// <summary>
    /// Állapot viselkedési tervezési minta
    /// média lejátszó
    /// két gomb
    /// 4 állapot
    /// a két gomb viselkedése más és más lesz a 4 belső állapottól függően
    /// lesz egy: Állapot
    /// Play gomb
    /// Audió forrás gomb
    /// Állapotváltozások:
    /// Állapotok: készenlét, mp3 lejátszás, mp3 megállítás, rádió hallgatás
    /// Lejátszás: stop-paused, start-play, next station
    /// Audió forr: mp3 play, rádió play, rádió play, készenlét
    /// </summary>
    public class AudioPlayer
    {
        private AudioPlayerState state; // ebben tároljuk a belső állapotot
        public AudioPlayer(AudioPlayerState state) { this.state = state; }
        public AudioPlayerState SetState
        {
            get { return state; }
            set { state = value; }
        }
        public void PressPlay() { state.PressPlay(this); }
        public void Press AudioSource() { state.Press AudioSource(this); }
    }
    public abstract class AudioPlayerState // állapot reprezentálása
    {
```

```

    // a két gomblenyomása
    public abstract void PressPlay(AudioPlayer player);
    public abstract void Press AudioSource(AudioPlayer player);
}

public class StandbyState : AudioPlayerState // készenléti állapot
{
    public StandbyState() { Console.WriteLine("StandBy"); }
    public override void PressPlay(AudioPlayer player)
    {
        Console.WriteLine("Play pressed: no effect");
    }
    public override void Press AudioSource(AudioPlayer player)
    {
        player.SetState = new MP3PlayingState();
    }
}
public class MP3PlayingState : AudioPlayerState // mp3 hallgatás állapot
{
    public MP3PlayingState() { Console.WriteLine("Playing MP3"); }
    public override void PressPlay(AudioPlayer player)
    {
        player.SetState = new MP3PausedState();
    }
    public override void Press AudioSource(AudioPlayer player)
    {
        player.SetState = new RadioState();
    }
}
public class MP3PausedState : AudioPlayerState // a megállított mp3 állapot
{
    public MP3PausedState() { Console.WriteLine("Paused MP3"); }
    public override void PressPlay(AudioPlayer player)
    {
        player.SetState = new MP3PlayingState();
    }
    public override void Press AudioSource(AudioPlayer player)
    {
        player.SetState = new RadioState();
    }
}
public class RadioState : AudioPlayerState // a rádió állapot
{
    public RadioState() { Console.WriteLine("Playing Radio"); }
    public override void PressPlay(AudioPlayer player)
    {
        Console.WriteLine("Switch to next Station");
    }
    public override void Press AudioSource(AudioPlayer player)
    {
        player.SetState = new StandbyState();
    }
}
class Program

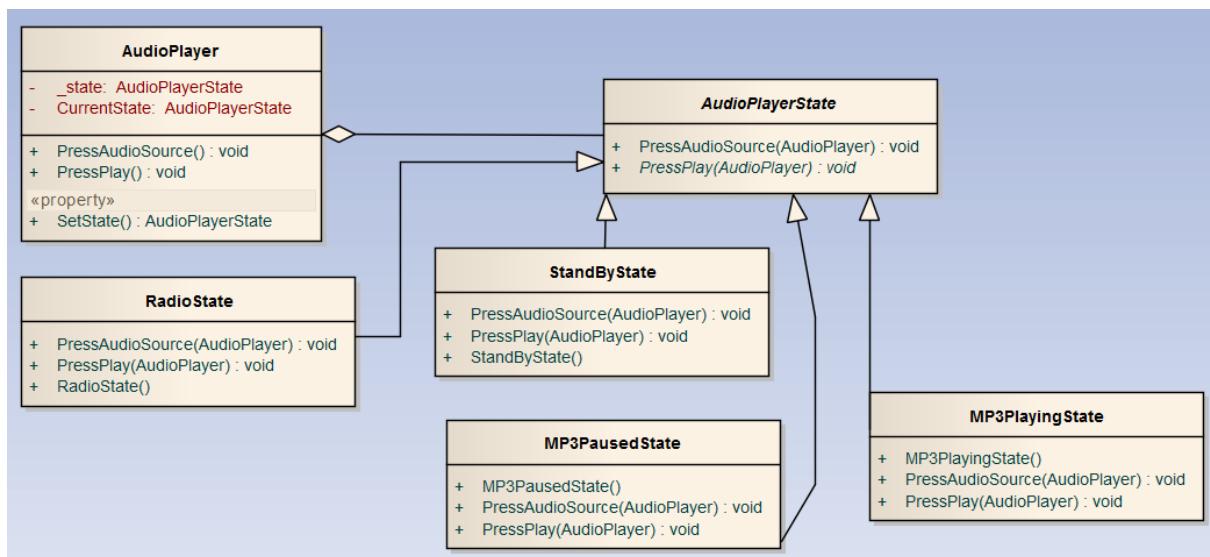
```

```

{
    static void Main(string[] args)
    {
        AudioPlayer player = new AudioPlayer(new StandbyState());
        player.PressPlay();
        player.Press AudioSource();
        player.PressPlay();
        player.PressPlay();
        player.Press AudioSource();
        player.PressPlay();
        player.Press AudioSource();
        Console.ReadLine();
    }
}

```

#### 4.4.1.3. UML ábra



23. ábra: Példa UML ábra az állapot tervezési mintára

#### 4.4.1.4. Gyakorlófeladat

Az alábbi leírás szerint készítsünk forráskódot. A megoldáshoz használjuk az állapottervezési mintát!

#### Feladat

Egy napon az egyik munkatársunk kiment a kávéautomatához egy frissítő italért, pár perc múlva vörös fejjel és kezében az automata programjával jött vissza. Kérdezünkre elmondta, hogy nem először jár úgy, hogy a gép elnyeli az aprót, de kávét nem ad, úgyhogy gondolta, a programmal lesz a baj. Elkezdtük tanulmányozni a szoftvert, mely tele volt csúnya és néhol egymásba ágyazott IF feltételekkel (ezek váltották a belső állapotot, ha bedobtuk a pénzt vagy elfogyott a kávépor stb.). Rögtön gondoltuk, hogy erre van jobb módszer. Először hívtunk egy grafikust, aki lerajzolt minden, majd egy állapotdiagramban a gép lehetséges belső állapotait (pl. nincs apró, apró bedobva stb.). Ebből rögtön láttuk, hogy egyes állapotokban a gépnek meg kell, hogy változzon a viselkedése (ha nincs apró és megnyomom a kávégombot, nem adhat kávét, míg ha van apró, akkor illik legalább valami sötét lötöt

adni). Így már tudtunk csinálni egy interfészt az állapotoknak. Ebből az osztályból dolgoztuk ki a konkrét állapotokat külön osztályokba. Alapesetben a gép a „nincs apró” állapotban (osztályban) van, de ha dobunk be aprót, akkor lecseréli az állapotát (osztályát) „apró bedobva” típusúra. Mióta megírtuk a programot, nekünk már apró sem kell a kávéhoz.

#### 4.4.2. Megfigyelő – Observer

A megfigyelő (angolul: observer) egy viselkedési tervezési minta, amely egy esemény által kiváltott változékony metódust emel ki egy egy-sok kapcsolat sok oldalára, és amely műsorszórással hívja meg a kiemelt metódusokat. A műsorszórás (angolul: broadcast) egy olyan programozási módszer, amikor egy metódus sok más, előre nem ismert, metódust hív meg egy lista alapján. Ez a tervezési minta a Hollywood Principle tervezési alapelvet valósítja meg.

A megfigyelőnek két fajtája van a húzó (angolul: pull) és a toló (angolul: push), amelyek abban különböznek, hogyan adjuk át az eseményt a megfigyelőknek. A húzó megfigyelő egy referenciát ad át a megfigyelőknek, amin keresztül lehúzhatják az eseményt. A toló megfigyelő magát az eseményt adja át a megfigyelőknek paraméterként.

A megfigyelő tervezési minta lehetővé teszi, hogy egy objektum megváltozása esetén értesíteni tudjon tetszőleges más objektumokat anélkül, hogy bármit is tudna róluk. Részei:

- Alany: Tárolja a beregisztrált megfigyelőket, interfészt kínál a megfigyelők be- és kiregisztrálására valamint értesítésére.
- Megfigyelő: Interfészt definiál azon objektumok számára, amelyek értesülni szeretnének az alanyban bekövetkezett változásról. Erre a frissít (Update) metódus szolgál.

Kétfajta megfigyelő megvalósítást ismerünk:

- „Pull-os” vagy húzó megfigyelő: Ebben az esetben a megfigyelő lehúzza a változásokat az alanytól.
- „Push-os” vagy toló megfigyelő: Ebben az esetben az alany odanyomja a változásokat a megfigyelőnek.

A kettő között ott van a különbség, hogy a Frissít metódus milyen paramétert kap. Ha az alany átadja önmagát (egy Frissít(this) hívás segítségével) a megfigyelőnek, akkor ezen a referencián keresztül a megfigyelő képes lekérdezni a változásokat. Azaz ez a „pull-os” megoldás.

Ha a Frissít metódusnak az alany azokat a mezőit adja át, amik megváltoztak és amiket a megfigyelő figyel, akkor „push-os” megoldásról beszélünk. A következő példában épp egy ilyen megvalósítást láthatunk.

##### 4.4.2.1. Forráskód

```
using System;
using System.Collections.Generic;

namespace Megfigyelő
{
    public interface ISubject
    {
```

```

// observer regisztrálásra
void RegisterObserver(IObserver o);
// observer törlésre
void RemoveObserver(IObserver o);
// meghívódik, hogy értesítse az megfigyelőket
// amikor a Subject állapota megváltozik
void NotifyObservers();
}

public interface IObserver
{
    // értékéket amiket megkapnak az observerek a Subjecttől, push-os megoldás
    void Update(float temp, float humidity, float pressure);
}

public interface IDisplayElement
{
    // megjelenítés
    void Display();
}

// implementáljuk a Subject interfészt
public class WeatherData : ISubject
{
    // hozzáadunk egy listát amiben observereket tárolunk
    private List<IObserver> observers;
    private float temperature;
    private float humidity;
    private float pressure;
    public WeatherData()
    {
        // létrehozzuk az observereket tároló listát
        observers = new List<IObserver>();
    }
    public void RegisterObserver(IObserver o)
    {
        // amikor egy observer regisztrál, egyszerűen hozzáadjuk a listához
        observers.Add(o);
    }
    public void RemoveObserver(IObserver o)
    {
        // amikor egy observer kéri a törlését, egyszerűen töröljük a listából
        int i = observers.IndexOf(o);
        if (i >= 0)
        {
            observers.Remove(o);
        }
    }
    // itt szólunk az observereknek az állapotról
    // mivel minden observer, van Update() metódusuk, így tudjuk őket értesíteni
    public void NotifyObservers()
    {
        for (int i = 0; i < observers.Count; i++)
        {
            IObserver observer = (IObserver)observers.ElementAt(i);
            observer.Update(temperature, humidity, pressure); // ez push-os
        }
    }
}

```

```

        // observer.Update(this); // ez pull-os
    }
}
// amikor a Weather Station-tól megkapjuk a frissített értékeket,
//értesítjük az observereket
public void MeasurementsChanged()
{
    NotifyObservers();
}
// értékek beállítása hogy tesztelhessük a megjelenítést
public void SetMeasurements(float temperature, float humidity, float pressure)
{
    this.temperature = temperature;
    this.humidity = humidity;
    this.pressure = pressure;
    MeasurementsChanged();
}
// egyéb metódusok
}
// a display implementálja az Observevert,
//így fogadhat változásokat a WeatherData objektumtól
// továbbá implementálja a DisplayElement-et, mivel
// minden display element-nek implementálnia kell ezt az interfészt
public class CurrentConditionsDisplay : IObserver, IDisplayElement
{
    private float temperature;
    private float humidity;
    private ISubject weatherData;
    // a konstruktor megkapja a weatherData objektumot
    // (a Subject) és arra használjuk, hogy
    // a display-t observerként regisztráljuk
    public CurrentConditionsDisplay(ISubject weatherData)
    {
        this.weatherData = weatherData;
        weatherData.RegisterObserver(this);
    }
    // amikor az Update() meghívódik, mentjük a temperature-t és a humidity-t
    // majd meghívjuk a Display()-t
    public void Update(float temperature, float humidity, float pressure)
    {
        this.temperature = temperature;
        this.humidity = humidity;
        Display();
    }
    // Megjelenítjük a legújabb eredményeket
    public void Display()
    {
        Console.WriteLine("Current conditions: " + temperature + "F degrees and "
+ humidity + "% humidity");
    }
}
public class WeatherStation
{

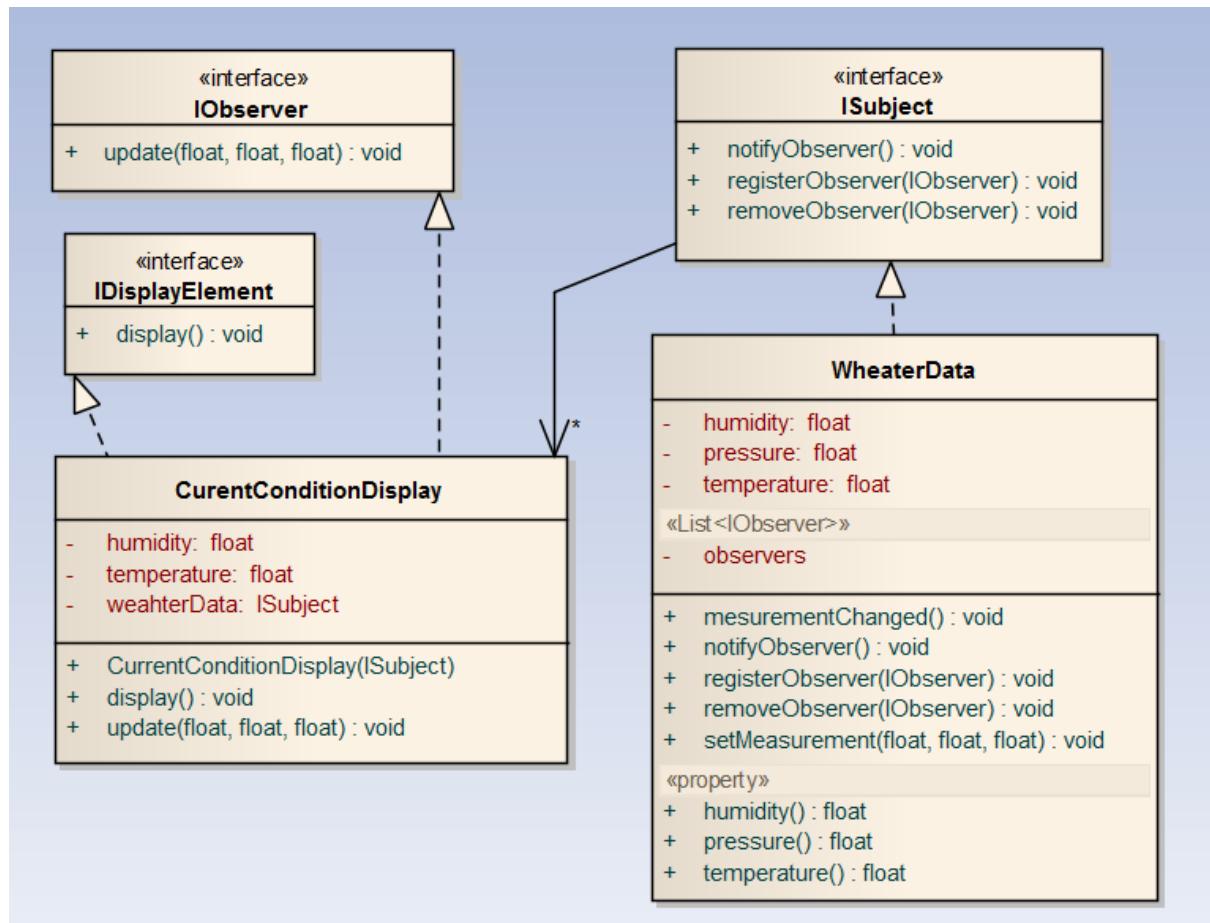
```

```

        static void Main(string[] args)
    {
        // létrehozzuk a weatherData objektumot
        WeatherData weatherData = new WeatherData();
        // létrehozzuk a displayt és odaajuk neki a weatherData-t
        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        // új időjárási mérésértékek szimulálása
        weatherData.SetMeasurements(80, 65, 30.4f);
        weatherData.SetMeasurements(82, 70, 29.2f);
        weatherData.SetMeasurements(78, 90, 29.2f);
        Console.ReadKey();
    }
}

```

#### 4.4.2.2. UML ábra



24. ábra: Példa UML ábra a megfigyelő tervezési mintára

#### 4.4.2.3. Gyakorlófeladat

Az alábbi leírás szerint készítünk forráskódot. A megoldáshoz használjuk a megfigyelő tervezési mintát!

#### Feladat 1.

A fenti példakódot alakítsuk át „pull-os” megfigyelővé.

### **Feladat 2.**

Cégünk azt a megtisztelő feladatot kapta, hogy kalózhajót kellett programozni, ami több heti kódírás után egész szépen úszott a vízen, egy picike probléma volt csak vele. Mégpedig, hogy a főárboc tetején lévő kosárban ülő őrszemnek nem volt rádiója, hiszen még nem találták fel, ezért minden alkalommal, ha valaki alatta ment el, az felkiabált, hogy látja-e már a gazdag zsákmányt vagy az ellent. Szegény emberünknek úgy kiszáradt a torka, hogy egy hordó rumot kellett délig meginnia. A probléma orvoslására és a rumkészlet megmentésére azt találtuk ki, hogy aki szeretne értesülni a hírekrol (Observer), az köt egy kötelet a csuklójához, majd a másik végét feldobja (RegisterObserver()) az őrszemnek (Subject). Amikor az őrszem említésre méltót lát, akkor megrángatja a kötelek végét (NotifyObservers()), és az összegyűlt „megfigyelőknek” lekiabálja a hírt (Update()). Aki nem kíváncsi tovább az eseményekre, az egész egyszerűen lehúzza a kötelének a végét a kosárból (RemoveObserver()).

### **Feladat 3.**

Készítsen példát a Megfigyelő (angolul: Observer) tervezési mintára. A konkrét példa legyen a klasszikus Rajongók – Lady GaGa példa.

Az a feladatunk, hogy tegyük lehetővé a sok rajongónak, hogy kövessék Lady GaGa posztjait. Amikor Lady GaGa posztol valamit, akkor arról minden rajongója értesül. A rajongok fel-, illetve lejelentkezhetnek Lady GaGa híroldaláról.

Gondoljuk végig, hogy mi kell ennek a feladatnak a megoldásához. Vegyük észre, hogy itt egy sok kapcsolat van a híresség híroldala és a rajongók között, és egy esemény, a posztolás, hatására kell értesíteni mindenkit a sok oldalon. Ez egy klasszikus Megfigyelő tervezési mintának megfelelő feladat.

A híroldalon van egy lista, a rajongók listája, erre lehet fel-, illetve leiratkozni. Kell tovább egy Értesít() metódus, ami az utolsó posztot küldi minden a listán lévő rajongónak.

Mivel a posztot küldjük, ezért ez a megoldás egy úgynévezett toló (fél angolul: „push”-os) megoldás lesz. Ha csak refenciát küldenénk, ahonnan a poszt lehúzható, akkor húzó (fél angolul: „pull”-os) megoldásunk lenne.

Híroldal osztály:

- híresség: A híroldal tulajdonosának neve.
- rajongók: A rajongók listája.
- poszt: Az utolsó poszt szövege.
- Híroldal(híresség): Konstruktur, beállítja a híresség nevét. A rajongok listáját létrehozza üresen.
- Feliratkozás(rajongó): A rajongót felveszi a rajongók listájára.
- Leiratkozás(rajongó): A rajongót leveszi a rajongók listájáról.
- Értesít(): minden rajongónak meghívja a Frissít(poszt) metódusát az utolsó poszttal.
- Posztol(poszt): Beállítja a poszt mezőt és meghívja az Értesít() metódust.

Rajongó osztály:

- Frissít(poszt): Megkapja az utolsó posztot. Legegyszerűbb, ha csak kiírjuk a képernyőre a poszt tartalmát.
- Ezen túl lehet más mezője, metódusa.

Főprogram: Hozzon létre két rajongót. Hozza létre Lady GaGa híroldalát. Mindkét rajongót írassa fel a híroldalra. Lady GaGa posztolja, hol lesz a következő koncertje. Gondoljuk végig, hogy mit fog kiírni a program és miért.

Írja át a programot az IObserver és az IObservable interfészek használatával!

Tanulmányozza figyelmesen az értesít kódját: forach(rajongó raj in rajongók) raj.Értesít(poszt); Vegye észre, hogy ez az ügynevezett műsorszórás, hiszen egy eseményről nem csak egy objektum, hanem minden érdekelt objektum értesül.

#### 4.4.3. Sablonmetódus – Template Method

A sablonmetódus (angolul: template method) egy viselkedési tervezési minta, amely egy vagy több változékony metódust emel ki egy gyermekosztállyba, és amely IoC segítségével hívja meg a kiemelt metódusukat.

A sablonmetódus egy olyan tervezési minta, amit akkor használunk, ha van egy általános receptünk, ami alapján több hasonló doleg is gyártható. Klasszikus példa a teafőzés és kávédőzés, amit részletesen is ismertetünk. A sablonmetódus tervezési mintát gyakran hasonlítják össze a stratégia tervezési mintával az alábbi mondattal:

- Stratégia: Ugyanazt csináljuk, de másképp;
- Sablonmetódus: Ugyanúgy csináljuk, de másról.

A receptben háromféle lépés lehet:

- kötelező és közös: bármit készítünk a recepttel, ez a lépés minden ugyanaz,
- kötelező, de nem közös: bármit készítünk a recepttel, ez a lépés szükséges, de minden esetben másról kell konkrétan csinálni,
- opcionális: ez a lépés nem minden esetben szükséges.

Ezeket programozástechnikailag így valósíthatjuk meg:

- A kötelező és közös lépések olyan metódusok, amelyek már az ősből konkrétek és azokat általában nem is szabad felülírni. Ilyen a forró ital főzésénél a vízforraló metódus.
- A kötelező, de nem közös lépések az ősből absztrakt metódusok, amit a gyermek osztályok fejtenek ki. Ilyen a forró ital főzésénél az édesítés.
- Az opcionális lépések az ősből hook metódusok, azaz van törzsük, de az üres. Ezek a metódusok virtuálisak, hogy aki akarja, az felülírhassa őket.

Mivel a gyermek osztálynak implementálnia kell minden absztrakt metódust, ezért az ilyenek kötelezők. Igaz, hogy akár az implementáció üres is lehet. Mivel a hook metódusoknak van

implementációjuk, de üres az üres, ezért nem muszáj őket felülírni, de lehet az OCP elv megszegése nélkül. Ezért ezek az opcionális lépések. A hook metódusokat C# nyelven virtuálisnak kell deklarálni.

Maga a recept a sablonmetódus. Gyakran csak ez az egy metódus publikus, minden más metódus, azaz a recept lépései, privát vagy védett metódusok (a szerint, hogy a gyermek felülírhatja-e vagy sem). Erre azért van szükség, hogy az egyes lépéseket ne lehessen össze-vissza sorrendben hívni, csak a recept által rögzített sorrendben.

Elméletben a sablonmetódus egy algoritmus, amelyben a lépések nem változnak, de a lépések tartalma igen. Ha esetleg mégis bejön egy új lépés, azt érdemes hook metódusnak felvenni.

Érdekes megfigyelni, hogy az absztrakt ős és a gyermek osztályai IoC (angolul: Inversion of Control) viszonyban állnak hasonlóan, mint a gyártómetódus esetén. Ugyanúgy itt is: nem a gyermek hívja az ős metódusait, hanem az ős a gyermekét. Ezt úgy érjük el, hogy a sablonmetódus absztrakt-, illetve virtuális metódusokat hív. Amikor a gyermek osztály példányán keresztül hívjuk majd a sablonmetódust, akkor a késői kötés miatt ezen metódusok helyett az őket felülíró gyermekbéli metódusok fognak lefutni.

#### 4.4.3.1. Példa

A sablonmetódust egy példával szemléltetjük: Nincs is jobb a visszatérő ügyfélnél, és már nekünk is van ilyen. A gyártómetódus kapcsán megismert zsíroskenyérbolt visszatér. Olyan jól ment az üzlet, hogy kávét és teát is elkezdték árusítani. Két kolléga el is kezdte a munkát, egyikőjük a teát, míg a másik a kávét kapta feladataul. Estig nyugalom is volt az irodában, amikor is a teás kolléga meggyanúsította a kávést, hogy tőle lopta a kódot. Megnéztük és tényleg nagyon hasonló dolgokat írtak, például: mindenkitőjüknel volt vízforraló függvény és kitölt függvény. Ezek a dolgok mindenkit ital esetében ugyanúgy történnek. Nosza, tegyük őket egy absztrakt osztályba. Ennek Ital lett a neve. Az elkészítési lépéseket (függvényeket: vízforraló, kitölt) betettük egy elkészít nevű függvénybe, nehogy valaki előbb tudja kitölteni az italt, mint a vizet felforralni. Majd jött a következő ötlet, a hozzávalók hozzáadását is ide tettük, sőt csináltunk egy főz függvényt is, hogy teljes legyen a csapat. Mindkét előző függvény absztrakt, mert ezt majd a konkrét osztály kidolgozza, hiszen a kávéba tej és cukor kell, míg a teába citrom. Most már csak a két konkrét Tea és Kávé osztályt kellett kidolgozni, ahol a főz és az édesít (ami néha ugyan savanyít) függvények implementálása volt a feladat. Hazafelé menet még hallottuk a két kollegánk veszekedését: akkor is tőlem loptad! Másnap jött az ötlet, hogy a teába rumot is lehet tenni. Ez egyelőre opcionális lehetőség és nem is élünk vele, mert még a főnök nem egyezett bele. Talán majd télen.

#### 4.4.3.2. Forráskód

```
using System;
namespace ItalKészítő
{
    public abstract class Ital
    {
        public void Elkészít()
        {// Ez a függvény nem kapta meg a virtual jelzőt a sorrend betartása miatt.
            vízforraló(); // kötelező és közös lépés
            főz(); // inversion of control, mert a gyermek főz metódusa fog futni
            édesít(); // kötelező és nem közös lépés
            rum(); // opcionális
```

```

        kitölt();
    }
    private void vízforraló() // kötelező közös lépés
    {// Ennek szintén nem kell felúlíthatónak lennie
        Console.WriteLine("Vízforralás 98..99..100c");
    }
    protected abstract void főz(); // Ki kell dolgozna a konkrét osztálynak.
    protected abstract void édesít(); // Ezek kötelező nem közös lépések.
    protected virtual void rum(){}
    // Ez egy hook, vagyis egy opcionális lépés.
    private void kitölt() // kötelező közös lépés
    {
        Console.WriteLine("Egy szép porceláncsészébe öntöm az italt\n");
    }
}
public class Tea : Ital
{
    protected override void főz()
    {
        // ezt ki kell dolgozni, hiszen másképp főzök teát, mint kávét
        Console.WriteLine("Belógatom és tunkolom a tea filtert");
    }
    protected override void édesít()
    {
        Console.WriteLine("Kis cukor, és egy kis citrom ízlés szerint");
    }
}
public class Kávé : Ital
{
    protected override void főz()
    {
        // ezt ki kell dolgozni, hiszen másképp főzök teát, mint kávét
        Console.WriteLine("Leforrázom a kávét egy jó török kávé kedvéért.");
    }
    protected override void édesít()
    {
        Console.WriteLine("Kis cukor, és egy kis tej ízlés szerint");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Ital tea = new Tea();
        Ital kávé = new Kávé();
        tea.Elkészít(); // a késői kötés miatt a Tea főz és édesít metódusa fut
        kávé.Elkészít(); // a késői kötés miatt a Kávé főz és édesít metódusa fut
        Console.ReadKey();
    }
}
}

```

#### 4.4.3.3. Gyakorló feladat

Az alábbi leírás szerint készítsünk forráskódot. A megoldáshoz használjuk a sablonmetódus tervezési mintát!

#### Feladat

Egészítse ki az alábbi kódrészletet! Írja meg a hiányzó metódusokat és a főprogramot! A feladat megoldásához döntse el, hogy melyik lépés

- közös és kötelező,
- kötelező, de nem közös,
- opcionális.

A kódrészlet:

```
abstract class BuktaSütés
{
    public void Recept()
    {
        tésztaGyúrás();
        töltelékBele();
        beSűtőbe();
        porcukorTetejére();
    }
    private void tésztaGyúrás() // ezt megadtuk segítségnek
    {
        Console.WriteLine("Meggyúrom a téstát.");
    }
    // írja mega hiányzó metódusokat
}
class TurósBuktaSütés : BuktaSütés
{
    // írja mega hiányzó metódusokat
}
class LekvárosBuktaSütés : BuktaSütés
{
    // írja mega hiányzó metódusokat
}
```

#### 4.4.4. Stratégia – Strategy

A Stratégia (angolul: Strategy) egy viselkedési tervezési minta, amely mindenkor egy, és mindenkor csak egy változékony metódust emel ki egy osztály hierarchiába, és mindenkor felelősséget átadással hívja meg a kiemelt metódust.

A stratégia tervezési mintát akkor használjuk, ha van egy (se több, se kevesebb) változékony metódusunk. Változékony metódus például a kosárban lévő áruk árának kiszámítása, hiszen a folyton változó akciók miatt mindenkor másképp kell kiszámolni az árat. Változékony metódus a Kacsa osztályon belül lévő Hápog metódus is, hiszen másképp hápog a házi kacsa, a néma kacsa, a gumikacsa és ott van még Donald kacsa is, aki rekeden hápog.

A változékony metódus kódját nem szabad csak úgy felülírni lépten-nyomon, hiszen ezzel megsértenénk az OCP elvet. A megoldás az, hogy a stratégia tervezési mintát alkalmazva a változékony metódust kiemeljük egy osztályhierarchiába. Figyeljük meg, hogy ez a lépés egy szétválasztás, azaz a Separation of Concerns elv alkalmazása. Szétválasztjuk az osztályt és a változékony metódust. Majd a szétválasztott dolgokat újra összerakjuk objektum-összetétel segítségével.

Az osztályhierarchia tetején egy absztrakt osztály van, ami csak a kiszervezett metódus fejét tartalmazza. Ennek az osztálynak a gyermekei a konkrét stratégiák. Az az osztály, amiből kiszerveztük a metódust, az objektum-összetétel segítségével kap egy referenciát a stratégiára. Ezen referencián keresztül hívjuk a kiszervezett metódust.

A kacsás példánál maradva: hápogási stratégiát alkalmazva a kacsa hápogási viselkedése attól fog függni, hogy melyik konkrét hápogási stratégiát injektáltuk be az objektum-összetételt megvalósító mezőbe.

A kiemelt metódus tevékenységét az eredeti osztály átadja (más szóval: delegálja) a stratégiának az objektum-összetételt megvalósító mezőn keresztül. Így a késői kötés miatt a beinjektált konkrét stratégia metódusa fut le.

Figyeljük meg az alábbi forráskódokban, hogy minden stratégiában csak egy metódus van! Ez így helyes, nem is szabad többnek lennie, mert akkor már nem stratégiának hívnánk. Ha több összetartozó változékony metódust emelünk ki, akkor azt állapottervezési mintának hívjuk.

Az alábbi forráskódban, azaz a kacsás példában minden fenti megállapítás kiemelésre került megjegyzések segítségével, azaz az objektum-összetételt megvalósító sorok, a felelősség beinjektálását megvalósító sorok, a felelősség átadását (idegen szóval delegálását) megvalósító sorok.

#### 4.4.4.1. Forráskód 1.

```
using System;

abstract class Kacsa
{
    public abstract void Hápog();
    public abstract void Repül();
}

class SzépKacsa : Kacsa
{
    HápogásiStratégia hs; // objektum-összetétel
    RepülésiStratégia rs; // objektum-összetétel
    public SzépKacsa(HápogásiStratégia hs, RepülésiStratégia rs)
    {
        this.hs = hs; // felelőség beinjektálása
        this.rs = rs; // felelőség beinjektálása
    }
    public override void Hápog()
    {
        hs.Hápog(); // felelőség átadása a stratégiának, azaz delegáció
    }
    public override void Repül()
    {
        rs.Repül(); // felelőség átadása a stratégiának, azaz delegáció
    }
}
```

```

// ide emelem ki a változékony Hápog metódus kódját
abstract class HápogásStratégia
{
    public abstract void Hápog();
}
// ide emelem ki a változékony Repül metódus kódját
abstract class RepülésiStratégia
{
    public abstract void Repül();
}
// itt jönnek a konkrét repülési és hápogási stratégiák
class NormálHápogás : HápogásStratégia
{
    public override void Hápog()
    {
        Console.WriteLine("HápHáp");
    }
}
class RekedtHápogás : HápogásStratégia
{
    public override void Hápog()
    {
        Console.WriteLine("HákrrrHákrrr");
    }
}
class NemRepül : RepülésiStratégia
{
    public override void Repül()
    {
        Console.WriteLine("Nem tud repülni");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Kacsa donald =
            new SzépKacsa(
                new RekedtHápogás(), // felelőség beinjektálása
                new NemRepül()       // felelőség beinjektálása
            );
        donald.Hápog(); // késői kötés miatt a RekedtHápogás-ból hívja
        donald.Repül(); // késői kötés miatt a NemRepül-ből hívja
        Console.ReadLine();
    }
}

```

#### 4.4.4.2. Forráskód 2.

```

using System;

public abstract class KávéStratégia
{
    public abstract void KávéFőzés();
}
public class GyengeKávé : KávéStratégia
{
    public override void KávéFőzés()
    {
        Console.WriteLine("Gyenge kávét főztél. Ha csak kicsit vagy fáradt.");
    }
}

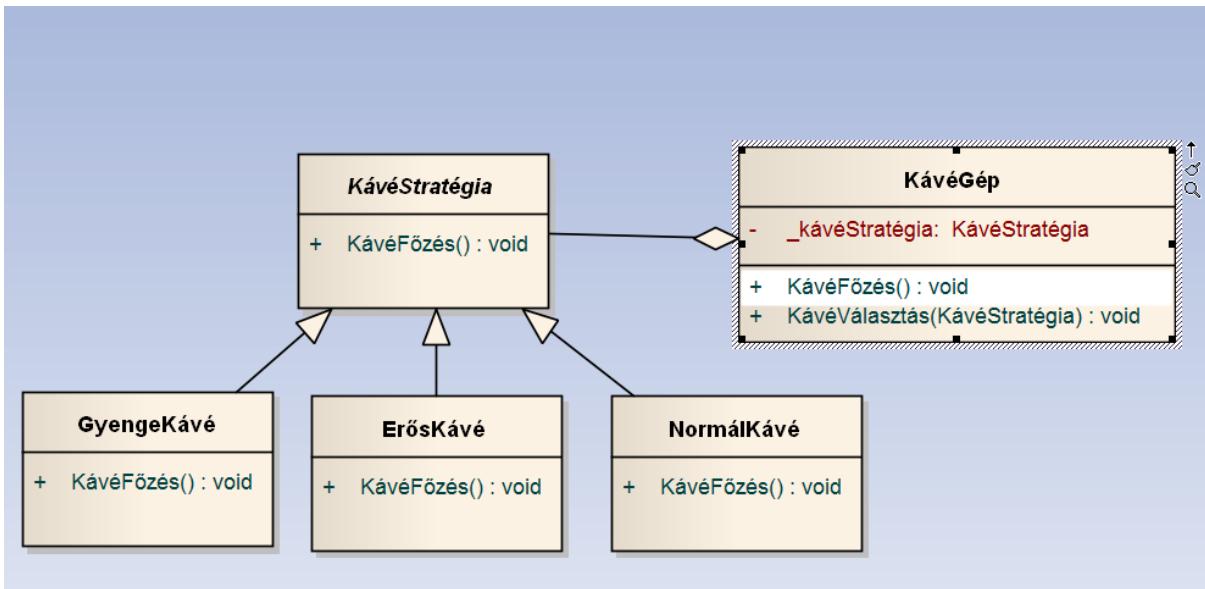
```

```

public class NormalKávé : KávéStratégia
{
    public override void KávéFőzés()
    {
        Console.WriteLine("Normál kávét főztél. Egy átlagos napra.");
    }
}
public class ErősKávé : KávéStratégia
{
    public override void KávéFőzés()
    {
        Console.WriteLine("Egy erős kávé. A hosszú és fárasztó napokra.");
    }
}
public class KávéGép
{
    private KávéStratégia kávéStratégia;
    public void KávéValasztás(KávéStratégia k)
    {
        kávéStratégia = k;
    }
    public void KávéFőzés()
    {
        kávéStratégia.KávéFőzés();
    }
}
class Program
{
    static void Main(string[] args)
    {
        KávéGép automata = new KávéGép();
        automata.KávéValasztás(new GyengeKávé());
        automata.KávéFőzés();
        automata.KávéValasztás(new ErősKávé());
        automata.KávéFőzés();
        automata.KávéValasztás(new NormalKávé());
        automata.KávéFőzés();
        Console.ReadLine();
    }
}

```

#### 4.4.4.3. UML ábra



25. ábra: Példa UML ábra a stratégia tervezési mintára

#### 4.4.4.4. Gyakorlófeladat 1.

A Kacsa osztályt egészítsük ki az Úszik metódussal. Mivel ez egy változékony metódus, szervezzük ki egy külön stratégiába, az ÚszásiStratégia nevű osztályhierarchiába. Ezzel egészítsük ki a SzépKacsa osztályt a RepülésiStartégia és az HápogásiStartégia megoldását követve.

#### 4.4.4.5. Gyakorlófeladat 2.

A KávéStratégia példában az alábbi két sor sorrendjét cseréljük fel:

```
automata.KávéValasztás(new GyengeKávé());
automata.KávéFőzés();
```

A kérdés az, hogy ez milyen futási hibához fog fezetni, illetve, hogyan lehet megakadályozni, hogy ez a hiba fellépjen a KávéGép osztály kódjának megváltoztatásával.

#### 4.4.4.6. Gyakorlófeladat 3.

Készítsük el az alábbi leírásnak megfelelő forráskódot. A feladat megoldásához használjuk a stratégia tervezési mintát!

### Feladat

Micsoda megtiszteltetés! A Magyar Forma-1 istálló megkért minket, hogy készítsünk nekik programot, hogy milyen időben milyen vezetési stílust válasszanak a pilóták. Tettük is a dolgunkat a feladat komolyoságához méltón. Elkészült a kód és mi az esti tévétés helyett kivetítettük, hogy gyönyörködjünk a jól végzett munkánk eredményében. A kreációnk tele volt switch és if-else-if elágazásokkal, amik az időjáráshoz képest más és más vezetési stílusú függvényt hívtak meg. Halk, elégedett mormogás a sötét teremben, amikor valaki felkiáltott: én azt olvastam, hogy ha egy kódban sok a feltételes utasítás, akkor nézd meg, hátha rá tudsz húzni egy stratégiát. Sajnos a főnök is ott volt, úgyhogy nem volt mit tenni. A már megszokott absztrakt osztállyal kezdtük. Ennek neve VezetésiStratégia lett. Ennek van egy

absztrakt függvénye, a Vezet, ez írja le a technikát. Ebből születnek a konkrét osztályok, mint például a CsúszósÚtStratégia vagy a NaposIdőStratégia. Ezek az osztályok kidolgozzák a legjobb stílust a Vezet függvényükben. Már csak egy osztály hiányzott, a Pilóta, melynek van egy Stílusválasztás függvénye, ami paraméterként egy VezetésiStratégia példányt vár. Pilótánk másik függvénye a Versenyez. Ez csak a beállított stratégia Vezet függvényét hívja meg. A verseny napján létrehozunk egy Pilóta példányt az útviszonyoknak megfelelő VezetésiStratégiával, meghívjuk a Versenyez függvényt és vasárnap délután elégedetten nézzük, ahogy megnyeri a versenyt.

#### 4.4.5. Látogató – Visitor

A látogató (angolul: visitor) viselkedési tervezési minta egy vagy több változékony metódust szervezik egy másik osztály hierarchiába. A kiszervezett metódusokat ágens technológiának megfelelően képes fogadni és kiszolgálni.

A látogató tervezési mintát akkor használjuk, ha már van egy kiforrott adatszerkezetünk, pl. bináris fa, lista, sor vagy verem, ami már előreláthatóan nem fog változni, de az adatszerkezetet feldolgozó algoritmusok gyakran változnak. Ilyenkor érdemes szétválasztani az adatszerkezetet és a feldolgozó metódusokat. Az adatszerkezetbe mindenkorban egy fogad (angolul: accept) metódus kell, ami fogadja a látogatókat. A fogad metódus kódja mindenkorban ugyanaz: v.VisitXY(this); ahol az XY az az adatszerkezet, amiben megvalósítjuk a fogad metódust.

Ily módon nagyon egyszerű új látogatót írni, de ha megváltozik az adatszerkezet, akkor az összes látogató kódját frissíteni kell. Ez a látogató minta hátránya.

##### 4.4.5.1. Ágensalapú programozás

Az ágensalapú (angolul: agent based) programozás a kliens–szerver architektúra alternatívája. A kliens–szerver architektúra remekült bevált a gyakorlatban, de néha azért vannak gondok. Abban az esetben, ha a kliens által elvégzendő számoláshoz nagyon sok adatra van szükség a szerverről, de maga a számolás eredménye ehhez mérten elenyésző méretű, akkor jobb az ágensalapú programozás.

A kliens–szerver architektúrában adat mozog a szerver és a kliens között. Ezzel szemben az ágensalapú programozásnál végrehajtható kód mozog a két gép között. Ezt a végrehajtható kódot hívjuk ágensnek.

Az ágens a kliensről indul valamilyen speciális feladattal. Eljut a szervergépre. Az ott lévő adatok segítségével elvégzi a feladatát, az eredményt megjegyzi, majd visszatér, és a kliensen tájékoztatja megbízóját az eredményről.

Legyen most az a feladatunk, hogy megtudjuk, hány perces a legújabb szuper mozifilm. Két lehetőségünk van:

- Vagy letöltsük a filmet a szolgáltatótól és miután lejött, megnézzük, hány perces. Ez a klasszikus kliens–szerver megoldás.
- Vagy írunk egy kis scriptet, feltöltsük a scriptet a szolgáltató szerverére, a script megnézi, hány perces a film, és visszakapjuk az értéke. Ez már nagyon hasonlít az ágensalapú megoldáshoz, annyi a különbség, hogy az ágenst nemnek kell távolról indítani, nekem csak elküldeni kell, és ha megérkezik, akkor magától elindul.

Látható, hogy a második megoldás sokkal kisebb hálózati adatforgalommal jár. Tehát látható az ágensalapú megoldás előnye. Ugyanakkor ez a módszer távolról se terjedt el, hiszen ki szeretné, hogy mindenféle külvilágóból érkező kód lefuthasson a szerverén. Ezek a kódok könnyen lehetnek rosszindulatúak is.

A megoldás az úgynevezett homokozó (angolul: sandbox) használata. A szerveren létrehozok egy homokozót, ami lényegében egy virtuális szerver saját erőforrásokkal, csak olvasható állományokkal, de egy kicsi írható-olvasható háttértárral is. A homokozó általában a CPU és a memória 10%-át kaphatja meg maximálisan és időnként újraindul a homokozó, hogy a beragadt ágensek ne foglalják a helyet a működők elől.

A homokozóban minden szabad, kivéve egyet, kijönni a homokozóból. Tehát szabad lökdösődni, futkározni, sőt verekedni is, de nem szabad a homokozón kívül üldögélő szülőket megdobálni.

Az ágensalapú programozás tervezési minta megfelelője a látogató (angolul: visitor) tervezési minta. A látogató mintában az adatszerkezet a szerver, a fogad metódus a homokozó, a látogató pedig az ágens.

#### 4.4.5.2. Forráskód

```
using System;

public abstract class BFa
{
    BFa bal, jobb;
    public BFa Bal { get { return bal; } }
    public BFa Jobb { get { return jobb; } }
    public BFa(BFa bal, BFa jobb)
    {
        this.bal = bal;
        this.jobb = jobb;
    }
    public BFa() : this(null, null) { }
    public abstract void Accept(Visitor v);
}

public class Fa : BFa
{
    int szám;
    public int Szám { get { return szám; } }
    public Fa(int szám, BFa bal, BFa jobb) : base(bal, jobb)
    {
        this.szám = szám;
    }
    public override void Accept(Visitor v)
    {
        v.VisitFa(this);
    }
}

public class Levél : BFa
{
    int szám;
    public int Szám { get { return szám; } }
    public Levél(int szám) : base()
    {
        this.szám = szám;
    }
    public override void Accept(Visitor v)
    {
```

```

        v.VisitLevél(this);
    }
}
public abstract class Visitor
{
    public abstract void VisitFa(Fa f);
    public abstract void VisitLevél(Levél f);
}
public class SumVisitor : Visitor
{
    int sum = 0;
    public int Sum { get { return sum; } }
    public override void VisitFa(Fa f)
    {
        sum += f.Szám;
        if (f.Bal != null) f.Bal.Accept(this);
        if (f.Jobb != null) f.Jobb.Accept(this);
    }
    public override void VisitLevél(Levél f)
    {
        sum += f.Szám;
    }
}
public class MaxVisitor : Visitor
{
    int max = int.MinValue;
    public int Max { get { return max; } }
    public override void VisitFa(Fa f)
    {
        if (f.Szám > max) max = f.Szám;
        if (f.Bal != null) f.Bal.Accept(this);
        if (f.Jobb != null) f.Jobb.Accept(this);
    }
    public override void VisitLevél(Levél f)
    {
        if (f.Szám > max) max = f.Szám;
    }
}

class Program
{
    static void Main(string[] args)
    {
        BFa fa = new Fa(5, new Levél(8), new Levél(3));
        SumVisitor sumv = new SumVisitor();
        MaxVisitor maxv = new MaxVisitor();
        fa.Accept(sumv);
        fa.Accept(maxv);
        Console.WriteLine("A fában lévő számok összege: {0}", sumv.Sum);
        Console.WriteLine("A fában lévő legnagyobb szám: {0}", maxv.Max);
        Console.ReadLine();
    }
}

```

#### 4.4.5.3. Gyakorlófeladat

Ezt a mintát elsőre nehéz megérteni, de ha a fenti fához írunk egy-két további látogatót a meglévő Sum és Max mellé, mondjuk a Min, vagy a ToString látogatót, akkor elsajátítjuk a minta működését. Tehát a fenti példát egészítsük ki a MinVisitor és a ToStringVistor osztályokkal, illetve ezek egy-egy példányát hívjuk meg a főprogramból.

## 5. Programozási technológiák – Technológiák

Ebben a fejezetben azokat a technológiákat ismertetjük, amelyek használata szükséges a tárgyhoz kötődő beadandó program elkészítéséhez.

### 5.1. Tiszta kód

A tiszta kód (angolul: clean code) fogalmának nagyon sok jelentésrétege van. Ebben a jegyzetben ezeknek csak egy részével foglalkozunk. Sokkal részletesebb foglalkozik a témaival Robert C. Martin Tiszta kód című könyve (lásd ajánlott irodalom).

A tiszta kód egyik legfontosabb ismérve, hogy könnyen olvasható. Ez azért fontos, mert a forráskódot nem magunknak írjuk, hanem más programozóknak, akik tovább fogják azt fejleszteni. Még abban az esetben, ha saját magunk fejlesztjük tovább, akkor is egy idegen programozónak dolgozunk, mert gyakran megfigyelhető, hogy egy néhány hónappal korábban írt programról már nem ismerjük fel, hogy azt mi írtuk.

A könnyen olvashatóság egyik ismérve, hogy a kód olvasása közben nem kell ide-oda ugrálnunk a forráskódban. Ezt úgy érjük el, hogy a belépési pont az első metódus. Ha ebben a metódusban meghívjuk az A, B, és C metódusokat, akkor ezek ilyen sorrendben követik ezt a metódust.

A tiszta kód másik ismérve, hogy nem tartalmaz megjegyzést. Természetesen az elején lehet copyright megjegyzés, de nem lehet benne olyan megjegyzés, ami egy metódust, egy mezőt vagy egy kód részletet magyaráz. Ha ilyen van a kódunkban, azzal beismérjük, hogy nem használunk beszédes neveket, illetve, hogy konvenciókat használunk, amit ha a többi programozó nem tart be, akkor összeomlik a programunk. A másik probléma a megjegyzésekkel, hogy gyakran elszakadnak a kódtól, nem változnak együtt a kóddal.

A tiszta kód ajánlást ad a metódusok méretére. Egy tiszta kód 5-6 soros, csak 1-2 if utasítást tartalmaz, 1 ciklust, kivételes esetben többet. Az ilyen metódusokat könnyű megérteni. Ha ezek a metódusok olyan metódusokat hívnak, amelynek beszédes neve van, akkor gyakran ezeket már meg se kell nézni.

Fontos, hogy kerüljük a mellékhatásokat. Mellékhatás az, ha egy metódus megváltoztatja a környezetét, azaz, ha megváltoztatja valamelyik mező értékét (belőállapot-átmenet), kiír a képernyőre, valamelyik portra üzenetet küld. Ugyanakkor nem lehet teljesen elkerülni a mellékhatást. Az ajánlás szerint, ha egy metódus visszaad valamit, azaz kiszámol valamit, akkor ne legyen mellékhatása. Ha void-ot add vissza, akkor lehet mellékhatása.

A paraméterlistára is vannak ajánlások. Törekedjünk a rövid paraméterlistákra. A legjobb, ha egy paraméterünk sincs, egy vagy kettő elfogadható, három már soknak számít. Ne használunk logikai paramétert, illetve ne használunk kimeneti paramétert.

#### 5.1.1. Cserkészszabály

A tiszta kód egyik hajtóereje a cserkészszabály. A cserkészszabály (angolul: boy scout rule), amely így szól:

- Mindig hagyd a táborhelyet tisztábban, mint ahogy találtad.

Ez szoftverfejlesztés esetén azt jelenti, hogy ha a verziókövető rendszerből kiveszünk egy osztályt, akkor kötelesek vagyunk tisztábban visszatenni. Gondoljunk bele, ha ezt mindenki betartaná, akkor a forráskódunk egyre tisztább és tisztább lenne.

#### 5.1.2. Rothadó kód

A tiszta kód ellentéte a rothadó kód (angolul: rotting code). Akkor mondjuk, hogy a kód rothad, ha a programozóink már nem mernek hozzányúlni, mert attól félnek, hogy egy hibajavítás egy másik hibát fog eredményezni.

A rothadó kód nagyon lelassítja a fejlesztést, ami ahhoz vezet, hogy a versenytársak megelőzik a cégeket, végül tönkremegyünk.

Rothadó kód egy szép tiszta kódból lesz. Tudjuk, hiszen ez a fő elvünk, hogy a kód állandóan változik. A szép tiszta kód is változik, és ahogy változik, kezd szép lassan csúnya rothadó kóddá változni. Ennek legfőbb oka, hogy nem tartjuk be a tervezési alapelveket, például switch szerkezetet vagy if – else – if szerkezetet használunk, illetve nem refaktoráljuk a kódot, hogy bevezessünk egy-egy tervezési mintát, amikor már látszik, hogy érdemes lenne használni.

A rothadó kódot megelőzik a kódszagok (angolul: code smell). A szakirodalom nagyon sok ilyen szagot ismer. Ezek közül a legegyszerűbbek:

- túl nagy metódusok,
- túl nagy osztályok,
- osztályok sok mezővel,
- duplikált kódrészletek, copy/paste kódrészletek,
- burjánzó if – else –if és/vagy switch szerkezetek,
- megvalósításra programozás,
- erős csatoltság, pl. Demeter törvényének megsértésével.

Illetve a rossz szagok. Amíg a kódszag inkább csak jelzés, hogy a szag forrását érdemes lenne ellenőrizni, addig a rossz szag már komoly figyelmeztetés, hogy hozzá kell fogni a kód kitisztításához. A szakirodalom a következő rossz szagokat különbözteti meg:

- Merevség (angolul: rigidity): A rendszert nehéz megváltoztatni, egy-egy változtatás sokkal több időbe telik, mint amennyit becsültünk a feladatra, egy-egy változás nem várt nehézségekbe ütközik.
- Törékenység (angolul: fragility): A rendszer nagyon érzékeny a változásokra, egy-egy változás nagyon sok további változást von maga után, egy-egy hibajavítás minden újabb és újabb hibákat szül.
- Mozdulatlanság (angolul: immobility): A rendszer egyes részeit funkciójuk szerint jó lenne újra felhasználni, de az újrafelhasználás komoly erőfeszítéseket igényelne.
- Viszkozitás (angolul: viscosity): Egy-egy változtatás nagyon körülményes, inkább a „vállunk felett átnyúlós”, „workaround” megoldást választjuk. Másik formája, amikor olyan lassú a fordítás, hogy arra törekszünk, hogy csak kevés osztályt kelljen újrafordítani, ezért nem oda teszem a metódusokat, ahova kellene, hanem oda, amit könnyű újrafordítani.

- Szükségtelen komplexitás (angolul: needless complexity): Sok változata van. A legmarkánsabb, amikor túlbonyolítunk egy osztályt, hogy egy későbbi változást könnyű legyen kezelní, de az a változás sohasem következik be.
- Homályosság (angolul: opacity): Nehezen olvasható, nehezen érthető kód.

## 5.2. Egységeszt

Az egységeszt, vagy más néven unit-teszt a metódusok tesztje. Egy egységesztben rögzítjük a metódus paramétereit, illetve megmondjuk, hogy erre a bemenetre mi az elvárt kimenet. Lefuttatjuk a metódust a rögzített paraméterekkel. Így megkapjuk az aktuális visszatérési értéket. A kettőt összehasonlítjuk, s ha ugyanaz, akkor az egységeszt zöld lesz, ha nem, akkor piros.

Egy példa egységeszt:

```
[TestMethod]
public void TestMethod1()
{
    SzámolóGép target = new SzámolóGép();
    int a = 2;
    int b = 2;
    int expected = 4;
    int actual = target.Összead(a, b);
    Assert.AreEqual(expected, actual);
}
```

Jól látható, hogy a SzámolóGép osztály Összead metódusát teszteljük. Ennek két paramétere van, a és b. Ezeket fixáljuk, jelen esetben mind a kettő 2 lesz. Megadjuk, hogy erre a bemenetre az elvárt kimenet (a kódban: expected) 4. Kiszámoljuk az aktuális kimentet (a kódban: actual). Majd a kettőt összehasonlítjuk.

Az egységesztek legnagyobb előnye, hogy egyszerre el tudom indítani az összes tesztesetet, egyetlen gombnyomással. Így a regressziós teszt nagyon egyszerűvé válik, hiszen egy-egy változtatás után csak le kell futtatnom az összes egységesztet, ami csupán egy gombnyomás!

### 5.2.1. Egységeszt készítése Visual Studio 2013 esetén

Ahhoz, hogy egységesztet, vagy más néven unit-tesztet tudunk írni Visual Studio 2013 segítségével, ahhoz az alábbi 3 előkészítő lépést kell megtennünk:

1. Unit Test Projekt hozzáadása az aktuális Solution-höz: A SolutionExplorer-ben (View -> Solution Explorer) a Solution-ön jobb klikk, majd Add -> New project ... -> Visual C# -> Test -> Unit Test Project -> OK.
2. A Unit Test Projekten belül referenciák kibővítése a projekttel, amit tesztelni akarunk: A Solution Explorer-ben a UnitTestProjecten1-en belül a References-en jobb klikk, majd Add Reference... -> Solution -> Projects, az Application neve mellett jelölő négyzetbe pipa, majd OK.
3. A UnitTest1.cs fájlba beírni egy using utasít, hogy hivatkozni tudunk a tesztelendő osztályokra: ha a tesztelni kívánt osztályok a ConsoleApplication4 névtérben vannak, akkor a „using ConsoleApplication4;” utasítást kell beleírni.

Ha minden sikerült, akkor a UnitTest1.cs fájlunk így fog kinézni:

```

using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using ConsoleApplication4;

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}

```

Ha a fenti leírás alapján nem sikerülnek az előkészítő lépések, akkor érdemes megnézni egy sokkal részletesebb útmutatót, amely itt található: <http://www.visualstudio.com/en-us/get-started/create-and-run-unit-tests-vs.aspx>.

#### 5.2.2. Egységeszt készítése Visual Studio 2008 és 2010 esetén

Egységesztet, vagy más néven unit-tesztet Visual Studio 2008 és 2010 programmal is készíthetünk. Ehhez a következő menükön kell végigmenni: Test -> New Test... -> Unit Test Wizard -> a metódus kiválasztása, amelyhez egységesztet szeretnénk készíteni.

Egy példa egységeszt amelyet a fenti eljárással generáltunk a maximum értékét visszaadó max(int a, int b) metódushoz:

```

[TestMethod()]
public void MaxTest()
{
    Program target = new Program(); // TODO: Initialize to an appropriate value
    int a = 10; // TODO: Initialize to an appropriate value
    int b = 20; // TODO: Initialize to an appropriate value
    int expected = 20; // TODO: Initialize to an appropriate value
    int actual;
    actual = target.max(a, b);
    Assert.AreEqual(expected, actual);
}

```

A két mező értékét és az elvárt visszatérési értéket kézzel állítjuk be.

### 5.3. Tesztvezérelt programozás – Test Driven Development – TDD

A tesztvezérelt programozás (angolul: Test Driven Development, TDD) egy jól bevált módszer (angolul: Best Practice), amely az agilis módszerekkel, illetve a tiszta kód eszményével együtt terjedt el. A TDD első hallásra nagyon furcsa: ne a metódust írjuk meg, majd utána a hozzá tartozó egységesztet, vagy más néven unit-tesztet, hanem épp fordítva, először írjuk az egységesztet, úgy, hogy a tesztelt metódusnak még a feje sincs kész, majd csak ezután írjuk meg a metódust.

A TDD még ettől is továbbmegy, azt ír elő, hogy a metódusból csak annyit írunk meg, hogy épp átmenjen a legutoljára megírt egységeszten. Ezután írunk egy új egységesztet a már félkész metódushoz, ami egy eddig le ne fedett esetet tesztel, aztán írjuk meg a metódusba ezt az esetet. És így tovább, amíg minden esetet le fedünk teszesettel, illetve kóddal a metódusban.

Tehát a TDD lépései így foglalhatók össze:

- Egységesztet írunk először, csak utána metódust.
- minden egységeszt az eddig nem tesztelt lehető legegyszerűbb esetet írja le.
- A metódusból csak annyit írunk meg, hogy épp átmenjen az összes egységeszten.

Robert C. Martin (népszerű nevén Uncle Bob) is három pontban írja le a TDD lényegét. Az ő megfogalmazását eredeti angol nyelven adjuk meg:

- „Write no production code except to pass a failing test.” Azaz: Ne írj éles kódot, kivéve hogy egy hibás tesztet kielégíts.
- „Write only enough of a test to demonstrate a failure.” Azaz: Csak annyit tesztet írja, amely elegendő egy hiba demonstrálására.
- „Write only enough production code to pass the test.” Azaz: Csak annyi éles kódot írj, hogy kielégítsd a tesztet.

A két megfogalmazás egy kicsit máshova helyezi a hangsúlyt, de mind a kettő kiemeli, hogy a metódusokat apró lépésekben fejlesztjük, úgy, hogy minden lépés előtt írunk egy egységeszst.

Ha betartjuk a TDD ajánlásait, akkor annak rengetek pozitív hatása van. Lássuk ezeket:

- Így nem felejtünk el egységeszstet írni, ami azért jó, mert így sok-sok tesztesetünk lesz.
- Illetve így a tesztelő fejünkkel gondolkozunk először és nem a programozó fejünkkel. A programozó fejünk szép és hatékony kódot szeretne írni, ami gyakran nehezen tesztelhető. Viszont a tesztelő fejünket használva először biztosak lehetünk abban, hogy a metódusunkat könnyű lesz tesztelni.
- Nyugodt szívvvel merünk hozzájárulni a kódhoz, mert egy-egy változás után a regressziós teszt futtatása nagyon egyszerű, így van egy biztosítás a zsebünkbe, hogy ha esetleg a változás hibát okozna, akkor azt észre fogjuk venni. Tehát van bátorságunk szépíteni a kódot. Ezzel elkerüljük, hogy a forráskód ne váljon spaghetti kódodá, vagy ami még rosszabb, bűzhödt, rothadó kódodá.
- A tesztek a kód nagy részét lefedik, így a program magas minőségeben biztosak lehetünk.
- Sokkal kevesebb időt kell hibakereséssel, nyomkövetéssel töltünk.
- Az egységeszt a legjobb, programozóknak szóló dokumentáció, így könnyebb megérteni a kódunkat.

### 5.3.1. TDD és a tiszta kód

A tiszta kód alapja a TDD. Ha TDD-t használunk, akkor van sok-sok egységesztünk, ami bátorságot ad, hogy hozzájárulunk a kódhoz. Ha látjuk, hogy kezd romlani a kódunk, akkor merjük azt refaktorálni. Emiatt a TDD a tiszta kód alapja!

### 5.3.2. Piros – Zöld – Piros

Tudjuk, hogy a sikertelen egységeszt, más néven unit-teszt jelzése minden keretrendszerben piros, a sikeresé zöld. Azt is tudjuk, hogy a TDD előírja, hogy a metódusból csak annyit írunk meg, hogy épp átmenjen a legutoljára megírt egységeszten. Ezután írunk egy új egységesztet a már félkész metódushoz, ami egy eddig le ne fedett esetet tesztel, aztán írunk meg a metódusba ezt az esetet. És így tovább, amíg minden esetet lefedünk tesztesettel, illetve kóddal a metódusban.

Azaz először lesz egy piros egységeszt, majd ez zöld lesz, aztán írunk egy újat, ami szintén piros lesz, majd zöld, és így tovább. A TDD ezen tulajdonsága miatt a TDD-t szokás Piros – Zöld – Piros tesztelésnek is hívni.

### 5.3.3. TDD esettanulmány

Lássunk egy egyszerű példát TDD használatára. TDD segítségével fogjuk lefejleszteni a bináris fában lévő számok összegét kiszámító metódust. minden egységeszt után közöljük az egész bináris fa osztályt.

A lenti példában, ami több körre van osztva, az egységeszteket a UnitTest1.cs fájlba kell másolni, a BFa osztályt pedig a Program.cs nevű fájlba.

#### 5.3.3.1. Első kör

```
[TestMethod]
public void TestAzÜresFábanASzámokÖsszegeNulla ()
{
    BFa target = new BFa();
    int expected = 0;
    int actual = target.Szum();
    Assert.AreEqual(expected, actual);
}

public class BFa
{
    public int Szum() { return 0; }
```

#### 5.3.3.2. Második kör

```
[TestMethod]
public void TestSzumCsakGyökérbőlÁllóFa()
{
    BFa target = new BFa(1);
    int expected = 1;
    int actual = target.Szum();
    Assert.AreEqual(expected, actual);
}

public class BFa
{
    int szám;
    public BFa() { szám = 0; }
    public BFa(int szám) { this.szám = szám; }
    public int Szum() { return szám; }
}
```

#### 5.3.3.3. Harmadik kör

```
[TestMethod]
public void TestSzumeEgyBalágúFával()
{
    BFa target = new BFa(1, new BFa(2), null);
    int expected = 3;
    int actual = target.Szum();
    Assert.AreEqual(expected, actual);
}

public class BFa
{
    int szám;
```

```

    BFa bal, jobb;
    public BFa() { szám = 0; }
    public BFa(int szám) { this.szám = szám; }
    public BFa(int szám, BFa bal, BFa jobb) : this(szám)
    {
        this.bal = bal;
        this.jobb = jobb;
    }
    public int Szum()
    {
        int szum = szám;
        if (bal != null) szum += bal.Szum();
        return szum;
    }
}

```

#### 5.3.3.4. Negyedik kör

```

[TestMethod]
public void TestSzumEgyNagyFával()
{
    BFa target = new BFa(1,
        new BFa(2, new BFa(3), null),
        new BFa(4, null,
            new BFa(5, new BFa(6), new BFa(7))));
```

```
    int expected = 1+2+3+4+5+6+7;
```

```
    int actual = target.Szum();
```

```
    Assert.AreEqual(expected, actual);
```

```
}
```

```

public class BFa
{
    int szám;
    BFa bal, jobb;
    public BFa() { szám = 0; }
    public BFa(int szám) { this.szám = szám; }
    public BFa(int szám, BFa bal, BFa jobb) : this(szám)
    {
        this.bal = bal;
        this.jobb = jobb;
    }
    public int Szum()
    {
        int szum = szám;
        if (bal != null) szum += bal.Szum();
        if (jobb != null) szum += jobb.Szum();
        return szum;
    }
}
```

#### 5.3.4. Piros – Zöld – Kék – Piros

Mint már írtuk, a TDD másik neve a Piros – Zöld – Piros tesztelés. A fenti esettanulmányban nagyon jól látható, hogy a Szum metódust lépésről lépésre írjuk meg, úgy, hogy először megírjuk az új esetet lefedő egységesztet, majd az ezt megvalósító kódot.

A fenti esettanulmányból nem látszik viszont, hogy néha szépítjük is (idegen szóval refaktoráljuk) a kódot. Ha azt vesszük észre, hogy a kódban redundáns sorok vannak, vagy lehetne alkalmazni egy tervezési mintát, esetleg szét lehetne választani valamit, amire eddig nem volt szükség, akkor

kódszépítés következik. Ez a kék fázis. Elvileg minden zöld lépés után kék jön, ezért a TDD-t nevezhetjük Piros – Zöld – Kék – Piros tesztelésnek is.

A fenti példánál maradva, amikor bevezetjük a harmadik BFa konstruktort, akkor általában az először így néz ki:

```
public BFa(int szám, BFa bal, BFa jobb)
{
    this.szám = szám;
    this.bal = bal;
    this.jobb = jobb;
}
```

Ez tartalmaz egy redundáns sort is, hiszen a „this.szám = szám;” utasítást végrehajtjuk az egy paraméteres konstruktorkban is. Miután ezt észrevettük, érdemes kódszépítést végrehajtani, hogy eltüntessük a redundáns sort. Ennek ez lesz a végeredménye:

```
public BFa(int szám, BFa bal, BFa jobb) : this(szám)
{
    this.bal = bal;
    this.jobb = jobb;
}
```

### 5.3.5. TDD a szoftverfejlesztés kettős könyvelése

A szoftverfejlesztés nagyon érzékeny, könnyű helytelen kódsort írni. Szerencsé esetben a kódsor csak szintaktikailag helytelen. Ilyenkor a fordító kiírja, hogy hibás a sor, mi a hiba benne, jó esetben még megoldási javaslatot is ad. Viszont, ha szemantikai hiba van benne, akkor nagyon nehéz megtalálni a hibát, ami könnyen futási hibához vezethet.

A könyvelés is hasonlóan érzékeny terület. Elég egy számot rosszul bevezetni a könyvelésbe, és már az egész elromlik. A könyvelés területén ezt a kettős könyveléssel védi ki. minden számlát felvesznek a Tartozik és a Követel oldalon is. A két oldal összegének mindig nullának kell lennie. Ha valamelyik oldalon rosszul veszünk fel egy számlát, akkor rögtön látjuk, hogy hiba van. Könnyen belátható, hogy a kettős könyvelés nagyon hasznos!

A TDD a szoftverfejlesztés kettős könyvelése. minden esetet lefedünk teszttel és kóddal is. Ha bármelyik oldalon hibázunk, akkor azt azonnal észrevesszük.

Erre a gyönyörű összefüggésre Robert C. Martin (népszerű nevén Uncle Bob) hívja fel a figyelmünket Clean Code című videósorozatában.

## 5.4. Naplózás

A naplózás vagy log készítés (angolul: logging) az a folyamat, amikor a program futása közben naplóbejegyzéseket készítünk, hogy váratlan hiba esetén, azaz amikor a program futási hibával áll meg, képesek legyünk megmondani, hogy mi volt a hiba oka. Arra nem számíthatunk, hogy a felhasználó pontosan elmondja, hogy mit csinált, ami a hibát okozta. Ehelyett olyan részletes naplót kell vezetni, hogy abból kiderüljön a hiba oka.

Általában minden metódus elején és végén készítünk egy naplóbejegyzést, de akár minden értékkadást is körbevehetünk naplózással. Szokás naplózási szinteket használni, amit egy konfigurációs

állományban lehet beállítani. Egyes szinten csak a fő eseményeket naplózzuk, kettes szinten a program fő funkciói már követhető a naplóból, hármás szinten minden metódushívásnál készítünk naplóbejegyzést, négyes szinten szinte minden értékkadásról készítünk naplóbejegyzést.

A naplózást úgy kell felfognunk, mintha ez lenne az egyetlen eszközünk a nyomkövetés helyett. Amit nyomkövetés során nézni szoktunk, azt kell naplózni is!

A naplóállományok általában nagyok, nehezen kezelhetők sima szövegszerkesztővel. Ezért speciális szövegnézegető szoftverrel érdemes megtekinteni őket, amiket tail (magyarul: farok) programoknak hívunk.

Ha tail-lel nézzük a naplót, lásd a tail parancs -f kapcsolóját, és keletkezik egy új sor, akkor az rögtön meg is jelenik. Az egyes szavakhoz színeket rendelhetünk, például az error szó általában piros, így könnyebben megtalálhatjuk, amit keresünk. Egy egyszerű tail program pl. a BareTail, de sok más hasonlóan színvonalas és ingyenes tail program is található.

Naplóbejegyzéseket Java esetén általában a Log4J csomag használatával készítünk. Ez egy apache.org-os projekt és széles körben elfogadott, ipari szabványnak tekinthető. Az apache.org-os projektekről általában is elmondható, hogy magas színvonalúak.

C# esetén a beépített Trace osztályt használhatjuk, amely a System.Diagnostics névtérben van. Az alábbi egyszerű példaprogram létrehoz egy napló állományt C:\Temp\log.txt néven, illetve minden naplóbejegyzést kiír a konzolra is. Ezt úgy érjük el, hogy két TraceListener-t adunk hozzá a Trace-hez. Megjegyzés: Ebből látható, hogy a Trace és a TraceListener-ek között megfigyelő (angolul: observer) tervezési minta van.

```
using System;
using System.Diagnostics;
class Program
{
    static void Main(string[] args)
    {
        TraceListener t1 = new TextWriterTraceListener(@"C:\Temp\log.txt");
        TraceListener t2 = new ConsoleTraceListener();
        Trace.Listeners.Add(t1);
        Trace.Listeners.Add(t2);
        Trace.TraceInformation("Napló indul");
        // todo: hasznos rész
        Trace.TraceInformation("Napló vége");
        Trace.Flush();
        Console.ReadLine();
    }
}
```

Fontos megjegyezni, hogy a TraceInformation hatására a bejegyzés csak egy pufferbe kerül be, amit a rendszer csak akkor ürít, ha az betelt, vagy kiadjuk a Trace.Flush parancsot. Ezért érdemes beállítani az AutoFlush lehetőséget.

A naplózásról tudni kell, hogy lassítja a program futását, de naplózni ennek ellenére is kell, mert ha az ügyfélnek hiba történik, akkor minden pénzt megér egy jó naplófájl.

Érdemes még megjegyezni, hogy kritikus rendszereknél, pl. pénztári rendszer, arra is fel kell készíteni a naplót, hogy segítségével visszaállíthassuk az utolsó konzisztens állapotot. Erről bővebbet a Memento tervezési mintából tudhatunk meg, amit nem tárgyal ez a jegyzet.

## 5.5. Aspektusorientált programozás

Az aspektusorientált programozás alapgondolata az, hogy használunk tiszta osztályokat, olyan osztályokat, amiket nem szennyez be a naplázás, a jogosultság kezelés, és más ilyen, mindenhol előjövő feladatok. Az egyszerűség kedvéért vegyük a naplázást (általában minden jegyzetben ez a példa).

Naplózni kell a hallgató osztályban és a kurzus osztályban is. Ez azt mutatja, hogy a naplázás egy aspektus. Aspektusok azok a feladatok, amiket minden, vagy legalábbis nagyon sok különböző feladatú osztálynak el kell végeznie. Ezeket átmetsző ügyeknek (angolul: crosscutting concerns) is nevezzük.

Az aspektusokat érdemes külön szervezni, mert így tisztább osztályokat kaphatunk. Ha tiszta osztályokra gondolok, nemek minden a Java nyelv POJO fogalma jut eszembe. Ez a Plain Old Java Object rövidítése. A Java programozók ezt olyan értelemben használják, hogy olyan osztály, amiben nincs JEE (Java Enterprise Edition) specifikus kód részlet, nincs benne naplázás és más egyéb „fertőzés”. A POJO megfelelője C#-ban a POCO, Plain Old C# Object.

Az aspektusorientált programozás fogalmai:

- Aspektus (aspect): Ide kell kiszervezni az átmetsző követelményeket.
- Szövés (weaving): Az a folyamat, ami során az aspektusokat beleszőjük a programba.
- Programpontok (join point): Ezeken a pontokon lehet beleszóni a programba az aspektusokat. Általában metódusok előtt és után érdemes aspektusokat beleszóni a programba, de akár a metódus helyére is kerülhet az aspektus, azaz lecserélhető a metódus kódja.
- Pontszűrő (pointcut): Itt rendeljük össze az aspektusokat és a programpontokat, azaz megmondjuk, hogy melyik aspektust melyik programpontról kell beleszóni a programba.

A szövésnek több módja van, aszerint, hogy az aspektusok mikor kerülnek bele a program kódjába:

- fordítás idejében történő szövés:
  - az aspektus a forráskódba kerül,
  - az aspektus a köztes (byte) kódba kerül.
- futási időben történő szövés:
  - az aspektus a köztes (byte) kódba kerül.

Ha a forráskódba kerül az aspektus, akkor az megoldható egy egyszerű előfordítóval. A mai modern rendszerek a köztes kódba szövik az aspektusokat.

Az alábbiakban megmutatjuk, hogyan lehet C#-ban a PostSharp keretrendszer segítségével aspektusorientáltan programozni.

```
using System;
using PostSharp.Aspects;
using System.Diagnostics;
[Serializable]
public sealed class TraceAttribute : OnMethodBoundaryAspect
{
```

```

private readonly string category;
public TraceAttribute(string category)
{
    this.category = category;
}
public string Category { get { return category; } }
public override void OnEntry(MethodExecutionArgs args)
{
    Trace.WriteLine(string.Format("Entering {0}.{1}.",
        args.Method.DeclaringType.Name, args.Method.Name), this.category);
}
public override void OnExit(MethodExecutionArgs args)
{
    Trace.WriteLine(string.Format("Leaving {0}.{1}.",
        args.Method.DeclaringType.Name, args.Method.Name), this.category);
}
}
public class HasznosOsztály
{
    [Trace("Hasznos")]
    public void HasznosMetódus1()
    {
        Console.WriteLine("Valami hasznos");
    }

    [Trace("Hasznos")]
    public void HasznosMetódus2()
    {
        Console.WriteLine("Valami nagyon hasznos");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Trace.Listeners.Add(
            new TextWriterTraceListener(Console.Out));
        HasznosOsztály hasznos = new HasznosOsztály();
        hasznos.HasznosMetódus1();
        hasznos.HasznosMetódus2();
    }
}

```

A fenti példakód csak akkor fog működni, ha installáltuk a PostSharp rendszert a Visual Studio-hoz és a projekthez hozzáadtuk a PostSharp-ot. Ezt a SolutionExplorer-ben a projektre jobb kíkk után előjövő menüben tehetjük meg.

Figyeljük meg, hogy a fenti példában a HasznosOsztály egy POCO, amit attributomokkal láttunk el. Az attributomok az osztály, vagy a metódus neve elő írt szögletes zárójel közti kifejezések, amelyek befolyásolják a fordító működését.

Ebből a kis példakódból látható, hogy új aspektust írni úgy lehet, hogy az OnMethodBoundaryAspect osztályból származtattunk egy osztályt. Természetesen lehet más alaposztályból is származtatni, de erre most nem térünk ki.

Az új osztálynak szerializálhatónak kell lennie, amit a [Serializable] attributommal jelölünk. Az osztály, amit létrehozunk, végső soron egy attributum lesz, így az osztály nevének XYZAttribute alakúnak kell lennie és az XYZ-t attributomként használhatjuk. A mi esetünkben az XYZ helyén Trace áll. Az

attributom lehet paramétere is, ami abból látszik, hogy a konstruktornak van-e paramétere. A mi esetünkben egy paramétere van.

Az aspektuson belül hozzuk létre a programpontokat (join point) az „On” kezdetű metódusokkal. Ez esetben kettőt írtunk meg: az OnEntry és az OnExit metódusokat. Amit az OnEntry-be írunk, az a kiszűrt metódusok előtt fog lefutni, amit az OnExit-be írunk az utána.

A Pontszűrő (pointcut) ez esetben a Trace attributum. Az a metódus, ami elé odaírjuk ezt a metódust, azelőtt le fog futni az OnEntry, utána pedig az OnExit. Ez esetben naplázás történik. Gondoljunk bele, mennyivel egyszerűbb így a naplázás kezelése!

A szövés automatikusan történik, amikor lefordítjuk a projektet. A PostSharp rendszer a köztes kódba fordítja bele az aspektusainkat, így az ilyen kód nyomkövetése nehéz.

Végezetül, vegyük észre, hogy itt is a Separation of Concerns elvet használtuk. A naplázás felelősségeit és a naplózással bepiszkított metódusokat szétvágtuk naplózó aspektusra és POCO osztályokra, amiket később majd szövéssel rak össze a PostSharp.

## 6. Rendszerfejlesztés technológiája

Ez a fejezet „A rendszerfejlesztés technológiája” című tárgyhoz kapcsolódik szervesen. A fejezet két nagy részre tagolható: a rendszerszervezés című tárgyból megismert dokumentumok elkészítéséhez szükséges technikák ismertetésére, illetve a rendszerfejlesztés során használt segédeszközök bemutatására.

### 6.1. Követelményspecifikáció

A követelményspecifikáció (angolul: requirement specification) helyét a szoftverfejlesztés életciklusában megismerhettük a Rendszerszervezés című fejezetből. Itt az elkészítéséhez szükséges technikákat ismertetjük.

A követelményspecifikáció alapjai a megrendelővel készített riportok. Főbb részei:

- Jelenlegi helyzet leírása.
- Vágylomrendszer leírása.
- A rendszerre vonatkozó pályázat, törvények, rendeletek, szabványok és ajánlások felsorolása.
- Jelenlegi üzleti folyamatok modellje.
- Igényelt üzleti folyamatok modellje.
- Követelménylista.
- Irányított és szabad szöveges riportok szövege.
- Fogalomszótár.

A fenti listából csak a **követelménylista** kötelező, de a többi segíti ennek megértését. Egy példa követelménylista található a mellékletben. Nagyon fontos megérteni, hogy nem minden követelmény kerül bele a program következő verziójába. Erről az ütemterv leírásában szólunk részletesen.

#### 6.1.1. Szabad riport

A szabad szöveges riport egy-két óránál ne legyen hosszabb, különben sok tévedés lesz a szövegben. Óránként tartsunk szünetet. Az elkészült szöveget a végén érdemes felolvasni. A véglegesített szöveget el kell küldeni a megrendelőnek jegyzőkönyv formájában. A jegyzőkönyvet tanácsos eltárolni a projekt verziókövető rendszerébe. A mellékletben megtalálható egy jegyzőkönyvsablon.

A jegyzőkönyveket gyakran emlékeztetőnek is hívjuk. Ez utal a jegyzőkönyv legfontosabb feladatára, arra, hogy a feleket emlékeztesse, mit beszéltek meg. Erre főleg a vitás esetek kezelésénél van szükség.

A szabad szöveges riport elején egy kérdést teszünk fel: Hogyan működjön az új rendszer, hogyan működjön együtt a meglévő rendszerekkel? Az erre adott választ kell lehetőleg szó szerint leírni. Csak akkor szabad kérdést közbeszúrni, ha a megrendelő:

- önmagával ellentmondásba keveredik, vagy
- egy fogalom nem világos, vagy
- az elmondottak nem fednek le minden estet.

Ahhoz, hogy felfedezzük, hogy az elmondott rendszer lyukas, azaz nem minden esetet fed le, szükség van rendszerszemléletre. minden programozónak van algoritmikus gondolkozása és szerencsére sokuknak van rendszerszemlélete. Belőlük lehet rendszerszervező.

A szabad szöveges leírás a későbbi elemzés során nagyon hasznos lesz. A szövegben lévő főnevek a rendszer lehetséges szereplői (angol: actor), illetve osztályai; míg az igék, különösen a műveltető igék, a rendszer lehetséges használati esetei (angolul: use case), illetve metódusai.

Egy-két szabad szöveges riport után következik az irányított riport. Erre általában egy egyoldalas kérdőívvel készülnk. Lehet hosszabb is a kérdőív, de tapasztalat szerint egy egyoldalas kérdőív esetén is egy-két óra kell az irányított riporthoz.

#### 6.1.2. Irányított riport

Az irányított riport feladata a követelmények számszerűsítése (Pontosan hány alkalmazott van a cégnél?), a nemfunkcionális igények felderítése (Milyen gyors legyen a rendszer?), illetve a megrendelő figyelmének felhívása a szokásos megoldásokra (Legyen jelentéskészítő alrendszer?). A mellékletben megtalálható néhány példakérdőív.

Kétféle irányított riportról beszélhetünk:

- tisztázó riport,
- alrendszeriport.

A tisztázó riport, ahogy a nevében is benne van, tisztázó kérdéseket tartalmaz. Ezek a kérdések általában egy fogalom vagy folyamat felderítésére szolgálnak.

Az alrendszeriport valamely alrendszer követelményeinek feltárására használatos. Általában ezek a riportok változtatás nélkül újrafelhasználhatóak. Ha sikerül általánosan megírnunk egy alrendszert, ami könnyen újrafelhasználható, akkor érdemes elkészíteni a hozzá tartozó alrendszeriportot is. Néhány lehetséges alrendszer:

- riportozó alrendszer,
- jogosultságkezelő alrendszer,
- felhasználókezelő alrendszer.

Az igények felmérésénél tudnunk kell, hogy a megrendelő szervezetén belül más és más igények vannak, gyakran belső ellentéktől terhes a megrendelő. Ennek egyik fő oka, hogy az informatikai rendszerek bevezetése változást jelent a megrendelő életében, ami lehet, hogy néhány munkatársnak nem érdeke. Általában a szoftverfejlesztő céghöz akkor jut el egy igény, ha a megrendelő legfelső szervezeti szintjén döntés születik a változások szükségszerűségéről. Ugyanakkor a rendszert nem az igazgatók, hanem a munkatársak fogják használni. Gyakran e két szint között is félreértések vannak, nem megfelelő a kommunikáció, ellenérdekek lehetnek, talán egymás fogalomrendszerét se értik. Ezért fontos, hogy a rendszert a jövőben használatba vevő munkatársakhoz is jutassunk el kérdőíveket.

#### 6.1.3. Követelménylista

A követelménylista nagyon fontos, hiszen ezeket fogjuk megvalósítani. Nagyon fontos, hogy ne tartalmazzon félreértést. Ha nincs félreértés, akkor elvileg ez alapján tökéletesen lefejleszthető az igényelt informatikai rendszer. Másik probléma, hogy hiányos lehet a követelménylista és a hiányzó követelmények csak menet közben derülnek ki. Ez iteratív módszertanok esetén nem okoz problémát. Harmadik gond lehet, hogy a fejlesztés során változnak az igények. Ezt az agilis módszertanok oldják meg.

A követelmények a szabad szöveges riportban általában a következő nyelvi panelek kíséretében szerepelnek:

- Fontos / szükséges / jó lenne, ha ...
- Legyen / legyenek ...
- Kínáljon / nyújtson / tegye lehetővé ...
- Elvárás / igény / követelmény ...
- Tőmondatok.
- Felsorolások.

Ezen túl érdemes irányított riportokban rákérdezni az elvárásokra. Néhány példakérdés:

- Mik a rendszer fontos tulajdonágai?
- Kérem, mondja el forgatókönyvszerűen, hogyan szeretné használni a rendszert a belépéstől a kilépésig!
- Milyen kivételes helyzetekre kell felkészülni?

A követelménylistában általában ezek az oszlopok szerepelnek:

- Modul: Nem kötelező. Itt adhatjuk meg, hogy a követelmény melyik nagy modulhoz / komponenshez tartozik. Ennek segítségével már a tervezés legelején komponensekre bonthatjuk a rendszert.
- ID: Kötelező. Ezzel a rövid azonosítóval hivatkozhatunk a követelményre minden dokumentációban, megjegyzésben. Általában egy sorszám, pl.: K1, K2, ... .
- Név: Kötelező. A követelmény 2-3 szavas megfogalmazása. Önmagában nem feltétlenül értelmes, de beszédes.
- V.: Nem kötelező. Annak a verziónak a száma, ahol szeretnénk, ha már ez a követelmény ki lenne fejtve. A 0.1. verzió a legkisebb. Ezt az architekturális követelményeknek tartjuk fenn. Az 1.0. verzió, illetve ez alattiak a reális elvárások, az e felettiek vágyáлом, vagy reálisan csak későbbi verzióban elérhető követelmények. Az itt megadott verziók az ütemtervben még felülbírálhatók a prioritás segítségével.
- Kifejtés: Kötelező. A követelmény 2-3 mondatos, esetleg ennél is hosszabb kifejtése. Önállóan is értelmes. Megjegyzésekkel tartalmazhat a programozóknak, tesztelőknek vagy más szerepkörű olvasóknak.

Az ennek megfelelő követelménylista-sablon a következő:

Modul	ID	Név	V.	Kifejtés

Egy példa-követelménylista a mellékletekben található.

#### 6.1.4. Félreértések elkerülését szolgáló módszerek

Ha egyszer adottak a követelmények, akkor a módszertanok többé-kevésbé biztosítják, hogy a szoftvercégek ezeknek megfelelő szoftvert készítsenek. Tehát egy félreértés a követelménylistában

olyan szoftverhez vezethet, amire nincs is szüksége a megrendelőnek. Klasszikus példa erre a bicikli – autó példa:

Megrendelő: Szeretnék egy járművet, amivel reggelente eljuthatok a munkahelyemre.

Szoftvercég: Mik a jármű fontos tulajdonágai?

Megrendelő: Gyorsan eljussak a munkahelyemre. Kényelmes legyen.

Erre a szoftvercég leszállít egy biciklit egy kényelmes üléssel.

Megrendelő: De ez nem véd meg az esőtől! Ezt így nem veszem át!

Erre a szoftvercég a bicikli kormányára egy esernyőt erősít.

Megrendelő: De ezen nincs hely a táskámnak! Ezt így nem veszem át!

Erre a szoftvercég kosarat tesz az elejére.

A megrendelő ezt sem veszi át, hanem körülnéz a járműpiacon, és vesz egy autót!

A példa arról szól, hogy a megrendelő lelti szemei előtt nyilván már első alkalommal is egy autó lebegett, de fontos követelményeket (esőben is használható legyen, legyen hely a táskának, menő legyen, lehessen vele vidékre is utazni stb.) nem árult el. Vajon miért? Mert magától értetődő volt neki, hogy senki sem akar megázni az esőben munkába menet, és a táskának is kell hely. Végül is mondta, hogy kényelmes legyen. Tudjuk, hogy a riportok alatt rá kell kérdezni az ismeretlen fogalmakra. Miért nem kérdezett hát rá a rendszerszervező, hogy mit jelent az, hogy kényelmes? Mert ez neki ismert fogalom volt, csak másról értett alatt. Ez nagyon veszélyes! Nagyon sok egyszerű fogalom (ember, autó, kényelmes) másról és másról jelenthet a megrendelőnek és a szoftvercégnek.

A követelményspecifikáció nehézsége, hogy a megrendelő a számára evidens követelményeket nem mondja el. Ennek a problémának több ismert megoldása is van:

- Használati esetek (angolul: use cases) gyűjtése: Használatmód pontos leírása megszemélyesített életképpel.
- Szerepkör felvétele.
- Prototípuskészítés.
- Mutasson egy hasonlót.

Az első megoldás semmibe sem kerül, csak néhány plusz megbeszélésbe. Azt kell kérnünk a megrendelőtől, hogy nagyon részletesen, és ami nagyon fontos, a valós szereplők megnevezésével írja le, hogyan képzelik használni a rendszert. Itt kérdezzük rá azokra az esetekre, amik az üzleti folyamatok ismeretében bekövetkezhetnek. Ha úgy érezzük, hogy lyukas a leírás, azaz valamely esetre nem tér ki, arra rá kell kérdezni. Mivel a valós szereplők is szóba kerülnek, kiderülhetnek olyan megszorítások is, amik teljesen meglepőek lehetnek. Például a portás diszlexiás, így nem lehet szöveges felületű a beviteli ablak. Itt csak annyi a trükk, hogy nem elégünk meg egy nyúlfarknyi követelményspecifikációval, hanem egy sokkal részletesebbet szeretnénk. minden olyan technika, ami ez irányba hat, hasonlóan jó fokú.

A szerepkörfelvétel sokkal nehezebb megoldás. A megrendelő szervezetébe kell bekerülnünk, ott azt a munkát végezni néhány napig, amire a rendszert szeretnék használni. Ezzel a módszerrel sok rejtejt követelményre jöhettünk rá, és nem mellékesen megismerhetjük későbbi végfelhasználóinkat, ami nagyban csökkenti az ellenállásukat az új rendszerrel szemben.

A prototípuskészítés a legdrágább. Ekkor egy teljes felhasználói felülettel, de csak elnagyolt funkcionalitással rendelkező prototípust mutatunk be a megrendelőnek, hogy ilyenre gondolt-e. Ha igen, akkor a felhasználói felület kész is, így nincs elveszett munka, de ez nagyon valószínűtlen. Sokkal valószínűbb, hogy a felhasználói felület nem fog tetszeni. A megrendelő általában fontos információkat mond el a fejében lévő ideális megoldásról, miközben megindokolja, miért nem jó a felület. Ami ennél is fontosabb, kiderül, hogy félreértektük-e a megrendelő követelményeit. E nélkül általában csak az implementáció után derül fény a félreértesekre. Néhány módszertan, például az extrém programozás, előírja a prototípus készítését.

Az utolsó megoldás nagyon természetes. Általában az új igények abból fakadnak, hogy a megrendelő látott egy olyan megoldást, amihez hasonlót ő is szeretne. Ez weblapok esetén nagyon gyakori. Ha a riportok során mégse veti fel a megrendelő, hogy az általa elképzelt szoftverhez, informatikai rendszerhez van hasonló, akkor kérjük meg, hogy soroljon fel általa ismert hasonló megoldásokat. Mondja el, mi az, amit ezekben kedvel és mi az, ami szerinte ezekben rossz. Ez nem kerül semmibe és sok félreérts elkerülhető a segítségével.

#### 6.1.5. Üzleti modellezés fogalmai

Jól látható, hogy a követelményspecifikáció fontos része az üzleti folyamatok megértése. Az üzleti modellt leírhatjuk UML nyelven, illetve a legújabb irányzat szerint BPMN (angolul: Business Process Modeling Notation) nyelven, ami egy viszonylag új OMG szabvány.

Az üzleti folyamatok modellezése során a következő fogalmakat használjuk:

**Üzleti folyamat (angolul: Business Use Case):** Egy helyen és egy időben végrehajtott folyamat, amely valamely üzleti szereplő vagy üzleti munkatárs számára értéket termel. Üzleti folyamat például a vásárlás: Az ügyfél bemegy az üzletbe, odamegy a pulthoz, kéri az árut, kap egy cetlit, amivel elmegy fizetni a pénztárhoz, utána visszamegy és megkapja a kifizetett árut.

Üzleti folyamatokat nemcsak külső szereplők, hanem belső szereplők is használhatnak, azaz minden üzleti folyamatot fel kell venni, amit a szereplők az üzleti folyamat során végeznek.

**Üzletifolyamat-diagram (angolul: Business Use Case Diagram):** Az üzletifolyamat-diagramok megmutatják, hogy mely üzleti szereplő mely üzleti folyamatokat használhatja. A vásárlás példánál maradva, a „Vásárló” használhatja az „Áruvásárlás” folyamatot.

Az üzletifolyamat-diagram megmutatja a szereplők és az üzleti folyamatok kapcsolatait, valamint az üzleti folyamatok egymás közötti kapcsolatait. Egy szereplő és üzleti folyamat között akkor van kapcsolat, ha a szereplő kezdeményezheti az üzleti folyamatot. Két üzleti folyamat között kétféle kapcsolat lehet:

- extend
- include

Extend kapcsolatról akkor beszélünk az A és a B üzleti folyamat között, ha az A üzleti folyamat során bekövetkezik egy bizonyos esemény, akkor a B üzleti folyamat lesz végrehajtva.

Include kapcsolatról akkor beszélünk az A és a B üzleti folyamat között, ha az A üzleti folyamatnak része a B üzleti folyamat.

**Üzleti folyamat lefutása (angolul: Business Use Case Realization):** Egy üzleti folyamat lefutása megmutatja az üzleti folyamat

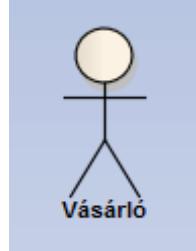
- üzleti tevékenységeinek,
- üzleti szereplőinek és
- az üzleti entitások kapcsolatát.

Az üzleti folyamatok lefutását aktivitási diagramokkal (angolul: activity diagramok) mutatjuk meg. Az aktivitási diagramokat úgynevezett partíciókra (angolul: swimlane) osztjuk. minden partícióhoz rendelünk egy üzleti szereplőt. Az egyes partíciókba a megadott üzleti szereplő tevékenységeit vesszük fel.

Az üzleti folyamat lefutásokba két tevékenység közé be kell szúrni egy entitást, ha az első tevékenység módosítja azt.

Az üzleti folyamat lefutásokat a hozzá tartozó üzleti folyamat alá kell felvenni a Business Use Case Modell package-ben.

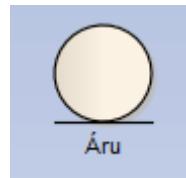
**Üzleti szereplő (angolul: Business Actor):** Azokat a külső szereplőket azonosítja, amelyek a szervezeten kívül vannak és interakcióba lépnek a szervezet üzleti folyamataival. Például a „Vásárló”, ennek UML jelölése:



26. ábra: Egy üzleti szereplő UML jelölése

**Üzleti munkatárs (angolul: Business Worker):** A szervezetbeli szerepköröket azonosítja. A vásárlás üzleti folyamatban egy példa egy szerepkörre a pénztáros.

**Üzleti entitások (angolul: Business Entity):** Azok a dolgok, amelyeket az üzleti munkatársak használnak az üzleti folyamatok során. A vásárlás üzleti folyamatban egy példa az üzleti entitásra az áru fogalma. Ennek UML ábrája:



27. ábra: Egy üzleti entitás UML jelölése

**Üzletifolyamat-modell (angolul: Business Use Case Model):** Azt mutatja meg, hogy külső szereplők (partnerek és ügyfelek) milyen, a megrendelő által nyújtott funkciókat használhatnak. Az üzletifolyamat-modell tartalmazza:

- az üzleti folyamatokat,
- az üzleti szereplőket,
- az üzleti entitásokat,
- az üzleti folyamat lefutásokat,

továbbá megmutatja azt is, hogy melyik üzleti szereplő milyen üzleti folyamatokat kezdeményez és milyen üzleti folyamatokban vesz részt.

#### 6.1.6. [Üzleti folyamatok modellezése](#)

Az üzleti folyamatok modellezésének célja, hogy megismerjük a megrendelő szervezetét, az üzleti folyamatait, illetve meghatározzuk azokat a folyamatokat, amelyeket a megrendelt informatikai rendszer támogatni fog. Ehhez meg kell ismernünk a megrendelő szakkifejezéseit is, amiből fogalomszótárt szoktunk építeni. A modellezés során a következő tevékenységeket szoktuk elvégezni:

- üzleti szereplő és munkatárs elemzése,
- üzleti folyamat elemzése,
- üzleti entitás elemzése,
- üzleti folyamatok modellezése.

##### 6.1.6.1. [Üzleti szereplő és munkatárs elemzése](#)

Az üzleti szereplő és munkatárs elemzése során a következő termékeket kell előállítani:

- Üzleti szereplő (angolul: Business Actor)
- Üzleti munkatárs (angolul: Business Worker)

Üzleti szereplők azok a személyek vagy külső rendszerek, amelyek interakcióba lépnek az üzleti folyamatokkal. Üzleti munkatársnak nevezzük azokat a szerepköröket, amelyek a szervezeten belül lépnek kapcsolatba az üzleti folyamatokkal.

A riportokban előforduló főnevek, főleg a személynevek, lehetnek a külső és belső üzleti szereplők. A leggyakrabban előforduló üzleti szereplők (azaz külső szerepkörök):

- Partner,
- Vásárló,
- Ellenőrző szerv.

Általában a partnernek is több fajtája van:

- Beszállító,
- Értékesítő,
- Megbízott, azaz a kiszervezett folyamatokat (pl. weblapkészítés, könyvelés) ellátó külső cég.

A vásárlókat és a partnereket is szokás fontosságuk szerint megkülönböztetni:

- Kiemelt partner,
- Időszakos partner,
- Törzsvásárló,
- Visszatérő vásárló,
- Alkalmi vásárló,
- VIP személy.

A fenti felsorolás biztosan nem teljes, de rámutat, hogy milyen fontos tisztázni, hogy a külső szereplőknek vannak-e tulajdonságai.

Ellenőrző szervből sokféle van. Általában a megrendelő cég ágazatát szabályozó törvények és rendeletek írják elő, hogy milyen ellenőrző szerv felé milyen adatokat kell nyújtani, amit az információs rendszer automatizálhat.

Ha a fenti szerepkörök semelyike sem jön elő a riportok során, akkor érdemes rákérdezni, hogy pl. az ellenőrző szervek felé kell-e adatot nyújtania a megrendelő cégnek.

Az üzleti munkatársakat (azaz a belső szerepköröket) úgy tudjuk legegyszerűbben felderíteni, ha a riportok során rákérdezünk azoknak a beosztására, akik a megrendelt rendszert használni szokták. Néhány lehetséges üzleti munkatárs:

- Eladó,
- Pénztáros,
- Raktáros,
- Felügyelő,
- HR munkatárs,
- Vezető.

#### 6.1.6.2. Üzleti folyamat elemzése

Az üzleti folyamat elemzése során a következő termékeket kell előállítani:

- Üzleti folyamatok (angolul: Business Use Case)
- Üzletifolyamat-diagramok (angolul: Business Use Case Diagram)

Ahhoz, hogy megtaláljuk az üzleti folyamatokat, meg kell nézni, hogy melyik szereplőnek milyen értéket termel, milyen szolgáltatásokat ad az üzlet. A szabad riportokban található igék, főleg a műveltető igék, lehetnek üzleti folyamatok. Ha ezek értéket teremtenek, akár a külső, akár a belső szereplőknek, akkor biztosan üzleti folyamatok. Gyakori üzleti folyamatok:

- Vásárlás,
- Szolgáltatás lefoglalása,
- Megrendelés,
- Reklámozás,
- Nyersanyagszállítás,
- Értékesítés,
- Szolgáltatás,

- Weblapkészítés,
- Könyvelés,
- Üzemeltetés.

Ha már a külső és a belső szerepköröket tisztáztuk, akkor érdemes rákérdezni, hogy az egyes szereplők milyen üzleti tevékenységeket végezhetnek. Ezeket az igék elemzésével általában sikerül jól lefedni, de gyakran kimarad egy-két kevésbé jellemző, de a rendszer teljességéhez hozzáartozó tevékenység. Ezen túl érdemes feltenni a következő kérdéseket is valamely irányított riportban:

- Milyen szolgáltatásokat nyújt a cég a külső partnereinek, vagy azok neki?
- Ezeket milyen belső tevékenységgel támogatják a cég munkatársai?

#### 6.1.6.3. [Üzleti entitás elemzése](#)

Az üzleti entitás elemzése során a következő termékeket kell előállítani:

- Üzleti entitások (angolul: Business Entity)

Az üzleti entitás elemzés során meg kell határozni azokat a „dolgokat”, amelyeken az üzleti folyamatok valamilyen tevékenységet végeznek. A szabad riportban előforduló főnevek, főleg a gyűjtőnevek, lehetnek üzleti entitások (üzleti objektumok). Ha ezek a műveltető igék tárgyai, és a műveltető igéből üzleti folyamat lett, akkor nagyon valószínű, hogy üzleti entitást találtunk. Gyakori üzleti entitások:

- Áru,
- Raktár,
- Pénz,
- Iroda,
- Üzlethelység,
- Weboldal,
- Reklám.

A fenti listán a pénz csak akkor üzleti entitás, ha a pénzmozgás üzleti folyamat, például bankok esetében. Az üzleti entitásokat úgy is megtalálhatjuk, ha rákérdezünk, hogy milyen kézzelfogható változást eredményez egy-egy üzleti folyamat.

#### 6.1.6.4. [Üzleti folyamatok modellezése](#)

Az üzleti folyamatok modellezése során a következő termékeket kell előállítani:

- Üzleti folyamat lefutása (angolul: Business Use Case Realization)
- Üzletifolyamat-modell (angolul: Business Use Case Model)

Az üzletifolyamat-modell előállítása egyszerű feladat, hiszen ez az eddig elkészített részek összessége. Itt csak azt kell átgondolni, hogy vannak-e logikailag különálló részei az üzleti folyamatoknak. Ha igen, akkor ezeket érdemes külön csomagba (package) szervezni.

Az üzleti folyamat lefutása egy folyamatábra (illetve UML esetén egy aktivitási diagram), amely megmutatja, hogy a hozzá tartozó üzleti folyamat hogyan valósul meg. Itt használjuk fel az üzleti

entitásokat. Ha két lépés között felhasználunk egy üzleti entitást, akkor azt itt jelezzük. A lépések egyébként egyszerű tevékenységek sorozataként szoktuk leírni.

## 6.2. Funkcionális specifikáció

A funkcionális specifikáció (angolul: functional specification) a felhasználó szemszögéből írja le a rendszert. A követelményspecifikációból ismerjük az elkészítendő rendszer követelményeit, üzleti folyamatait. Ezeket kell átalakítanunk funkciókká, azaz menükké, gombokká, lenyíló listákká. A funkcionális specifikáció központi eleme a használati eset (angolul: use case). A használati esetek olyan egyszerű ábrák, amelyeket a megrendelő könnyen megért mindenféle informatikai előképzettség nélkül. A funkcionális specifikáció fontosabb részei:

- A rendszer céljai és nem céljai.
- Jelenlegi helyzet leírása.
- Vágylomrendszer leírása.
- A rendszerre vonatkozó külső megszorítások: pályázat, törvények, rendeletek, szabványok és ajánlások felsorolása.
- Jelenlegi üzleti folyamatok modellje.
- Igényelt üzleti folyamatok modellje.
- Követelménylista.
- Használati esetek.
- Megfeleltetés, hogyan fedik le a használati esetek a követelményeket.
- Képernyőtervezek.
- Forgatókönyvek.
- Funkció – követelmény megfeleltetés.
- Fogalomszótár.

Látható, hogy a követelményspecifikációhoz képest sok ismétlődő fejezet van. Ezeket nem fontos átemelni, elég csak hivatkozni rájuk. Az egyes módszertanok eltérnek abban, hogy mely fejezeteket és milyen mélységen kell elkészíteni. Általában elmondható, hogy a modern módszertanok használatieset-központúak.

A funkcionális specifikáció fontos része az úgynevezett megfeleltetés (angolul: traceability), ami megmutatja, hogy a követelményspecifikációban felsorolt minden követelményhez van-e azt megvalósító funkció.

### 6.2.1. Követelményelemzés

A követelményelemzés célja a rendszer behatárolása, a funkcionális és a nemfunkcionális követelmények meghatározása, ezek egyeztetése a megrendelővel. Azaz:

- Egyetértésre jutni a megrendelővel, hogy mit csináljon a rendszer.
- Felhasználói felületek meghatározása.
- Felvázolni a rendszer funkcionális működését.
- Meghatározni a rendszer használati eseteit.

### 6.2.2. Használati esetek

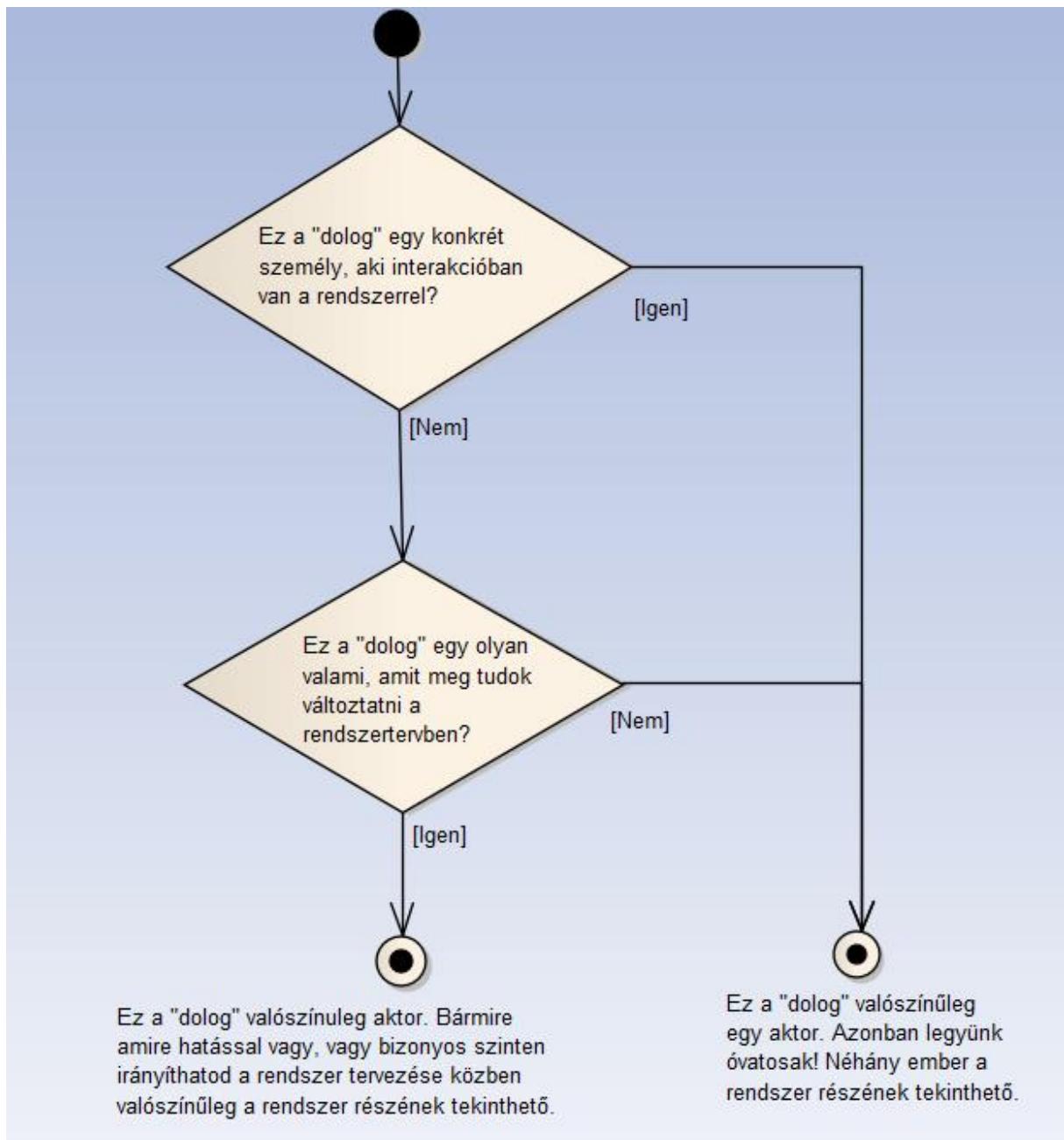
A használati eset (angolul: use case) a rendszer és a felhasználók közötti kommunikációt írja le. Olyan eseménysor, amely során az aktor számára érték, például információ, keletkezik. A rendszer belső működési részlete (pl. adattovábbítás) nem használati eset. Lehet olyan használati eset is, amelyet időzítő indít.

Egy használatieset-ábra részei:

- rendszerhatár,
- szereplő (aktor),
- használati eset.

A használati eset a rendszer egy funkciója. Ezek általában a rendszer határain belül vannak. Lehetnek kívül is, ha nem akarjuk megvalósítani ezt a funkciót. A szereplők vagy felhasználók (angolul: actor) a funkciókat használó emberek és egyéb informatikai rendszerek. A felhasználók a rendszer határain kívül esnek. Azok az informatikai rendszerek, amelyek nem részei a tervezett rendszernek, ennek a rendszernek a szemszögéből felhasználók. Szokás a rendszer adatbázisát, dokumentumtárát, azaz a statikus vetületét is felhasználóként kezelni.

A következő folyamatábra segít eldönten, hogy a szabad szöveges riport szövegében talált főnév aktor vagy sem.



28. ábra: „Aktor vagy sem” folyamatábra

A használatieset-ábrákon egyszerűen összekötjük azt a felhasználót a funkcióval, aki használja. Ezeket a bal oldalon helyezzük el. Jobb oldalra kerülnek azok a rendszerek (pl. adatbázis), amire hatással van a funkció.

A használati esetek közt UML jelölésrendszert használva többféle kapcsolat (pl. include, extend) is kialakítható, de ez inkább kerülendő, mert rontja az ábra érthetőségét. A logikai rendszertervben már használhatók ezek a jelölések is, hiszen az már a programozóknak szóló dokumentum.

#### 6.2.2.1. Szereplők elemzése

Szereplő (angolul: actor) minden, ami adatot cserél a rendszerrel. Egy szereplő lehet felhasználó, külső hardware, más rendszer. Egy felhasználó és egy szereplő nem ugyanaz, mert egy felhasználónak (aki egy ember), több szerepköre is lehet. A szereplőnek egy szerepköre van.

Hogyan találjuk meg a szereplőket? A következő kérdéseket tegyük fel magunknak:

- Ki vagy mi kerül kapcsolatba a rendszerrel?
- Ki fogja szolgáltatni, használni, törölni az információkat?
- Ki fogja használni a megfelelő funkcionálitást?
- Ki érdekelt egy bizonyos követelményben?
- Hol használják a szervezetben a rendszert?
- Ki/mi fogja támogatni és karbantartani a rendszert?
- Melyek a külső erőforrásai a rendszernek?
- Milyen külső rendszerekkel fog kommunikálni a rendszer?

Szereplők rendszerezése:

- Elsődleges szereplők (akiknek a rendszer készül)
- Másodlagos szereplők (pl. rendszer-adminisztrátor)
- Külső hardware
- Más külső rendszerek

Aktor leírása:

- mit vár a rendszertől
- felelősségi köre
- fizikai környezete
- hány aktorpéldányt képvisel
- milyen gyakran használja a rendszert
- van-e számítástechnikai tapasztalata?

#### 6.2.2.2. Használati eset tervezése

A használati esetek megtalálásához tegyük fel a következő kérdéseket:

- Mit várnak a szereplők a rendszertől?
- Fog-e adatot bevinni a rendszerbe, módosítani vagy törölni?
- Kell-e informálni a rendszerben bekövetkezett bizonyos eseményekről?
- Használ-e egyéb nem elsődleges funkciót?
- El fogja-e indítani, illetve le fogja-e állítani a rendszert?
- Végez-e karbantartást?
- Külső, hirtelen bekövetkezett eseményről kell-e a szereplőt tájékoztatni?
- Melyek azok a funkciók, amelyek módosítják a rendszer működését?

A feltárt használati eseteket össze is vonhatjuk. Több összefüggő folyamatot egy használati esetként kezelünk, ha a későbbiekben egységesen akarjuk ellenőrizni, jóváhagyatni, tesztelni, dokumentálni. Több összefüggő használati esetet akkor is egybe vonunk, ha hasonló a lefutásuk.

Használati eset leírása:

- *Leírás*: „ki” „mit” csinál a rendszer használati eset során.
- *Előfeltételek*: Azon korlátok, szükséges feltételek meghatározása, amelyek nélkül a használati eset nem indítható. Opcionális.
- *Utófeltételek*: A használatieset-lefutása milyen állapotban hagyja a rendszert. Olyan állapotokat kell ide írni, amelyek nem adatkör-specifikusak. Itt is fontos, hogy az informatikai rendszerre értelmezhető állapotokat határozzunk meg. Opcionális.
- *Egyéb*: nemfunkcionális követelményeket írjuk ide.

#### 6.2.2.3. Használatieset-diagramok

A használatieset-diagramok megmutatják azt, hogy melyik szereplő milyen használati eseteket kezdeményezhet. A diagramokon a szereplőket összekötjük a használati esetekkel, így jelezve azt, hogy az adott szereplő kezdeményezheti a használati esetet. A használati esetek között *include* és *extends* kapcsolatokat is megmutatunk.

#### 6.2.2.4. Használatieset-lefutások

A bonyolultabb használati esetekhez használatieset-lefutást kell tervezni. A használatieset-lefutást aktivitási diagramokkal (angolul: activity diagram) lehet megmutatni.

Az aktivitási diagramokat partíciókra (angolul: swimlane) osztjuk, és minden partícióra helyezünk egy szereplőt. Az aktivitási diagramokon a rendszer és a szereplő között lezajló kapcsolódó tranzakciók sorozatát mutatjuk meg. A használati eset végén a szereplő számára értéket termel a rendszer.

A rendszert nem funkcionális alrendszerekre, hanem szerepkörök mentén bontjuk alrendszerekre. Ezeket a szerepköröket helyezzük el az aktivitási diagramok partícióra és megmutatjuk az általuk végzett tevékenységeket az adott használati esetben.

A rendszer használati esetek dokumentációjára a lépésenkénti (angolul: step-by-step) módszert alkalmazzuk. Ez abból áll, hogy a diagram dokumentációjában leírjuk lépésről lépésre a tevékenységeket, amelyeket felvettünk a diagramon egy-egy hozzájuk tartozó rövid magyarázattal.

#### 6.2.3. Képernyőtervez

A képernyőtervez mutatják meg, hogy mely funkciók kerülnek egymás mellé, melyik képernyőről mely képernyőre juthatunk. A grafikus felhasználói felület (angolul: graphical user interface, röviden: GUI) tervezésnek ismert néhány szabálya, amelyeket itt csak felületesen ismertetünk. A következő elveket (kvíve az első két elvet) Ian Sommerville Szoftverrendszer fejlesztése című könyvből idézzük:

- **Teljes**: A felhasználói felületen keresztül a program minden funkciója elérhető legyen.
- **Átlátható**: A felhasználói felület legyen szellős, jól különüljenek el egymástól az egyes funkciók, funkciócsoporthoz.
- **Felhasználói jártasság**: A felületnek olyan kifejezéseket kell használnia, amelyek megfelelnek a rendszert legtöbbet használók tapasztalatainak.

- Konzisztencia: A felületnek konzisztensnek kell lennie, azaz lehetőség szerint hasonló műveleteket hasonló módon kell realizálnia.
- Minimális meglepetés: A rendszer soha ne okozzon meglepetést a felhasználóknak.
- Visszaállíthatóság: A felületnek rendelkeznie kell olyan mechanizmusokkal, amelyek lehetővé teszik a felhasználók számára a hiba után történő visszaállítást.
- Felhasználói útmutatás: A felületnek hiba bekövetkezése esetén értelmes visszacsatolást kell biztosítania, és környezetérzékeny felhasználói súgóval is rendelkeznie kell.
- Felhasználói sokféleség: A felületnek megfelelő interakciós lehetőségekkel kell rendelkeznie a rendszer különféle felhasználói számára.

Felhasználói hibákat okozhatunk, ha nem vesszük figyelembe a valódi felhasználók képességeit és munkakörnyezetét. Rossz felhasználói felület esetén a felhasználó úgy érezheti, hogy a használt szoftver gátolja őt annak a célnak az elérésében, amiért is azt használja.

#### 6.2.4. [Forgatókönyvek](#)

A forgatókönyvek a rendszer egy-egy tipikus felhasználását mutatják be. A forgatókönyvnek általában van egy célja, például egy iktató rendszerben: Levél érkeztetése, címzett értesítése, dokumentumárba helyezés. A forgatókönyv bemutatja, milyen funkciókat kell használni, milyen sorrendben a kívánt cél elérése érdekében. Ilyen értelemben egy telepítési útmutatóhoz hasonlítanak.

A forgatókönyvek nagyon hasznosak, hogy a rendszerszervező újból végiggondolja, hogyan is fogja használni a felhasználó a rendszert. Ahogy végiggondolja, nagyobb eséllyel vesz észre lyukakat, ellentmondásokat a tervben, mintha csak használati eseteket készítene.

#### 6.2.5. [Olvasmányos dokumentum készítése](#)

A funkcionális specifikáció a felhasználó szemszögéből készül a felhasználóknak. Egyetlen gond vele, hogy a megrendelő nem olvassa el és így nem derül ki, hogy nem azt a rendszert fogjuk készíteni, amit ő szeretne. Ezért trükkökhöz kell folyamodnunk. A dokumentum legyen:

- olvasmányos,
- vicces,
- tagolt,
- történetszerű,
- veszélyekkel teli.

Az emberi agy fő feladata, hogy megvédjen minket a sarok mögött ólálkodó vérszemomás tigristől. Így ha arra akarjuk rávenni, hogy unalmas dokumentumokat értelmezzen, akkor hamar elkalandozik figyelmünk. Ugyanakkor tapasztalatból tudjuk, hogy vannak olyan könyvek, amiket szívesen olvasunk, pl. Rejtő Jenő könyveit. Tehát a feladatunk csak annyi, hogy olvasmányosan írunk. Ezt szépírótanfolyamokon tanítják. Ezt minden rendszerszervezőnek ajánljuk.

Szerencsére néhány egyszerű tanácsot betartva könnyen írhatunk könnyen olvasható dokumentumokat. Először is, legyünk viccesek, például használunk abszurd személyneveket, lehetőleg állandó jelzőkkel: „Amikor Cicus-Micus, a titkárnők gyöngye, rábök a levél érkeztetése menüre, ...”. Ez sokkal élvezetesebb agyunknak, mint ha egyszerűen ezt írnánk: „A levél érkeztetése menüre kattintva ...”.

Fontos, hogy a szöveg tagolt legyen. Egy teljesen teleírt oldal elrémisztő! Használunk rövid sorokat, amit szemünk át tud fogni. Használunk sok felsorolást, táblázatot, ábrát. Képeket érdemes elhelyezni, akkor is, ha nem közvetlenül kapcsolódik a témahoz, de a „mesébe” beleszóhető.

#### 6.2.6. Három olvasmányos példa

Az emberi agy könnyebben ért meg történeteket, mint bármilyen más. Ezért érdemes törekedni, hogy egy történetet meséljünk el, amibe beleszójük magát a dokumentum tartalmát. Hasonlítsuk össze a három szöveget:

„A levél feladása előtt a levelet iktatószámmal kell ellátni. Ezt a számot a kimenő levelek alrendszer generálja és iktatja. A dokumentumtárban a levélhez tartozó válaszok (illetve az ezekre adott válaszok) lekérdezhetők.”

„Cicus-Micus, a titkárnők gyöngye levelet ír. Hosszan gondolkodik a levél szövegén, végül csak ennyit ír: „Este 10-kor a Néma Bikában”. A levélhez iktatószámot generáltat a kimenő levelek alrendszerrel. A munkaidő vége felé megnézi, hogy jött-e válasz, és hogy arra válaszolt-e ő maga. Tudja, hogy a dokumentumtárban minden, az eredeti levélhez tartozó levél lekérdezhető.”

„Cicus-Micus, a titkárnők gyöngye levelet ír bátyjának, Erős Pistának, a lusták leglustábbikának, hogy megcsalta a férje: „Megcsalt! Hozd a vadászpuskád!”. A levélhez iktatószámot generáltat a kimenő levelek alrendszerrel. Nem hiába ő a titkárnők gyöngye. Bátyjának, a lusták leglustábbikának válasza csak ennyi: „Megyek! Vegyél patron!.” Cicus-Micus, a titkárnők gyöngye szerencsére nem értette, milyen patron, így vér nem folyik. Utólag könnyű volt megírni a történetet, mert a dokumentumtárban minden, az eredeti levélhez tartozó levél lekérdezhető.”

A három szöveg a rendszer leírásának szempontjából megegyezik. A különbség a mesei szálban van. Ez elsőben nincs semmi körítés, így unalmas, a megrendelő nem fogja elolvasni. Az utolsóban részletes a történet, túl mesés, így a komoly megrendelő bugytának fogja tartani. Megint csak nem olvassa el. A középső az arany középút, amikor van is történet, mögé lehet gondolni egy érdekes mesét, de azért nem vesz el a lényeg, a rendszer leírása. Vegyük észre, hogy a középsőben direkt rájátszunk a szóke titkárőr sztereotípiára (Ha válaszolt volna, arra emlékeznie kellene!), amiért hálás az agyunk, megerősítést nyer egy bevésődött minta, így szívesen olvassa a történetet.

Utolsó tanács, hogy a szövegünk legyen veszélyekkel teli. Ez segít ébren tartani figyelmünket. Agyunk szerint a környezetünk veszélyekkel terhes, minden pillanatban betoppanhat egy vérszemomjas tigris. Ha a szövegünk veszélyeket rejt, pl. a Néma Bika biztosan veszélyes egy hely, akkor agyunk nem fogja félvállról venni az olvasást és nem alszunk bele a szövegbe.

Olvasmányos dokumentumokról bővebben a Joel on Software blog „Fájdalommentes funkcionális specifikáció” című bejegyzése ír: <http://hungarian.joelonsoftware.com/PainlessSpecs/4.html>.

### 6.3. Ütemterv

A megrendelőnek küldjük el a kész specifikációt. Érdemes néhány megbeszélésen bemutatni a képernyőtervezetet, a forgatókönyveket. minden megbeszélésről készítsünk jegyzőkönyvet. Ha a funkcionális specifikációt elfogadta a megrendelő, akkor következik az árajánlat kialakítása.

Az árajánlat legfontosabb része az ütemterv. Az ütemterv határozza meg, hogy mely funkciók kerülnek be a rendszer következő verziójába és melyek maradnak ki. Egy példaütemterv a mellékletben található. Az ütemtervet készíthetjük például MS Project segítségével, de ez csak bonyolulttá teszi a választ az egyszerű kérdésre: Mennyi idő kell a rendszer kifejlesztéséhez? A Joel on Software blog alapján ajánljuk, hogy használjunk egy egyszerű Excel táblát az alábbi oszlopokkal:

- Funkció
- Feladat
- Prioritás
- Becslés
- Aktuális becslés
- Eltelt idő
- Hátralévő idő

Az egyes feladatokat bontsuk olyan apró részfeladatokra, amelyek elkészítése maximum 8 óra. Ezt írjuk a becslés oszlopba. Az aktuális becslés, az eltelt idő és a hátralévő idő oszlopok abban segítenek, hogy a jövőben pontosabb becslést tudjunk adni.

A funkciók kis részfeladatokra történő bontása azért nagyon hasznos, mert általános tapasztalat szerint minél kisebb egy feladat, annál pontosabban tudjuk megbecsülni. Ezen túl, ha egy funkciót részfeladatokra bontunk, akkor egyúttal végig is gondoljuk azt, és így könnyebben vehetünk észre tervezési hibákat. Fontos, hogy a becslést az a programozó végezze, aki a funkciót programozni fogja. Így pontos képet kaphatunk a feladat nagyságáról, illetve a programozó felelősséggel tartozik a saját becsléséért, ami jó motivációs tényező.

A prioritás adja meg, milyen fontos az adott funkció / feladat a rendszer működése szempontjából. Az 1-es prioritás elengedhetetlen, az 2-es nagyon hasznos funkció, a 3-mas kényelmi funkció. A 0-s prioritás jelentése, hogy az adott funkció már kész van. Természetesen ezek csak ajánlások.

Érdemes az ütemtervben tartaléket hagyni, hogy csúszás esetén legyen hova csúszni. Tapasztalatunk szerint minden szoftverprojekt csúszik. A kérdés csak az, lett-e elegendő idő bekalkulálva a csúszásra vagy sem. Természetesen ezt a tartalék időt nem akarja kifizetni a megrendelő, így vagy minden feladatot megszorzunk 1.2-vel, de ez elrontja a becslést, vagy olyan nevet adunk a csúszásnak, ami a megrendelőnek is elfogadható, pl.: projektvezetés.

Az ütemezésről részletesen olvashatunk a Joel on Software blog „Fájdalommentes szoftverütemezés” című bejegyzésében: <http://hungarian.joelonsoftware.com/Articles/PainlessSoftwareSchedules.html>.

### 6.3.1. Napidíj

A napidíj megállapításánál szokásos az a módszer, hogy kiszámoljuk a cégnk működési költségeit és a napidíjat úgy állítjuk be, hogy 50%-os kihasználtság esetén nullszaldós legyen a cégt. A napidíjnak fedeznie kell a programozó bérét, az egy programozóra jutó bérleti költségeket és rezsit, a menedzsment fizetését és egyéb járulékos költségeket.

Egy cégt általában nem egy, hanem több napidíjjal dolgozik. Általában legalább megkülönböztetjük a szenior (azaz tapasztalt) és a junior (azaz tapasztalatlan) programozók bérét. A napidíjat befolyásoló másik tényező a projekt hossza. Minél hosszabb a projekt, annál jobb lesz a programozónk

kihasználtsága, így kisebb óradíjat tudunk adni. Az alábbi kis lista egy közepes költségekkel és ismertséggel bíró (minél ismertebbek vagyunk, annál nagyobb napidíjjal dolgozhatunk) budapesti cég lehetséges napidíjait tartalmazza:

## Óradíjak projektméret alapján

A projekttapasztalatnál nem az éveket, hanem az effektíve projektben töltött nettó időt vesszük alapul.

**A táblázatban szereplő árak NETTÓ árak (2015. márciusi állapot)**

### 1 hónapnál rövidebb projekt esetén:

Beosztás	Óradíj	Elvárások
Junior tesztmérnök	32 000,00 Ft	3 évnél kevesebb projekttapasztalat
Junior programozó	35 200,00 Ft	3 évnél kevesebb projekttapasztalat
Szenior tesztmérnök	38 400,00 Ft	3-5 év projekttapasztalat, certified
Szenior programozó	44 800,00 Ft	3-5 év projekttapasztalat, certified
Vezető tesztmérnök	51 200,00 Ft	5+ QM területen szerzett tapasztalat, certified
Vezető fejlesztő	51 200,00 Ft	5-7 év között, speciális területre is certified
Szoftvertervező (softver architect)	64 000,00 Ft	7+ tapasztalat, certified, tervezői tapasztalat
Junior projekt menedzser	41 600,00 Ft	3 évnél kevesebb projektvezetői tapasztalat
Projekt menedzser	64 000,00 Ft	3 év feletti projektvezetői tapasztalat
Rendszergazda	32 000,00 Ft	Linux és Win tapasztalat 3+
PHP programozó	24 000,00 Ft	2+ év tapasztalat webes fejlesztésben

### 1-3 hónap közötti projekt esetén

Beosztás	Óradíj	Elvárások
Junior tesztmérnök	28 000,00 Ft	3 évnél kevesebb projekttapasztalat
Junior programozó	30 800,00 Ft	3 évnél kevesebb projekttapasztalat
Szenior tesztmérnök	33 600,00 Ft	3-5 év projekttapasztalat, certified
Szenior programozó	39 200,00 Ft	3-5 év projekttapasztalat, certified
Vezető tesztmérnök	44 800,00 Ft	5+ QM területen szerzett tapasztalat, certified
Vezető fejlesztő	44 800,00 Ft	5-7 év között, speciális területre is certified
Szoftvertervező (softver architect)	56 000,00 Ft	7+ tapasztalat, certified, tervezői tapasztalat
Junior projekt menedzser	36 400,00 Ft	3 évnél kevesebb projektvezetői tapasztalat
Projekt menedzser	56 000,00 Ft	3 év feletti projektvezetői tapasztalat
Rendszergazda	28 000,00 Ft	Linux és Win tapasztalat 3+
PHP programozó	21 000,00 Ft	2+ év tapasztalat webes fejlesztésben

### 3 hónapnál hosszabb projekt esetén

Beosztás	Óradíj	Elvárások
Junior tesztmérnök	24 000,00 Ft	3 évnél kevesebb projekttapasztalat
Junior programozó	26 400,00 Ft	3 évnél kevesebb projekttapasztalat
Szenior tesztmérnök	28 800,00 Ft	3-5 év projekttapasztalat, certified
Szenior programozó	33 600,00 Ft	3-5 év projekttapasztalat, certified
Vezető tesztmérnök	38 400,00 Ft	5+ QM területen szerzett tapasztalat, certified
Vezető fejlesztő	38 400,00 Ft	5-7 év között, speciális területre is certified
Szoftvertervező (softver architect)	48 000,00 Ft	7+ tapasztalat, certified, tervezői tapasztalat

Junior projekt menedzser	31 200,00 Ft	3 évnél kevesebb projektvezetői tapasztalat
Projekt menedzser	48 000,00 Ft	3 év feletti projektvezetői tapasztalat
Rendszergazda	24 000,00 Ft	Linux és Win tapasztalat 3+
PHP programozó	18 000,00 Ft	2+ év tapasztalat webes fejlesztésben

A fenti táblázatban ezek a munkakörök szerepeltek (a munkakör utáni szám a napidíj szorzója az elsőhöz képest):

- Junior tesztmérnök (1,0)
- Junior programozó (1,1)
- Szenior tesztmérnök (1,2)
- Szenior programozó (1,4)
- Vezető tesztmérnök (1,6)
- Vezető fejlesztő (1,6)
- Szoftvertervező (softver architect) (2,0)
- Junior projekt menedzser (1,3)
- Projekt menedzser (2,0)
- Rendszergazda (1,0)
- PHP programozó (0,75)

Lehet látni, hogy egy tesztmérnök után általában kisebb napdíj kérhető, mint egy tapasztalt programozó után, de egy vezető programozó és egy vezető tesztmérnök már ugyanolyan értékes. Általában elvárás, hogy a szenior programozónak / tesztelőnek már legyen egy-két szakmájába vágó ipari vizsgája is, pl. Microsoft Certified Professional, azaz MCP vizsgája.

#### 6.3.2. COCOMO modell

A Constructive Cost Model (röviden: COCOMO) egy algoritmikus szoftverköltség-közelítő módszer, amelyet Barry Boehm dolgozott ki és publikált 1981-ben. Ezt nevezzük COCOMO 81-nek. A modell konkrét projektekből levezetett paramétereket használ, amelyek állíthatók a konkrét projekt karakterisztikájához. A COCOMO-t gyakran konstruktív költség-modellnek fordítjuk.

A modell első változata a vízesés módszertanhöz alkalmazkodott. Az újabb módszertanokhoz, azaz az iteratív módszertanokhoz adaptált változata a COCOMO II, ami 2000-ben jelent meg a „Software Cost Estimation with COCOMO II” című könyvben.

A COCOMO 81 szintjei:

- Basic COCOMO – Alap COCOMO: gyors, nyers, egy nagyságrenden belül pontosan becsül.
- Intermediate COCOMOs – Középszintű COCOMO-k: költségtényezőket (angoul: cost drivers) definiál, amelyeket a projekt karakterisztikájához állíthatók és ez alapján becsül.
- Detailed COCOMO – Részletes COCOMO: az életciklus minden eleméhez külön becsli a költségtényezőket.

Az alap COCOMO a tervezett forráskódban lévő sorok számából (angolul: source lines of code, SLOC) számolja a közelítést a következő képlet alapján:

- Emberhónapok száma =  $A * (\text{SLOC}/1000)^B$
- Fejlesztésre szánt hónapok száma =  $C * (\text{Emberhónapok száma})^D$
- Programozók száma = Emberhónapok száma / Fejlesztésre szánt hónapok száma

A képletekben használt konstansok attól függnek, hogy mennyire bonyolult a projekt. Ebből a szempontból az alap COCOMO a következő projekteket különbözteti meg:

- Organic project - Organikus (életszagú) projekt: kis fejlesztőcsapat sok tapasztalattal, kevésé specifikált feladat.
- Semi-detached project – Félig-meddig projekt: közepesen nagy fejlesztőcsapat, gyakorlott és gyakorlatlan programozókkal, jól és kevésbé jól specifikált követelményekkel.
- Embedded project – Beágazott projekt: jól specifikált követelmények jól definiált megszorításokkal.

Az egyes projekttípusok esetén a konstansok értéke:

Projekt típusa	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Szerencsére az Interneten több kalkulátor is található, amivel kiszámolhatjuk a COCOMO által becsült értékeket:

- COCOMO 81:  
[http://sunset.usc.edu/research/COCOMOII/cocomo81\\_pgm/cocomo81.html](http://sunset.usc.edu/research/COCOMOII/cocomo81_pgm/cocomo81.html)
- COCOMO II: <http://www.cms4site.ru/utility.php?utility=cocomoii>

További költségmodellek:

- [Cost Estimating Guidelines](#)
- [Cost Spreading Calculator](#)

#### 6.4. Árajánlat

A árajánlat kialakításához legalább két dolgot kell tudnunk:

- Hány embernapig fog tartani a fejlesztés, illetve
- mekkora a napidíjunk.

A kettőt csak össze kell szorozni, és kész az árajánlat. Azért ettől egy kicsit bonyolultabb a helyzet, de most maradjunk ennél az egyszerű módszernél. Az ütemezésben nem véletlenül adtuk meg a prioritásokat. A prioritások segítenek kiválasztani a funkciók közül azokat, amelyekre az árajánlatot adjuk. Érdemes egy minimál, egy optimál és egy maximál csomagra árajánlatot adni. A minimál csomagba az 1-es prioritásúak kerülnek. Az optimál csomagba az összes 1-es, néhány 2-es. A maximál

csomagba az összes 1-es és 2-es prioritású funkció és néhány 3-as. Ezzel megadjuk a megrendelőnek a választás érzését, egyúttal terelgetjük az optimum csomag felé. Innen tudjuk, hogy hány embernapra lesz szükségünk. A napidíj általában konstans, tehát megvan az ár.

Ha tudjuk, hogy mennyi pénzt szán a megrendelő a fejlesztésre, és ez nagyon kevés, akkor a minimál csomagnál kisebb csomagot is kialakíthatunk. Ehhez 1-es prioritású funkciót kell elhagynunk, ami elvileg nem lehetséges, de gyakorlatban sokszor a szükség törvényt bont.

Nézzük meg, milyen részekből áll egy tipikus árajánlat:

- Címlap
- Tartalomjegyzék
- Vezetői összefoglaló
- Cégbemutató, referenciák
- Felhasználni kívánt technológiák bemutatása
- A feladat bemutatása
- Funkcionális specifikáció és ütemterv
- Csomagok árral, átadási határidővel
- Megtérülés, gazdasági előnyök
- Árajánlat érvényessége, szerződési feltételek

Az árajánlat komoly marketinganyag, így minden cég kihasználja a lehetőséget, hogy bemutassa erőségeit, referenciáit. Ezek az ajánló levelei. A vezetői összefoglaló általában nagyon rövid, egy-két oldal. A felhasznált technikák bemutatásánál elsüthetünk olyan varázsszavakat, amik megnyitják a megrendelő pénztárcáját, pl.: .NET, XML, szerviz alapú szolgáltatás.

A feladat bemutatását a cég kiírásából, felkérőleveléből másoljuk ki, esetleg a követelményspecifikációból. Ha már elkészült a funkcionális specifikáció, akkor azt is elhelyezzük az árajánlatban. Ha nincs még ilyenünk, akkor is egy ütemtervet illik elhelyezni. Ezután több csomagot ajánlunk fel. Érdemes a nagyobb csomagok mellé ingyenes pluszszolgáltatásokat tenni.

A jelenlegi piacon elvárás, hogy minden egyedi szoftver mellé adjunk 1 év ingyenes hibajavítást. Ez nem jelenti új funkciók ingyenes fejlesztését, csak a meglévőkben felfedezett hibák javítását. Pluszszolgáltatásként ajánlhatunk 1 évesnél hosszabb ingyenes hibajavítás.

Általában megadjuk, hogy a hiba bejelentése után hány órával kezdünk neki a hibajavításnak, illetve hány munkanapon belül javítjuk a hibát. Ezek az értékek széles skálán mozognak. Ha rendelkezik a cégnk 24 órás felügyelttel, akkor vállalhatjuk, hogy a hiba bejelentése után 5-10 perccel megkezdjük a javítást. Ha ehhez nincs meg az infrastruktúránk, akkor általában a hiba bejelentése után 4-8 órával, vagy csak a következő munkanapon vállaljuk a hiba javításának elkezdését.

A következő kérdés, hogy milyen gyorsan javítjuk a hibát. Mivel nagyon nehéz előre megbecsülni, hogy mennyi idő szükséges egy előre nem látott hiba javítására, ezért erre bőven hagyunk időt. Ennek megfelelően a hibajavítás lehet 1 munkanapos, 2 munkanapos, 1 hetes vagy 2 hetes határidejű. A kritikus hibák nál, amik megakasztják a megrendelő munkamenetét, általában elvárás a minél gyorsabb reagálás és javítás.

Az árajánlat általában 1-2 hónapig érvényes. Mivel jogilag az árajánlat ajánlattételnek minősül, ezért itt már a szerződés feltételeire is ki kell térdílni. Például kérhetjük, hogy a megrendelő biztosítson egy, a sajátjával megegyező szervert, amin fejleszteni lehet.

Nagyon fontos rész a megtérülés és a gazdasági előnyök elemzése. Ettől függ, megveszik-e a rendszerünket. Általában elvárás, hogy egy szoftver vagy előnyt biztosítson a konkurenciával szemben, vagy 4-5 éven belül megtérüljön. Általában az első maga után vonzza a másodikat. Sajnos itt nem lehet mellébeszélni, de van néhány közismert előnye az informatikai rendszereknek:

- az automatizált folyamatok kevesebb munkaerőt igényelnek,
- a program által vezérelt gyártás kevesebb selejtet termel,
- az adatok gyorsan visszakereshetők.

Ugyanakkor egy szoftverrendszernek mindenkor vannak költségei a vételáron felül, amivel a megterülésnél számolni kell. Ezek általában a következők:

- rendszergazdák béré,
- szoftver-, hardverhibából adódó termeléskiesés.

A gazdasági adatokat gyakran megvalósíthatósági tanulmányba foglaljuk.

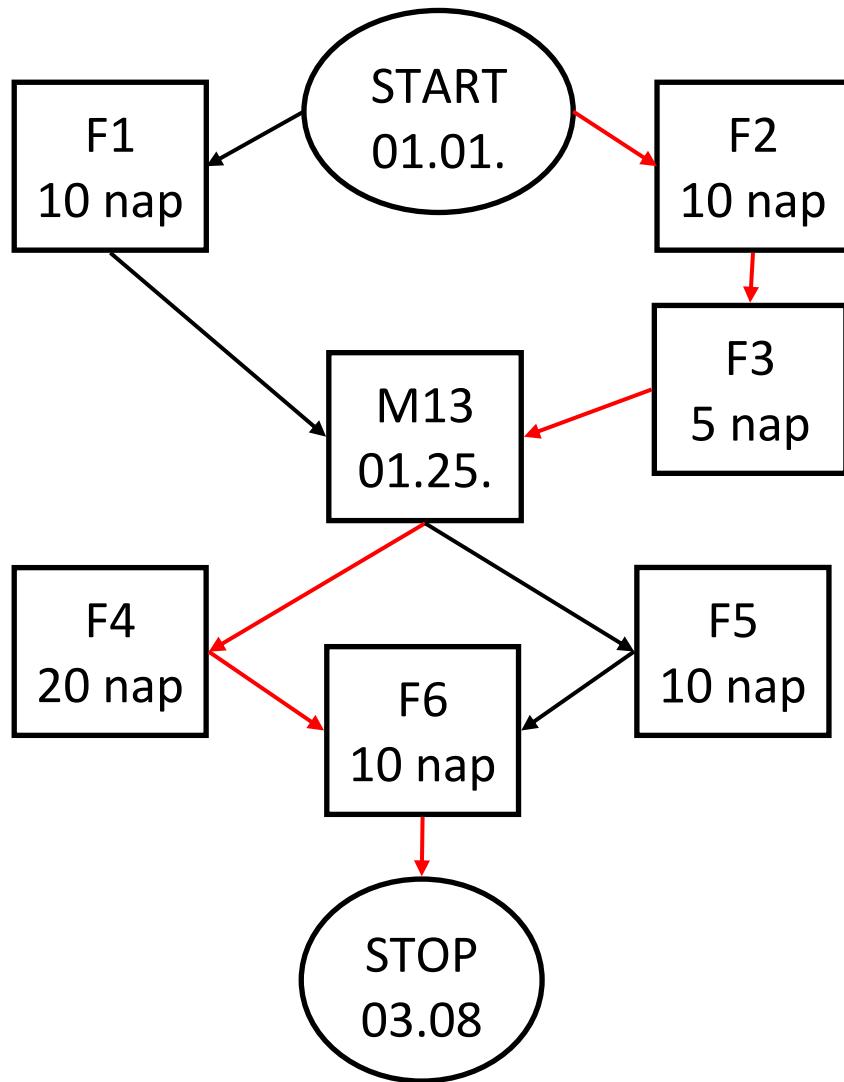
#### 6.4.1. Projektütemezés

Egyetlen egy kérdés maradt. Hogyan állapítsuk meg, hogy milyen határidővel tudjuk szállítani a rendszert. Erre a projektütemezés ad választ. Ehhez a feladatot részekre kell bontanunk. Ez általában megfelel a funkcióknak, tehát ez adott. Ezután megnézzük, mely részfeladatok végezhetőek párhuzamosan, és melyik épül valamelyik másikra. Ebből felállítjuk a függőségi gráfot.

Vegyük egy tipikus projektütemezést:

- F1, adatbázis tervezése, 10 embernap.
- F2, felhasználói felület tervének elkészítése, 10 embernap.
- F3, felhasználói felület elkészítése, 5 embernap.
- F4, fő funkció elkészítése és tesztelése, 20 embernap.
- F5, kényelmi funkciók elkészítése és tesztelése, 10 embernap.
- F6, bevezetés, 10 embernap.

Egy lehetséges függőségi gráf:



29. ábra: Egy függőségi gráf

#### 6.4.2. Kritikus út

A kritikus út módszert (angoul: Critical Path Method, CPM) az 1950-es évek végén dolgozták ki. minden olyan projektszervezési területen használható, ahol egymástól független részfeladatok is vannak. A módszer lényege, hogy a részfeladatokat függőségeik szerint lerajzoljuk. Lásd az „Egy függőségi gráf” című ábrát. Egy részfeladat közvetlenül függ egy másiktól, ha csak annak elvégzése után kezdhetünk hozzá. Ebből lesz a függőségi gráf, miután START és STOP csúcson adunk hozzá. Lehet még mérföldkőcsúcsokat is definiálni, amiről később lesz szó. A fenti függőségi gráfban egy mérföldkő van, az M13 jelű. minden részfeladathoz megadjuk, hogy hány embernapba kerül az elvégzése (ezt az ütemtervből tudjuk). Az így előálló gráfban a leghosszabb START-ból STOPba vezető út a kritikus út (angolul: critical path).

Az út hosszán az úton lévő részfeladatok embernapjainak összegét értjük. minden csúcshoz megmondható, hogy minimum, illetve maximum hány nap kell az eléréséhez. Ez a START-ból induló legrövidebb és leghosszabb út hosszából adódik. A két szám különbsége a részfeladat maximum időtartaléka (total float). A kritikus úton lévő csúcsoknak nincs időtartaléka. A fenti ábrán a kritikus út

az F2, F3, F4, F6. Ezt piros nyilak jelölik. Egy utat közel kritikusnak nevezünk, ha hossza közel megegyezik a kritikus útéval. A „közel megegyezik” általában 10%-20%-kal kisebb értékre utal.

A projektmenedzser feladata, hogy megfelelő erőforrásokat biztosítson a kritikus út feladatainak elvégzésére, illetve figyelemmel kísérje, hogy nem válik-e egy közel kritikus út kritikussá. Mivel a feladatok embernapban vannak megadva, ezért egy feladat ideje csökkenhető, ha több programozót állítunk a feladatra.

Vigyázat, a rendszerszervezés egyik alaptétele szerint ez veszélyes. Fred Brooks törvénye kimondja, hogy: **Új munkatárs felvétele egy késésben lévő szoftverprojekthez csak további késést okoz**. Ennek oka, hogy az új programozónak kommunikálnia kell a meglévőkkel. Ha N programozón van, akkor  $N(N-1)/2$  kommunikációs út létezik köztük. Könnyen beláthatjuk, hogy egy új programozó felvétele négyzetesen, matematikai jelöléssel  $O(N^2)$ , növeli a kommunikációs utak számát. A sok megbeszélés időt emész fel, ami csak további késést okoz.

A függőségi gráfba elhelyezhetünk mérföldköveket (milestone) is. minden mérföldkőnél átadás történik, ami általában fizetséggel is jár. Ha a megrendelő elfogadja az átadott részeket, akkor egy teljesítmény igazolást / jegyzőkönyvet kapunk, ami alapján kiszámlázzuk az elkészült mérföldkő árát, ami általában a szerződésben előre rögzített. Tehát érdemes 1-2 havonta egy-egy mérföldkövet definiálni.

A mérföldkő átadásának időpontja kiszámítható a START vagy az előző mérföldkő időpontjából, a mérföldkőbe vezető kritikus út hosszából és az addig eltelő szabad- illetve szünnapok számából. Érdemes az átadásokat hétfői napra csúsztatni, hogy legyen egy hétvége, hátha kell még egy kis idő a befejezéshez. A STOP dátuma, azaz a végső átadási határidő is így számítható.

Fontos, hogy a mérföldkövek kommunikációt jelentenek a megrendelővel. Így visszajelzést kaphatunk, ha esetleg valami változás történt, vagy más képp képzelték az eddig elkészült részeket. Ha minden rendben van, akkor kérjünk teljesítmény igazolást, amit mellékeljünk a számlánkhoz. Ennek elmaradása esetén általában nem fizet a megrendelő pénzügyi részlege. Ha valamit kifogásol a megrendelő, akkor ez általában csúszást okoz. Ennek megfelelően kezdeményezni kell a szerződés módosítását. Ha ezt nem tessük meg, akkor később a megrendelő kevésbé hajlandó erre.

Ha a kritikus utat automatikusan szeretnénk számoltatni, vagy ha egy feladathoz több programozót akarunk rendelni, vagy más erőforrásokat (pl. irodák, számítógépek) is kezelní szeretnénk, akkor már érdemes projektmenedzsment szoftvert használnunk. Erre a feladatra például jól megfelel az MS Project.

## 6.5. Megvalósíthatósági tanulmány

A projekt megvalósíthatósági tanulmánya általában egy 10-50 oldalas dokumentum a projekt nagyságától függően. A megvalósíthatósági tanulmány célja, hogy megfelelő információkkal lássa el a döntéshozókat a projekt indításával, finanszírozásával kapcsolatban. Mint ilyen, megelőzheti az árajánlat adását. Informatikai rendszereknél akkor jellemző, ha ez a rendszer más környezeti, társadalmi kockázatokat rejtő rendszerhez kapcsolódik, vagy egy pályázat előírja.

A megvalósíthatósági tanulmány feladata, hogy bemutassa a projekt pénzügyi megalapozottságát, fenntarthatóságát. A tanulmány ismeretében döntik el a döntéshozók, hogy a projekt megvalósítható-e, az elvárt időn belül megtérül-e.

Gyakran több lehetséges alternatívát is felsoroltat, amelyeknek általában különböző a befektetési, finanszírozási igényük és a megtérülésük is. Ugyanakkor minden alternatíva megvalósítja a projekt célját.

A megvalósíthatósági tanulmány elkészítésének főbb lépései:

- Projektötletek kidolgozása.
- Jelenlegi helyzet vizsgálata.
- A szükséglet vizsgálata, amelyre a projekt reagál.
- Alternatív megoldások elemzése.
- A projekt megvalósításának elemzése.
- Pénzügyi elemzés.
- Környezeti, környezetvédelmi hatások elemzése.
- Gazdasági-társadalmi hatások elemzése.
- A projekt megvalósíthatóságának és fenntarthatóságának értékelése.

Látható, hogy ennek a dokumentumnak sok része átemelhető a funkcionális specifikációból. Az egyéb részeit pénzügyi szakemberek bevonásával kell elkészíteni.

## 6.6. A rendszerterv fajtái

A rendszerterv egy írásban rögzített specifikáció, amely leírja

- mit (rendszer),
- miért (rendszer célja),
- hogyan (terv),
- mikor (időpont),
- és miből (erőforrások)

akarunk a jövőben létrehozni. Fontos, hogy reális legyen, azaz megvalósítható lépéseket írjon elő. A rendszerterv hasonló szerepet játszik a szoftverfejlesztésben, mint a tervrajz az építkezéseken, tehát elég részletesnek kell lennie, hogy ebből a programozók képesek legyenek megvalósítani a szoftvert. A rendszerterv vagy új rendszert ír le, vagy egy meglévő átalakítását.

Három fajta rendszertervet különböztetünk meg:

- konceptuális (mit, miért),
- nagyvonalú (mit, miért, hogyan, miből),
- részletes (mit, miért, hogyan, miből, mikor).

A konceptuális rendszerterv röviden írja le, mit és miért akarunk a jövőben létrehozni. Egy rendszernek több változata lehet, amelyek közül választunk. A követelményspecifikáció alapján jön létre. Része lehet az árajánlatnak.

A nagyvonalú rendszerterv a mit és miért részen túl kiegészül egy hogyan és miből résszel, azaz megadjuk, hogy milyen lépésekkel kell véghezvinni és az egyes lépésekhez milyen erőforrásokra van szükségünk. Elegendő nagyvonalakban megadni a tervet, mert feltételezzük, hogy a tervező részt vesz a végrehajtásban, így a felmerülő kérdésekre tud válaszolni.

A nagyvonalú rendszerterv fontos része az úgynevezett megfeleltetés, ami megmutatja, hogy a követelményspecifikációban felsorolt minden követelményhez van-e azt kielégítő lépés.

A részletes rendszerterv a mit, miért, hogyan és miből részeken túl tartalmaz egy mikor részt is, azaz megadja a lépések idejét. Az időpont lehet pontos vagy csak időintervallum. Ezeket olyan részletességgel adja meg, hogy a tervező részvételle nélkül is végrehajtható legyen.

Nagy Elemérné és Nagy Elemér Rendszervezés című főiskolai jegyzetéből (SzTE SzÉF 2005) idézünk egy-egy példát nagyvonalú, illetve részletes rendszertervre.

Példa nagyvonalú rendszertervre:

Mit: Fiatal házaspár használt lakást akar (Hogyan:) vásárolni Szegeden (Miből:) maximum 6 MFt-ért, 3 hónapon belüli beköltözéssel.

Miért (miért pont azt):

Maximum ennyi pénzt tudnak mozgósítani.

Fiatal házasok, albérletben laknak és jön a gyerek.

Mindketten Szegeden dolgoznak.

Most épülő lakás nem lesz kész három hónap alatt.

Példa részletes rendszerterv:

"Most" 2005. 03. 15. van.

\* Apróhirdetés feladása a helyi lapokban: "Fiatal házaspár használt lakást akar vásárolni Szegeden, 1 hónapon belüli beköltözéssel. Tel: (62)-123-456 18 óra után és vasárnap." 03.19-re és 03.26-ra. Hi: 03.16.

\* Eladási apróhirdetések figyelése 03.20-03.30.

\* Pénz "mozgósítás" megkezdése. Hi: 03.20.

\* Elemzések, tárgyalások, válogatások, alkudozások 03.16-03.30.

\* Döntés. Hi: 03.30.

\* Pénz "begyűjtésének" ütemezése: 03.31-04.02.

\* Ügyvéd szerzése: 03.31-04.01.

\* Pénz a szerződéskötéshez. Hi: 04.04.

\* Szerződéskötés: 04.04. és 04.08. között.

\* Szakember "lebiztosítása" festéshez. Hi: 05.03.

\* Pénz a lakás átvételhez. Hi: 05.11.

\* Üres lakás átvétele: 05.12-ig.

\* Pénz a szakemberekre, fuvarra és az új holmikra 05.13-05.31

\* Albérlet felmondása. Hi: 05.14.

\* Kölözés előkészítése (selejtezés, dobozok stb.) 05.22-05.31.

\* Festés, fali polcok szerelése, nagytakarítás stb. 05.13-05.31.

- \* Új holmik vásárlása (pl. nélkülözhetetlen bútorok) 05.15-06.06.
- \* Fuvar lebiztosítás, barátok, rokonok "mozgósítása" a költözéshez. Hi: 06.06.
- \* "Beköltözésre kész" a lakás. Hi: 06.06.
- \* Pénz a beköltözéshez
- \* Költozés, berendezkedés: 06.07-06.12

#### Megjegyzések.

- \* A hirdetésben nem közöljük, hogy mennyi pénzünk van.
- \* Nem 3, hanem 1 hónapon belüli beköltözést kértünk, mert időt tartalékoltunk a keresésre és az átadás utáni festésre stb.
- \* A telefonszám megadása gyorsíthatja a kapcsolatba lépést - nem érünk rá.
- \* Közben figyeljük az eladók hirdetéseit is.
- \* Általában tól-ig időintervallumokat adunk meg, pontos határidő (Hi:) csak a "sarkpontoknál" szerepel.
- \* A pénzmozgósítás ütemezése egy külön nagyvonalú rendszerterv lesz (mint ennek a rendszernek egy alrendszer). Most még nem tudjuk megtervezni, hiszen a részletek (mikor mennyit kell fizetnünk) csak 03.30. körül derülnek ki.
- \* A naptárat figyelembe vettük; pl. az ügyvéddel valószínűleg csak munkanap tudunk találkozni, a hétvégékre szükség lehet, ha pl. a pénzért utazni kell, a szakemberek Pünkösdkor nem dolgoznak, a költöztető barátok szombaton jobban ráérnek stb.
- \* Óvatosan tervezünk, inkább legyen tartalék időnk, mint feszített ütemezésünk, mert váratlan "apróságok" biztosan be fognak következni, csak most még nem tudjuk, hogy mik.

## 6.7. A rendszerterv elkészítése

Egy rendszerterv általában az alábbi fejezetekből és alfejezetekből áll:

- A rendszer célja: Definiálja a rendszer célját. Gyakran leírjuk azt is, ami nem cél, hogy ezzel is tisztázzuk a feladatkört (scope), amit meg akarunk oldani.
- Projektterv: Itt soroljuk fel a rendszer létrehozásához rendelkezésre álló erőforrásokat. Ezek közül a két legfontosabb az emberek és az idő. Fontos tisztázni a felelősségi köröket. Itt adjuk meg az ütemterv alapján a mérföldköveket. Részei:
  - Projektszerepkörök, felelősségek
  - Projektmunkások és felelősségeik
  - Ütemterv
  - Mérföldkövek
- Üzleti folyamatok modellje: Itt adjuk meg a támogatandó vagy kiváltandó üzleti folyamatokat. Leírjuk az eseményeket, a felhasznált erőforrásokat, a folyamatok bemeneteit, kimeneteit, a szereplőket. A modell célja a megértés, így sok példát is tartalmazhat. Részei:
  - Üzleti szereplők
  - Üzleti folyamatok
  - Üzleti entitások
- Követelmények: Itt a teljes követelménylistából csak azokat soroljuk fel, amelyek megvalósítását megcélozza a rendszerterv. Fontos, hogy az itt leírt követelmények a tényleges követelmények legyenek félreértések nélkül. A követelmények a fejlesztés során változhatnak.

Erre a különböző módszertanok más és más választ adnak. Itt kell felsorolnunk a vonatkozó törvényi előírásokat, szabványokat, amiket be kell tartanunk. Részei:

- Funktionális követelmények
- Nemfunkcionális követelmények
- Törvényi előírások, szabványok
- Funkcionális terv: A funkcionális terv a fejlesztők szemszögéből írja le az elkészítendő funkciókat a funkcionális specifikáció alapján (ami a felhasználó szemszögéből írja le a funkciókat). Nagyon fontos részei a használati esetek lefutásai. Ezek adják meg, hogyan kell megvalósítani a funkciót. Ezek általában aktivitási és szekvencia UML diagramok. A határ osztályok (presenter) a képernyők tartalmát és funkcionálitását leíró osztályok. Részei:
  - Rendszerszereplők
  - Rendszerhasználati esetek és lefutásaik
  - Határ osztályok
  - Menühierarchiák
  - Képernyőtervezek
- Fizikai környezet: Itt határozzuk meg, hogy milyen platformon (Java, .NET, ...) fogunk fejleszteni, milyen operációs rendszerre és hardverre. Gyakran fontos tudnunk a hálózat felépítését is, például, hogy van-e túzfal, az milyen portokat engedélyez. Ha vannak megvásárolt komponenseink, azokat is itt kell megadnunk. Részei:
  - Vásárolt softwarekomponensek és külső rendszerek
  - Hardver és hálózati topológia
  - Fizikai alrendszer
  - Fejlesztő eszközök
  - Keretrendszer (pl. Spring)
- Absztrakt domain modell: Itt írjuk le a megvalósítandó rendszer fogalmait, illetve a megvalósítás nagyon magas szintű vázát általában egy-két konkrét példán keresztül. Megadjuk a fő komponenseket és ezek kapcsolatait. Ez a rendszer nagyvonalú (vagy csak konceptuális) terve. Részei:
  - Domainspecifikáció, fogalmak
  - Absztrakt komponensek, ezek kapcsolatai
- Architekturális terv: A nemfunkcionális követelmények alapján kell kialakítani. Ez csak akkor lehetséges, ha ezek mögé nézünk. Pl. ha az a követelmény, hogy 10 000 felhasználót kell kiszolgálnia a rendszernek, akkor meg kell tudnunk, mi történik a 10 001. felhasználóval, mi van, ha ez a vezérigazgató. Fontos, hogy a választott architektúra könnyen tudjon alkalmazkodni a változásokhoz (pl. konfigurációs állományok használata) és rugalmasan bővíthető legyen. Itt adhatjuk meg a biztonsági funkciókat, pl. a jogosultság kezelését. Részei:
  - Egy architekturális tervezési minta (pl. MVC, 3-rétegű alkalmazás, ...)
  - Az alkalmazás rétegei, fő komponensei, ezek kapcsolatai
  - Változások kezelése
  - Rendszer bővíthetősége
  - Biztonsági funkciók
- Adatbázisterv: Meg kell adni a táblákat, a köztük lévő kapcsolatokat. Általában elvárás, hogy az adatbázis terv 3. normálformában legyen. Itt lehet megadni a tárolt eljárásokat is. Részei:
  - Logikai adatmodell

- Tárolt eljárások
  - Fizikai adatmodellt legeneráló SQL szkript
- Implementációs terv: Az implementációs terv adja meg a megvalósítás osztályait. Meg kell felelnie a kiválasztott architektúrának. A lenti alfejezetek 3-rétegű alkalmazás esetén használhatóak. Az implementációs tervben érdemes tervezési mintákat alkalmazni és betartani a tervezési alapelvek ajánlásait, hogy rugalmas, könnyen bővíthető és módosítható szerkezetet kapunk. Részei:
  - Perzisztencia osztályok
  - Üzleti logika osztályai
  - Kliens oldal osztályai
- Teszterv: Lefekteti a tesztelés elveit, folyamatát és kontrollját. Meghatározza a fő teszeseteket. Meghatározza a sikeres teszt kritériumait.
- Telepítési terv: A rendszer kialakítására (pl. szervertelepítés), a telepítő csomag elkészítésére vonatkozó elvek, megszorítások. Itt is meg lehet adni a fizikai környezetet.
- Karbantartási terv: A szoftver frissítésének módja, folyamata. Karbantartási teszterek. Általában csak akkor készül el, ha már egy verziót áadtunk és a következő verziót tervezük.

#### 6.7.1. Határosztály-tervezés

Az adatbeviteli felület tervezésének célja, hogy meghatározza az adatbeviteli felületeket, mezőszinten teremtse meg a kapcsolatot az adatmodell entitásosztályai és az adatbeviteli felületek között.

Egy határosztály alatt egy olyan logikai egységet/osztályt értünk amelyik:

- a szereplő és a rendszer közötti kommunikációért felelős,
- inputot fogad és továbbít,
- outputot generál.

A határosztályokban szereplő funkciókat a határosztályok metódusaiban kell megadni. A határosztályok mezői megadják a képernyőn megjelenő input mezőket.

A határosztályok közötti statikus kapcsolatokat osztálydiagramokkal mutathatjuk meg. Például határosztály lehet egy képernyő is, de határosztálynak tekinthetjük egy képernyő egyik részét is. Például egy keresési ablakban a keresőmező és a találati lista két különböző határosztálynak tekinthető.

Hatórosztályok közötti függőségeket, hogy melyik határosztályból hová lehet eljutni, úgynévezett kollaborációs diagramokkal mutatjuk meg. A diagramokon a határosztályok lesznek a csomópontok és az őket összekötő asszociációk reprezentálják a képernyők egymásutániságát.

#### 6.7.2. Menühierarchia-tervezés

A menühierarchia-tervezés célja, hogy meghatározza az adott alkalmazás menüszerkezetét, az egyes menüelemekhez hozzárendelje az adott alkalmazás által szolgáltatott funkciókat.

A menühierarchia-tervezést három lépésben valósítjuk meg:

- Először megtaláljuk az alkalmazás fő menüstruktúrájának elemeit.
- Majd a logikai alrendserek menüstruktúráját kell meghatározni.

- Legvégül a menüelemeket össze kell kötni a határosztályokkal egy osztálydiagramon, így mutatva meg azt, hogy melyik menüt választva melyik képernyő jelenik meg.

A menühierarchiát általában osztálydiagramon ábrázoljuk.

#### 6.7.3. [Storyboard](#)

A határosztály-modellezésnek egy felsőbb szintű nézetét mutatja a Storyboard-modellezés. Megmutatja, hogy melyik ablakból milyen szolgáltatások érhetők el.

*Előnyei:*

- ablak elérhetőségi hierarchiák megértését biztosítja
- user interface designer-eknek szól
- rövid tevékenység-leírások

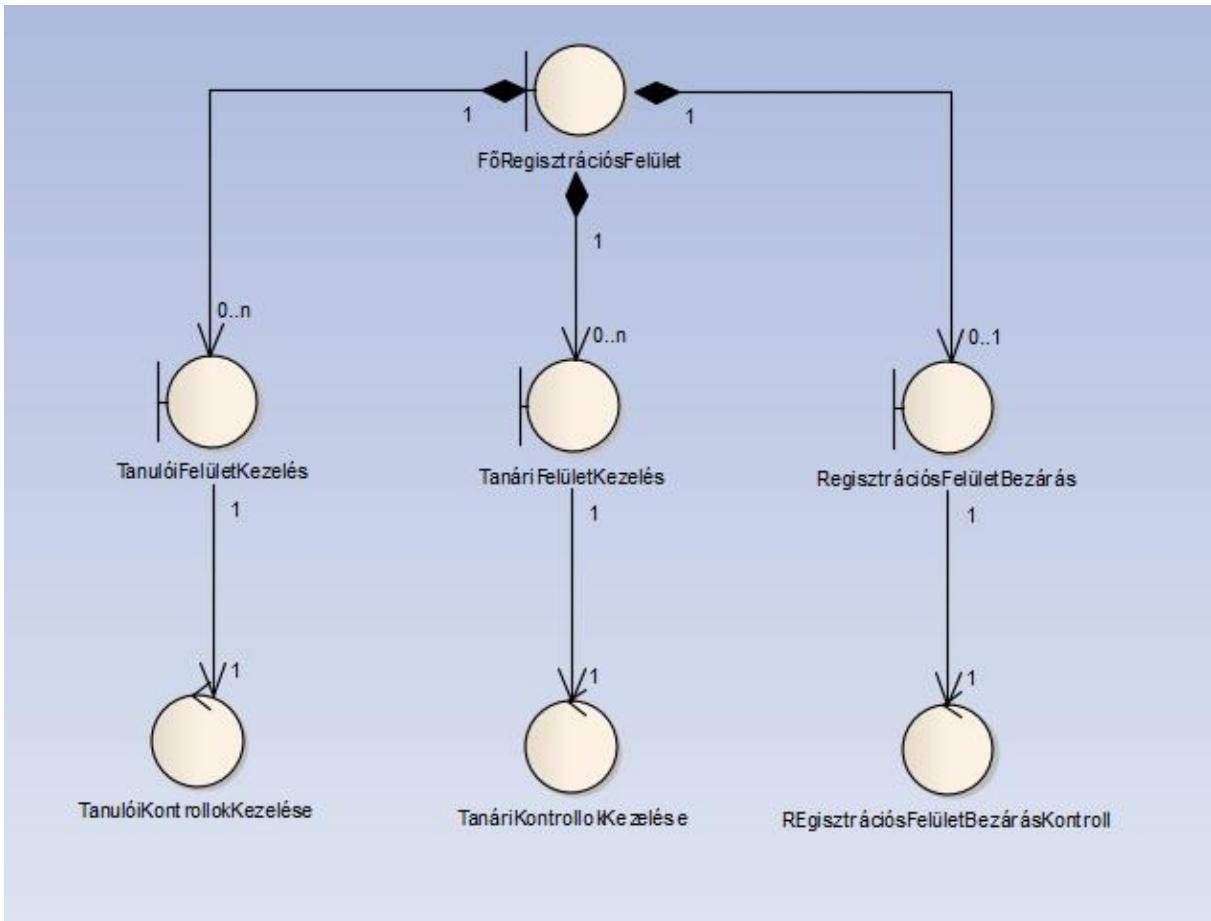
*Storyboard-tervezés kérdései:*

- Milyen szolgáltatásokra lenne a felhasználónak szüksége?
- Milyen szolgáltatásokat tud a rendszer adni?
- A fentiek közül milyen szolgáltatásokat kéne megvalósítani?

Storyboard példa:

A használati eset kezeli a bejövő üzeneteket, mielőtt használhatósági szempontokkal kiegészülne.

- a) A használati eset akkor indul, mikor a levelező user kérést küld, hogy üzeneteket kezelhessen, és a rendszer megjeleníti azokat.
- b) A levelező user ezután követhet a következő lépésekkel egyet vagy többet:
- c) Rendezheti az üzeneteket küldő vagy tárgy szerint.
- d) Elolvashatja az üzeneteket.
- e) Fájlként mentheti az üzenetet.
- f) Az üzenet csatolmányát mentheti fájlként.
- g) A használati eset véget ér, mikor a levelező user kérést indít a bejövő üzenetkezelőből.



30. ábra: Egy UML storyboard példa

#### 6.7.4. Logikai rendszerterv

A logikai rendszerterv a következő főbb részekből áll:

- üzleti folyamatok modellje
  - business domain modell
  - üzleti szereplők
  - üzleti entitások
- követelmények
  - funkcionális
  - nemfunkcionális
- feldolgozási folyamatok
  - használati esetek
  - aktivitási diagramok
  - állapotgépek
  - szekvenciadiagramok
- funkcionális felépítés
  - komponensek
- felhasználói felületek, menük
- adatszótár, logikai adatmodell

- adatfolyam-diagramok

#### 6.7.5. Fizikai rendszerterv

A fizikai rendszerterv a következő főbb részeiből áll:

- osztálytervek
- adatbázisterv
- teszterek
- telepítési terv
- rendszerspecifikációk (fejlesztési, futtatási környezet)
- szoftverarchitektúra
- az alkalmazás rétegei
- adatspecifikációk/objektumspecifikációk (környezetfüggő adattervezek)
- programspecifikációk (modulvázak)

### 6.8. A tesztelés elmélete

A tesztelés nem csak tesztek készítéséből és futtatásából áll. A leggyakoribb tesztelési tevékenységek a következők:

- tesztervezés,
- teszesetek tervezése,
- felkészülés a végrehajtásra,
- tesztek végrehajtása,
- kilépési feltételek vizsgálata,
- eredmények értékelése,
- jelentéskészítés.

A tesztervezés fontos dokumentum, amely leírja, hogy mit, milyen céllal, hogyan kell tesztelni. A tesztervezés általában a rendszerterv része, azon belül is a minőségbiztosítás (quality assurance, QA) fejezetéhez tartozik. A teszt célja lehet:

- megtalálni a hibákat,
- növelni a megbízhatóságot,
- megelőzni a hibákat.

A fejlesztői tesztek célja általában minél több hiba megtalálása. Az átvételi teszt célja, hogy a felhasználók bizalma nőjön a megbízhatóságban. A regressziós teszt célja megelőzni, hogy a változások a rendszer többi részében hibákat okozzanak.

A tesztervezéséhez a célon túl tudni kell, hogy mit és hogyan kell tesztelni, mikor tekintjük a tesztet sikeresnek. Ehhez ismernünk kell a következő fogalmakat:

- A teszt tárgya: A rendszer azon része, amelyet tesztelünk. Ez lehet az egész rendszer is.
- Tesztbázis: Azon dokumentumok összessége, amelyek a teszt tárgyára vonatkozó követelményeket tartalmazzák.

- Tesztadat: Olyan adat, amivel meghívjuk a teszt tárgyát. Általában ismert, hogy milyen értéket kellene erre adnia a teszt tárgyának vagy milyen viselkedést kellene produkálnia. Ez az elvárt visszatérési érték illetve viselkedés. A valós visszatérési értéket illetve viselkedést hasonlítjuk össze az elvárttal.
- Kilépési feltétel: minden tesztnél előre meghatározzuk, mikor tekintjük ezt a tesztet lezáráthatónak. Ezt nevezzük kilépési feltételnek. A kilépési feltétel általában az, hogy minden teszeset sikeresen lefut, de lehet az is, hogy a kritikus részek tesztlefedettsége 100%.

A teszterv leírja a teszt tárgyát, kigyűti a tesztbázisból a teszt által lefedett követelményeket, meghatározza a kilépési feltételt. A tesztadatokat általában a teszterv nem tartalmazza, azokat csak a teszesetek határozzák meg. Ugyanakkor a teszterv gyakran tartalmazza teszeseteket is.

A teszesetek leírják, hogy milyen tesztadattal kell meghajtani a teszt tárgyát, illetve, hogy mi az elvárt visszatérési érték vagy viselkedés. A tesztadatok meghatározásához általában úgynévezett ekvivalencia-osztályokat állítunk fel. Egy ekvivalencia-osztály minden elemére a szoftver ugyanazon része fut le. Természetesen más módszerek is léteznek, amikre később térünk ki.

A teszesetek végrehajtásához teszkörnyezetre van szükségünk. A teszkörnyezet kialakításánál törekedni kell arra, hogy a lehető legjobban hasonlítsan az éles környezetre, amely a végfelhasználónál működik. A felkészülés során írhatunk tesztszkripteket is, amik az automatizálást segítik.

A tesztek végrehajtása során tesztnaplót vezetünk. Ebben írjuk le, hogy milyen lépéseket hajtottunk végre és milyen eredményeket kaptunk. A tesztnapló alapján a tesztnak megismételhetőnek kell lennie. Ha hibát találunk, akkor a hibabejelentőt a tesztnapló alapján töltjük ki.

A tesztek után meg kell vizsgálni, hogy sikeresen teljesítettük-e a kilépési feltételt. Ehhez a teszesetben leírt elvárt eredményt hasonlítjuk össze a tesztnaplóban lévő valós eredménnyel a kilépési feltétel alapján. Ha a kilépési feltételek teljesülnek, akkor mehetünk tovább. Ha nem, akkor vagy a teszt tárgya, vagy a kilépési feltétel hibás. Ha kell, akkor módosítjuk a kilépési feltételt. Ha a teszt tárgya hibás, akkor a hibabejelentő rendszeren keresztül értesítjük a fejlesztőket. A teszteket addig ismétljük, míg mindegyik kilépési feltétele igaz nem lesz.

A tesztek eredményei alapján további teszteket készíthetünk. Elhatározhatjuk, hogy a megtalált hibákhoz hasonló hibákat felderítjük. Ezt általában a teszterek elő is írják. Döntethetünk úgy, hogy egy komponenst nem érdemes tovább tesztelni, de egy másikat tüzetesebben kell tesztelni. Ezek a döntések a teszt irányításához tartoznak.

Végül jelentést kell készítenünk. Itt arra kell figyelnünk, hogy sok programozó kritikaként éli meg, ha a kódjában a tesztelők hibát találnak. Úgy érzi, hogy rossz programozó és veszélyben van az állása. Ezért a tesztelőket nem várt támadások érhetik. Ennek elkerülésére a jelentésben nem szabad személyeskedni, nem muszáj látnia a főnöknek, kinek a kódjában volt hiba. A hibákat rá lehet fogni a rövid időre, a nagy nyomásra. Jó, ha kiemeljük a tesztelők és a fejlesztők közös célját, a magas minőségű, hibamentes szoftver fejlesztését.

#### 6.8.1. Tesztterv

A tesztelés első lépéseként a tesztelést el kell tervezni. Ehhez kell egy tesztterv. A tesztterv célja, hogy összegyűjtse a teszteléshez szükséges információkat ahhoz, hogy meg lehessen írni és szervezni a teszteket.

A tesztterv első lépéseként azonosítani kell a tesztelés követelményeit, a teszt terjedelmét és szerepét a rendszer fejlesztésében. A követelmények összegyűjtése után elő kell, hogy álljon egy dokumentum, amelyikben ezt leírjuk.

A tesztelés tervezésének következő lépésében megadjuk a prioritását a különböző teszteseteknek, majd meghatározzuk a tesztelési stratégiát. A tesztelési stratégia meghatározása alatt azonosítjuk a különböző használható teszteszközököt.

Minden használati esetre meghatározzuk a teszteseteket, amelyekben meghatározzuk a valid és az invalid értékeket a tesztesetre. minden használati esetre azonosítjuk a tesztelési eljárásokat. A nemfunkcionális követelményekre is tervezünk teszteseteket.

#### 6.8.2. Tesztmodell, -eset, -eljárás

A tesztmodell a teszteseteket, a teszteljárásokat és a köztük levő relációkat ábrázolja. A tesztmodell célja annak megmutatása, hogy mi és hogyan lesz tesztelve.

Egy teszteset bemeneti értékek, végrehajtási feltételek és várható értékek halmaza, amely arra lett kifejlesztve, hogy egy bizonyos tesztelési eljárást ellenőrizzünk.

A tesztelési eljárás egy részletes utasításhalmaz ahhoz, hogy beállítsuk, végrehajtsuk és kiértékeljük egy adott teszteset eredményeit. Egy teszteljárás leírásának a célja az, hogy azonosítsuk és kommunikáljuk a tesztelőnek szükséges információkat ahhoz, hogy beállítsa, implementálja és végrehajtsa a tesztesetet.

Tesztelési eljárás leírása:

- Lépések: Egymás után következő lépések halmaza, amelyek azt mutatják, hogy milyen lépéseket és milyen tevékenységeket kell, hogy tegyen a szereplő a teszteset során.
- Bemeneti értékek: A szereplő különböző lépései és tevékenységeinek a bemeneti értéke ahhoz, hogy produkáljuk a tesztesetet.
- Várt értékek: minden lépésnél leírjuk a várt értékek halmazát.
- Verifikációs metódus: Azok a módszerek, amelyeket használnia kell a tesztelőnek ahhoz, hogy összehasonlitsa a várt értékeket a teszt által produkált értékkal.

A tesztelési eljárásokat aktivitási diagramokkal is reprezentálhatjuk.

#### 6.8.3. Tesztszkriptek

Tesztszkriptek alatt olyan számítógép által végrehajtható utasítások halmazát értjük, amelyek automatizálják egy teszteljárás végrehajtását. A tesztszkriptek vagy eszközök generálhatók teszteszközökkel, vagy le lehet őket programozni, vagy lehet kombinálni e két metódust.

#### 6.8.4. Adattesztelés

Azokkal a tesztelésekkel ellentétben, amelyek rögzítik azokat az eseményeket, amelyeket egy felhasználó egy funkció tesztelése során csinál, az adattesztelések azon alapulnak, hogy pl. egy bizonyos szöveges mezőbe megpróbálunk több különböző adatot beírni automatikusan. Ezt addig tesszük, amíg vagy futási hibát nem kapunk, vagy az előre megadott tesztszámot el nem érjük. Futási hiba esetén a program által vezetett naplóból (log) tudunk visszakövetkeztetni, hogy mi lehetett a hiba.

### 6.9. Statikus tesztelési technikák

A statikus tesztelési technikák a szoftver forráskódját vizsgálják fordítási időben. Ide tartozik a dokumentáció felülvizsgálata is. A statikus tesztelés párja a dinamikus tesztelés, amely a szoftvert futásidőben teszteli.

A statikus tesztelési technikáknak két fajtája van:

1. felülvizsgálat és
2. statikus elemzés.

A felülvizsgálat a kód, illetve a dokumentáció, vagy ezek együttes manuális átnézését jelenti. Ide tartozik például a páros programozás. A statikus elemzés a kód, illetve a dokumentáció automatikus vizsgálatát jelenti, ahol a statikus elemzést végző segédeszköz megvizsgálja a kódot (illetve a dokumentációt), hogy bizonyos szabályoknak megfelel-e. Ide tartozik például a helyesírás ellenőrzése.

A statikus technikával más típusú hibák találhatók meg könnyen, mint a dinamikus tesztelési technikákkal. Statikus technikákkal könnyen megtalálhatók azok a kódsorok, ahol null referencián keresztül akarunk metódust hívni. Ugyanezt elérni dinamikus teszteléssel nagyon költséges, hiszen 100%-os kódlefedettség kell hozzá. Ugyanakkor dinamikus teszteléssel könnyen észrevehető, hogy ha rossz képlet alapján számítjuk pl. az árengedményt. Ugyanezt statikusan nehéz észrevenni, hacsak nincs egy szemfüles vezető programozónk, aki átlátja az üzleti oldalt is.

A statikus tesztelési technikák előnye, hogy nagyon korán alkalmazhatók, már akkor is, amikor még nincs is futtatható verzió. Így hamarabb lehet velük hibákat találni és így gazdaságosabb a hibajavítás.

#### 6.9.1. Felülvizsgálat – Bevezetés

A felülvizsgálat azt jelenti, hogy manuálisan átnézzük a forráskódot és fejben futtatjuk vagy egyszerűen csak gyanús részeket keresünk benne. Ezzel szemben áll a statikus elemzés, ahol szoftverekkel nézetjük át automatikusan a forráskódot. A felülvizsgálat fehérdobozos teszt, mivel kell hozzá a forráskód. A felülvizsgálat lehet informális, pl. páros programozás, de akár nagyon formális is, amikor a folyamatot jól dokumentáljuk, illetve a két szélsőség közti átmenetek.

Ezeket a hibákat könnyebb felülvizsgállal megtalálni, mint más technikákkal:

1. szabványuktól / kódolási szabályuktól való eltérések,
2. követelményekkel kapcsolatos hibák, pl. nincs minden funkcionális követelményhez funkció,
3. tervezési hibák, pl. az adatbázis nincs harmadik normál-formában,
4. karbantarthatóság hiánya, pl. nincs biztonsági mentés és visszaállítás funkció,
5. hibás interfész-specifikációk, pl. dokumentálatlan előfeltételek.

A felülvizsgálat legismertebb típusai:

1. informális felülvizsgálat (csoporton belüli),
2. átvizsgálás (házon belüli),
3. technikai felülvizsgálat (külsős szakérő bevonásával, rövid idejű),
4. inspekció (külsős szakérő bevonásával, hosszú idejű).

#### 6.9.2. [Felülvizsgálat – Informális felülvizsgálat](#)

Sok szoftvercégnél elfogadott megoldás, hogy egy tapasztalt programozó átnézi (angolul: review) a kezdők kódját. A kezdők a kritikából rengeteg tapasztalatot szerezhetnek. A kockázatosnak ítélt részeket (pl. amire gyakran kerül a vezérlés, vagy kevésbé ismert megoldást alkalmaz) több tapasztalt programozó is átnézheti. Ennek hatékonysága függ az átnézők rátermettségétől. Ez a leginkább informális felülvizsgálat, ezért is nevezik informális felülvizsgálatnak.

Ehhez hasonló a páros programozás (angolul: pair programming) is. Ekkor két programozó ír egy kódot, pontosabban az egyik írja, a másik figyeli. Ha a figyelő hibát lát vagy nem érti a kódot, akkor azonnal szól. A két programozó folyamatosan megbeszéli, hogy hogyan érdemes megoldani az adott problémát.

A kódszépítés (angolul: refactoring) egy másik módja a felülvizsgálatnak. Ilyenkor a már letesztelt, működő kódot lehet szépíteni, ami esetleg lassú, rugalmatlan, vagy egyszerűen csak csúnya. A kódszépítés előfeltétele, hogy legyen sok egységeszt. A szépítés során nem szabad megváltoztatni a kód funkcionalitását, de a szerkezet, pl. egy metódus törzse, szabadon változhat. A szépítés után minden egységesztet le kell futtatni, nem csak a megváltozott kódhoz tartozókat, hogy lássuk, a változások okoztak-e hibát. A kódszépítést a szerző és egy tapasztalt programozó végzi közösen.

Az informális felülvizsgálat legfőbb jellemzői:

1. informális, a fejlesztőcsapaton belüli felülvizsgálat,
2. kezdeményezheti a szerző vagy egy tapasztalabb fejlesztő, ritkán a menedzsment,
3. hatékonysága függ az átnéző személyétől, minél tapasztaltabb, annál több hibát vehet észre,
4. célja a korai, költséghatékony hiba felderítés.

#### 6.9.3. [Felülvizsgálat – Átvizsgálás](#)

Az átvizsgálás már egy kicsit formálisabb módja a felülvizsgálatnak. Általában az alkalmazott módszertan előírja, hogy az elkészült kisebb-nagyobb modulokat ismertetni kell a csapat többi tagjával, a többi csapattal. Célja, hogy mások is átlássák az általunk írt kódrészletet (ez csökkenti a kárt, amit egy programozó elvesztése okozhat, lásd kockázatmenedzsment), kritikai megjegyzésekkel segítsék a kód minőségének javítását. Aszerint, hogy hány embernek mutatjuk be az elkészült modult, ezekről beszélhetünk:

1. váll feletti átnézés (angolul: over-the-shoulder review),
2. forráskód átnézése (angolul: code review),
3. kódátvétel (angolul: code acceptance review),
4. körbeküldés (angolul: pass-around),
5. csoportos átnézés (angolul: team review),
6. felületátnézés (angolul: interface review),

## 7. kódprezentálás (angolul: code presentation).

Váll feletti átnézés (angolul: over-the-shoulder review): Az egyik programozó egy ideje nézi saját forráskódját, de nem találja a hibát. Valamelyik kollégáját megkéri, hogy segítsen. Mialatt elmagyarázza a problémát, általában rá is jön a megoldásra. Ha mégsem, akkor a kollégának lehet egy jó ötlete, hogy mi okozhatja a hibát. Általában ennyi elég is a hiba megtalálásához. Ha nem, jöhet a forráskód átnézése.

Forráskód átnézése (angolul: code review): A kód írója megkér egy tapasztalt programozót, hogy segítsen megtalálni egy nehezen megtalálható hibát. Együtt nyomkövetik a programot, miközben a szerző magyarázza, mit miért csinált. Ellenőrzik, hogy a kód megfelel-e a specifikációnak. Ezt addig folytatják, amíg meg nem találják a hibát.

Kódátvétel (angolul: code acceptance review): Az elkészült nagyobb modulokat, pl. osztályokat, a vezető fejlesztő vagy egy tapasztalt programozó átnézi, hogy van-e benne hiba, nem érthető, nem dokumentált rész. A modul fejlesztői elmagyarázzák, mit és miért csináltak. A vezető fejlesztő elmondja, hogyan lehet ezt jobban, szebben csinálni. Ha hibát talál (ez gyakran logikai hiba), akkor arra rámutat, vázolja a javítást.

Körbeküldés (angolul: pass-around): A kód szerzője körbeküldi az általa írt kódrészletet, ami akár egy egész modul is lehet. A címzettek véleményezik a kódot, például megírják, melyik részét érdemes tesztelni. A körbeküldés általában megelőzi a kód felvételét a verziókövető rendszerbe. Általában csak akkor használják, ha egy kódrészlet kritikus fontosságú, pl. egy sokak által használt interfész. Az intenzív kommunikációt előíró módszertanokra (pl. Scrum) nem jellemző.

Csoportos átnézés (angolul: team review): A csoportos átnézés a körbeküldést helyettesíti. Itt is egy érzékeny kódrészletet néznek át többen, de interaktívan. A kódot a szerző prezentálja, sorról sorra magyarázza. Általában elvárás, hogy ha valaki valamit nem ért, azonnal szóljon. A prezentáció végén a vezető programozó elmondja, szerinte mit lehetett volna jobban csinálni. Ehhez is gyakran hozzászólnak a többiek. Több módszertan (pl. extrém programozás) limitálja ezen alkalmak időhosszát fél vagy egy órában.

Felületátnézés (angolul: interface review): Hasonló a csoportos átnézéshez, de itt általában több embernek mutatjuk be azt az interfészt, amelyen keresztül a csoportunk fejlesztése lesz elérhető. Ez azért fontos, hogy az egyes csoportok egyeztetni tudják elvárásaikat egymás felé. Ezeket rögzítik és az integrációs teszt során felhasználják.

Kódprezentálás (angolul: code presentation): Hasonló a csoportos átnézéshez, de az érdekes kódot nem a csoporton belül, hanem a cégen belül mutatjuk be. Akkor gyakori, ha több telephelyen fejlesztik ugyanazt a szoftvert. Nem feltétlenül az egész cég vesz részt benne, lehet, hogy csak három ember, de könnyen előfordulhat, hogy ezek más-más kontinensen vannak. A kódprezentálás célja lehet egy hiba bemutatása, amit egy másik csapat talált és megkéri a kód tulajdonosát, hogy javítsa. Másik gyakori cél a csúcsfejlesztők összehozása, hogy a keretrendszer továbbfejlesztését megbeszéljék.

Az átvizsgálás legfőbb jellemzői:

1. a moderátor maga a szerző, lehet jegyzőkönyvvezető is, de az nem a szerző,

2. a résztvevők a cég alkalmazottai, külső szakértők nem jellemzőek,
3. lehet informális és formális is, ha formális, akkor van pl. jegyzőkönyv,
4. általában az alkalmazott módszertan írja elő vagy a menedzsment kezdeményezi,
5. a szerzők jól felkészülnek, pl. szemléltető ábrákat készítenek, a többi résztvevő átnézi a kapcsolódó dokumentációt,
6. célja az elkészült modulok ismertetése, megértések, azokban hibakeresés.

#### 6.9.4. [Felülvizsgálat – Technikai felülvizsgálat](#)

Technikai felülvizsgálatra általában akkor kerül sor, ha a szoftver teljesítményével nem vagyunk elégedettek. Azt általában könnyű megtalálni a felhasználói visszajelzések és úgynévezett profiler programok segítségével, hogy mi az a szűk keresztmetszet (angolul: bottleneck), ami a lassúságot okozza. Ugyanakkor az nagyon nehéz kérdés, hogy hogyan oldjuk fel ezeket a szűk keresztmetszeteket. Ha lenne egyszerű megoldás, akkor a programozók eleve azt használták volna, tehát ez általában a szoftver cég alkalmazottainak tudását meghaladó probléma.

Ilyenkor külső szakértőket szoktak felkérni, hogy segítsenek. Leggyakrabban egy-egy lekérdezés bizonyul túl lassúnak. Ilyenkor egy index hozzáadása a táblához nagyságrendekkel gyorsítja a lekérdezést. A kérdés már csak az, mit indexeljünk és hogyan. A külsős szakértők átnézik a megoldásunkat és javaslatokat adnak.

Mivel ez a fajta tanácsadás nagyon drága, ezért ez egy jól dokumentált folyamat. A szoftvercég leírja, hogy mi a probléma. Mind a cég alkalmazottai, mind a szakértők felkészülnek, átnézik a dokumentációkat. A megbeszélést általában egy moderátor vezeti, aki jegyzőkönyvet is ír. A moderátor nem lehet a program írója. A résztvevők megbeszélik, hogy mi a probléma gyökere. A szakértők több megoldási javaslatot is adnak. Kiválasztanak egy megoldást. Ezt vagy a szerző, vagy a szakértők implementálják.

A technikai vizsgálat másik típusa, amikor külső szakértők azt ellenőrzik, hogy a szoftver vagy a dokumentációja megfelel-e az előírt szabványoknak. Az ellenőrzést nem a megrendelő, hanem a szoftvercég vagy a szabvány hitelesítését végző szervezet kezdeményezi. Pl. az emberi életre is veszélyes (angolul: life-critical) rendszerek dokumentációjára az IEC61508 szabvány vonatkozik. Ennek betartása a cég érdeke, mert ha kiderül, hogy nem tartja be a szabványt, akkor a termékeit akár ki is vonhatják a piacról.

Akkor is ehhez a technikához fordulnak, ha a szoftverben van egy hiba, amit nagyon nehéz reprodukálni, és a szoftvercég saját alkalmazottai nem tudják megtalálni (megtalálhatatlan hiba). Ez többszálú vagy elosztott rendszereknél fordul általában elő egy holtpont (angolul: deadlock) vagy kiéheztetés (angolul: starvation) formájában, de lehet ez egy memóriaszivárgás (angolul: memory lake) is. Ilyenkor a szakértő megmutatja, hogyan kell azt a statikus elemző szoftvert használni, pl. egy holtpont keresőt (angolul: deadlock checker), ami megtalálja a hibás részt. Az így feltárt hibát általában már a cég szakemberei is javítani tudják.

A technikai felülvizsgálat legfőbb jellemzői:

1. a szoftvercég kezdeményezi, ha külső szakértők bevonására van szüksége,
2. moderátor vezeti (nem a szerző), aki jegyzőkönyvet is vezet,

3. inkább formális, mint informális,
4. a találkozó előtt a résztvevők felkészülnek,
5. opcionálisan ellenőrző lista használata, amit a felek előre elfogadnak,
6. célja a megtalálhatatlan hibák felderítése, vagy a szoftver lassúságát okozó szűk keresztmetszetek megszüntetés, vagy szabványok ellenőrzése.

#### 6.9.5. Felülvizsgálat – Inspekción

Ez a legformálisabb felülvizsgálat. Ezt is akkor használjuk, ha külső szakértő bevonására van szükségünk. A technikai felülvizsgálattól az különbözteti meg, hogy a szoftvercég és a szakértőt adó cég részletesebb szerződést köt, amely magában foglalja:

1. a megoldandó feladat leírását,
2. azt a célfeltételt, ami a probléma megoldásával el kell érni,
3. a célfeltételben használt metrikák leírását,
4. az inspekciós jelentés formáját.

Míg a technikai átnézésnél gyakran csak annyit kérünk a szakértőktől, hogy legyen sokkal gyorsabb egy lekérdezés, az inspekción esetén leírjuk pontosan, hogy milyen gyors legyen.

Az inspekción szó abból jön, hogy a probléma megoldásához általában nem elég csak a szoftver egy részét átvizsgálni, hanem az egész forráskódot adatbázissal együtt inspekción alá kell vonni. Inspekción alkalmazunk akkor is, ha egy régi (esetleg már nem támogatott programozási nyelven íródott) kódöt akarunk szépíteni / átírni, hogy ismét rugalmasan lehessen bővíteni.

Az inspektornak nagy tekintélyű szakembernek kell lennie, mert az általa javasolt változtatások általában nagyon fájóak, nehezen kivitelezhetőek. Ha nincs meg a bizalom, hogy ezekkel a változtatásokkal el lehet érni a célt, akkor a fejlesztőcsapat ellenállásán elbukhat a kezdeményezés.

Az inspektort általában egy-két hónapig is a fejlesztők rendelkezésére áll szemben a technikai felülvizsgálattal, amikor a szakértők gyorsan, akár néhány óra alatt megoldják a problémát. Ezért ugyanannak a szakértőnek a napidíja általában kisebb inspekción esetén, mint technikai felülvizsgálat esetén.

Az inspekción lehet rövid távú is (egy-két hetes), ha a szakértőre nincs szükség a probléma megoldásához, csak a feltárásához. Ekkor a szakértő egy inspekciós jelentést ír, amely leírja, hogyan kell megoldani a problémát. Ehhez általában csatolni kell egy példaprogramot is, egy úgynévezett PoC-t (angolul: Proof of Concept), amely alapján a cég saját fejlesztői is képesek megoldani a problémát. A „pocok”-nak demonstrálnia kell, hogy a kívánt metrika értékek elérhetőek a segítségével.

Az inspekción legfőbb jellemzői:

1. a szoftvercég kezdeményezi, ha hosszabb távon van szüksége külső szakértőre,
2. részletes szerződés szabályozza, ami a problémát, a célfeltételt és célban szereplő metrikákat is leírja,
3. opcionálisan „pocok” (angolul: Proof of Concept) készítése,
4. inspekciós jelentés készítése,

5. célja: teljesítményfokozás a szakértő által kiválóan ismert technológia segítségével vagy elavult kód frissítése.

#### 6.9.6. [Statikus elemzés - Bevezetés](#)

A statikus elemzés fehérdobozos teszt, hiszen szükséges hozzá a forráskód. Néhány esetben, pl. holtpontellenőrzés, elegendő a lefordított köztes kód (byte kód). A statikus elemzés azért hasznos, mert olyan hibákat fedez fel, amiket más tesztelési eljárással nehéz megtalálni. Például kiszűrhető segítségével minden null referencia - hivatkozás, ami futási hibához vezethet, ha benne marad a programban. Az összes null referencia - hivatkozás kiszűrése dinamikus technikákkal (pl. komponensteszttel vagy rendszerteszttel) nagyon sok időbe telne, mert 100%-os kódlefedettséget kellene elérnünk.

A statikus elemzés azt használja ki, hogy az ilyen tipikus hibák leírhatók egyszerű szabályokkal, amiket egy egyszerű kódelemző (parser) gyorsan tud elemezni. Például null referencia - hivatkozás akkor lehetséges, ha egy „a = null;” értékadó utasítás és egy „a.akárm;” hivatkozás közt van olyan végrehajtási út, ahol az „a” referencia nem kap null-tól különböző értéket. Ugyan ezt lehet dinamikus technikákkal is vizsgálni, de ahhoz annyi tesztesetet kell fejleszteni, ami minden lehetséges végrehajtási utat tesztel az „a = null;” és az „a.akárm;” közt.

A forráskód statikus elemzésnek két változata ismert, ezek:

1. statikus elemzés csak a forráskód alapján,
2. statikus elemzés a forráskód és modell alapján.

Ezen túl lehetséges a dokumentumok statikus elemzése is, de ezekre nem térünk ki.

A következő hibatípusokat könnyebb statikus elemzéssel megtalálni, mint más technikákkal:

1. null referenciára - hivatkozás,
2. tömbök túl- vagy alulindexelése,
3. nullával való osztás,
4. lezáratlan adatfolyam (angolul: unclosed stream),
5. holtpont (angolul: deadlock),
6. kiéheztetés (angolul: starvation).

Az egyes eszközök lehetnek specifikusak, mint pl. a holtpontkeresők, illetve általánosak, mint pl. a FindBugs.

#### 6.9.7. [Statikus elemzés csak a forráskód alapján](#)

Azok az elemzők, amelyek csak a forráskódot használják fel az elemzéshez, nagyon hasznosak olyan szempontból, hogy nem igényelnek plusz erőfeszítést a programozóktól a specifikáció megírásához. Ilyen eszköz például a FindBugs. Ezeket az eszközöket csak bele kell illeszteni a fordítás folyamatába. Ezután a statikus elemző felhívja a figyelmünket a tipikus programozói hibákra. Ezek általában programozásnyelv-specifikusak, de léteznek nyelvfüggetlenek, pl. a Sonar vagy a Yasca rendszer, amelyek egy-egy plugin segítségével adaptálhatóak a kedvenc nyelvünkhez.

Jelen jegyzetben a FindBugs használatát fogjuk bemutatni Eclipse környezetben. Először telepíteni kell a FindBugs plugint. Ehhez indítsuk el az Eclipse rendszert, majd válasszuk a Help -> Install New Software... menüt. A megjelenő ablakban adjuk hozzá a plugin források listájához az alábbi linket az Add gombbal: <http://findbugs.cs.umd.edu/eclipse>. Ezután néhány Next gomb és a felhasználási feltételek elfogadása után a rendszer elkezdi installálni a FindBugs plugint. Ez néhány percet vesz igénybe, ami után újraindul az Eclipse. Ezután már használhatjuk a FindBugs-t.

A használatához válasszunk ki egy projektet, majd a helyi menüben válasszuk a Find Bugs -> Find Bugs menüt. Ez egyrészt megkeresi azokat a sorokat, amelyek valamilyen szabálynak nem felelnek meg, másrészt átvisz minket a FindBugs perspektívába. Ha talál olyan sorokat, amelyek potenciálisan hibát okozhatnak, akkor ezeket bal oldalon egy kicsi piros bogárikonnal jelzi. Ha ezekre ráállunk vagy rákattintunk, akkor láthatjuk, milyen típusú hibát okozhat. Ezekről részletes információt is kérhetünk, ha a FindBugs perspektíva Bug Explorer ablakában kiválasztjuk valamelyiket.

Az egyes hibák ellenőrzését ki/be lehet kapcsolni a projekt Properties ablakának FindBugs panelén. Itt érdemes a Run automatically opciót bekapcsolni. Így minden egyes mentésnél lefut a FindBugs. Ebben az ablakban az is látható, melyik hiba ellenőrzése gyors, illetve melyik lassú. Például a null referenciára-hivatkozás ellenőrzése lassú.

Nézzünk néhány gyakori hibát, amit megtalál a FindBugs az alapbeállításaival:

```
public int fact(int n) { return n * fact(n - 1); }
```

Itt a „There is an apparent infinite recursive loop” figyelmeztetést kapjuk nagyon helyesen, hiszen itt egy rekurzív függvényt írtunk bázisfeltétel nélkül, és így semmi se állítja meg a rekurziót.

```
Integer i = 1, j = 0; if(i == j) System.out.println("ugyanaz");
```

Ebben a példában a „Suspicious comparison of Integer references” figyelmeztetést kapjuk. Ez azért van, mert referenciák egyenlőségét ugyan tényleg a dupla egyenlőségjellel kell vizsgálni, de a mögöttük lévő tartalom egyenlőségét az equals metódussal kell megvizsgálni. Tehát ez egy lehetséges hiba, amit érdemes a fejlesztőknek alaposan megnézni.

```
int i = 0; i = i++; System.out.println(i);
```

Itt több hibát is kapunk: „Overwritten increment” és „Self assignment of local variable”. Az első hiba arra utal, hogy hiába akartuk növelni az i változó értékét, az elvész. A második hiba azt fejezi ki, hogy egy változót önmagával akarunk felülírni.

Nézzünk olyan esetet is, aminél hibásan ad figyelmeztetést a FindBugs:

```
Object o; int i = 1; if (i == 1) { o = "hello"; } System.out.println(o.toString());
```

A fenti esetre a „Possible null pointer dereference of o” hibát kapjuk, habár egyértelműen látszik, hogy az o értéket fog kapni, hiszen igaz az if utasítás feltétele. Ugyanakkor a FindBugs rendszer nem képes kiszámolni a változók lehetséges értékeit az egyes ágakon, hiszen nem tartalmaz egy automatikus tételezetbonyítót. Ezzel szemben a következő alfejezetben tárgyalta ESC/Java2 eszköz képes erre, hiszen egy automatikus tételezetbonyítóra épül.

#### 6.9.8. Statikus elemzés a forráskód és modell alapján

Ebben az esetben a forráskód mellett van egy modellünk is, ami leírja, hogyan kellene működnie a programnak. A program viselkedése ilyen esetben elő- és utófeltételekkel, illetve invariánsokkal van leírva. Ezt úgy érjük el legkönnyebben, hogy szerződésalapú programozást (angolul: Design by Contract) használunk. Ez esetben minden metódusnak van egy szerződése, amely a metódus elő- és utófeltételében ölt testet. A szerződés kimondja, hogy ha a metódus hívása előtt igaz az előfeltétele, akkor a metódus lefutása után igaznak kell lennie az utófeltételének. Az invariánsok általában osztályszintűek, leírják az osztály lehetséges belső állapotait. A program viselkedését legegyszerűbben assert utasításokkal írhatjuk le.

Egy példa assert használatára:

```
public double division(double a, double b) { assert(b != 0.0); return a / b; }
```

A fenti példában azt feltételezzük, hogy a metódus második paramtere nem nulla. A program kódjában a feltételezéseinket assert formájában tudjuk beírni Java esetén. Java esetén az assert utasítások csak akkor futnak le, ha a JVM-et a –enableassert vagy az egyenértékű –ea opcionális opciót futtatjuk, egyébként nincs hatásuk.

C# esetén a fenti példa megfelelője ez:

```
public double Division(double a, double b)
{
    System.Diagnostics.Debug.Assert(b != 0.0);
    return a / b;
}
```

Az Assert csak akkor fog lefutni, ha a Debug módban fordítjuk az alkalmazást.

A program viselkedését legegyszerűbben assert utasításokkal lehet leírni, de lehetőségünk van magas szintű viselkedésleíró nyelvek használatára, mint például a JML (Java Modeling Language) nyelv. Ez esetben magas szintű statikus kód ellenőrzést (angolul: Extended Static Checking, röviden: ESC) tudunk végezni az ESC/Java2 program segítségével.

Egy példa JML használatára:

```
public class BankSzámla {
    private /*@ spec_public */ int balansz = 0;
    private /*@ spec_public */ boolean zárolt = false;
    //@ public invariant balansz >= 0;
    //@ requires 0 < összeg;
    //@ assignable balansz;
    //@ ensures balansz == \old(balansz) + összeg;
    public void betesz(int összeg) { balansz += összeg; }
    //@ requires 0 < összeg && összeg <= balansz;
    //@ assignable balansz;
    //@ ensures balansz == \old(balansz) - összeg;
    public void kivesz(int összeg) { balansz -= összeg; }
    //@ assignable zárolt;
    //@ ensures zárolt == true;
    public void zárol() { zárolt = true; }
    //@ requires !zárolt;
    //@ ensures \result == balansz;
```

```

//@ also
//@ requires zárolt;
//@ signals_only BankingException;
public /*@ pure @*/ int getBalansz() throws BankingException {
    if (!zárolt) { return balansz; }
    else { throw new BankingException("Zárolt a számla"); }
}
class BankingException extends Exception {
    public BankingException(String msg) { super(msg); }
}

```

Ebből a kis példából lehet látni, hogy a JML specifikációt megjegyzésbe kell írni, amely az @ jellel kezdődik. A spec\_public kulcsszóval teszünk láthatóvá egy mezőt a JML specifikáció számára. Az invariant kulcsszó után adjuk meg az osztály invariánsát, amelynek minden (nem helper) metódushívás előtt és után igaznak kell lennie. Az előfeltételt a requires kulcsszó után kell írni. Maga a feltétel egy szabályos Java logikai kifejezés. A kifejezésben lehet használni JML predikátumokat is. Az utófeltétel kulcsszava az ensures. Lehet látni, hogy az utófeltételekben lehet hivatkozni a visszatérési értékre a \result JML kifejezéssel. Az \old(x) JML kifejezés az x változó metódus futása előtti értékére hivatkozik. Az assignable kulcsszó segítségével úgynevezett keretfeltétel (angolul: frame condition) adható, amiben felsorolhatom, hogy a metóduson belül mely mezők értékét lehet megváltoztatni. Ha egyik mező értékét sem változtathatja meg a metódus, akkor azt mondjuk, hogy nincs mellékhatalma. Az ilyen metódusokat a pure kulcsszóval jelöljük meg. Elő- és utófeltételekben csak pure metódusok hívhatók. Az also kulcsszó esetszétválogatásra szolgál. A signals\_only kulcsszó után adható meg, hogy milyen kivételt válthat ki a metódus.

A fenti példában van egy BankSzámla osztályunk, amelyben a balansz mező tárolja, hogy mennyi pénzünk van. Az invariánsunk az fejezi ki, hogy a balansz nem lehet negatív. A fenti 4 metódus JML leírását így lehetne átfordítani magyarára:

- betesz(összeg)
  - Előfeltétel: Az összeg pozitív szám, mert nulla forintot nincs értelme betenni, negatív összeget pedig nem szabad.
  - Keretfeltétel: Csak a balanszmezőt írhatja.
  - Utófeltétel: A balanszot meg kell növelni az összeggel, azaz az új balansz a régi balansz plusz az összeg.
- kivesz(összeg)
  - Előfeltétel: Az összeg pozitív szám, mert nulla forintot nincs értelme kivenni, negatív összeget pedig nem szabad. Továbbá az összeg kisebb egyenlő, mint a balansz, mert a számlán lévő összegnél nem lehet többet felvenni.
  - Keretfeltétel: Csak a balanszmezőt írhatja.
  - Utófeltétel: A balanszot csökkenteni kell az összeggel, azaz az új balansz a régi balansz minusz az összeg.
- zárol()
  - Előfeltétel: Nincs, azaz minden igaz.
  - Keretfeltétel: Csak a zároltmezőt írhatja.
  - Utófeltétel: A zároltmezőnek igaznak kell lennie.

- `getBalansz()`, két esetet különböztetünk meg, ahol az előfeltételek kizárták egymást.
  - Előfeltétel 1.: A számla nem zárolt.
  - Utófeltétel: A visszatérési érték megegyezik a balansz értékével.
  - Előfeltétel: A számla zárolt.
  - Kivétel: A zárolt számla nem kérdezhető le, ezért `BankingException` kivételt kell dobni.
  - Keretfeltétel: Mindkét esetben egyik mező sem írható, tehát ez a metódus „pure”.

A JML nyelvhez több segédesköz is létezik. Az első ilyen az Iowa State University JML program. Ez a következő részekből áll:

- `jml`: JML szintaxisellenőrző
- `jmlc`: Java és JML fordító, a Java forrásban lévő JML specifikációt belefordítja a bájtkódba.
- `jmlrac`: a `jmlc` által instrumentált bájtkódot futtató JVM, futtatás közben ellenőrzi a specifikációt, tehát dinamikus ellenőrzést végez.

Nekünk a JML 5.4 és 5.5 verziót volt szerencsénk kipróbálni. Sajnos ezek csak a Java 1.4 verzióig támogatják a Java nyelvet. Nem ismerik például a paraméteres osztályokat. Ha simán hívjuk meg a `jmlc` parancsot, akkor rengeteg információt kiír, ami esetleg elfed egy hibát. Ezért érdemes a `-Q` vagy a `-Quite` opcióval együtt használni. A `BankSzámla` példát a következő utasításokkal lehet ellenőrizni, hogy megfelel-e specifikaciójának:

```
jmlc -Q BankSzámla.java
```

```
jmlrac BankSzámla
```

Persze ehhez a `BankSzámla` osztályba kell írni egy `main` metódust is, hiszen az a belépési pont.

Második példa:

```
public abstract class AbstractAccount {
    //@ public model int balance;
    //@ public invariant balance >= 0;
    //@ requires amount > 0;
    //@ assignable balance;
    //@ ensures balance == \old(balance + amount);
    public abstract void credit(int amount);
    //@ requires 0 < amount && amount <= balance;
    //@ assignable balance;
    //@ ensures balance == \old(balance) - amount;
    public abstract void debit(int amount);
    //@ ensures \result == balance;
    public abstract /*@ pure */ int getBalance();
}

class Account extends AbstractAccount {
    private /*@ spec_public */ int balance = 0; //@ in super.balance;
    //@ private represents super.balance = balance;
    public void credit(int amount) { balance += amount; }
    public void debit(int amount) { balance -= amount; }
    public int getBalance() { return balance; }
}
```

Ez a példa azt mutatja meg, hogyan lehet már az absztrakt ōs osztályban specifikálni az elvárt viselkedést. Ehhez egy modell mezőt kell definiálni az ōsben (vagy az interfészben) a „model” kulcsszóval. A konkrét gyermekben az ōsben specifikált viselkedést meg kell valósítani. Ehhez meg kell mondani, hogy melyik konkrét mező valósítja meg a modell mezőt. Ez a „represents” kulcsszó használatával lehetséges.

A fenti példát a következő utasításokkal lehet ellenőrizni:

```
jmlc -Q bank3/*.java
```

```
jmlrac bank3.Account
```

Persze ehhez az Account osztályba kell írni egy main metódust is, hiszen az a belépési pont.

Harmadik példa:

```
public interface Timer {
    //@ public instance model int ticks;
    //@ public invariant ticks >= 0;
    //@ assignable this.ticks;
    //@ ensures this.ticks == ticks;
    void setTimer(int ticks);
}
class Dish implements Timer{
    private /*@ spec_public */ int timer; //@ in ticks;
    //@ private represents ticks = timer;
    public void setTimer(int timer) { this.timer = timer;}
}
```

Ez a példa azt mutatja meg, hogyan kell modell mezőt létrehozni az interfészben. Mindent ugyanúgy kell csinálni, csak a „model” kulcsszó elő be kell írni az „instance” kulcsszót, ami azt fejezi ki, hogy a modell változó példányszintű. Erre azért van szükség, mert egyébként Javában minden interfész mező statikus.

Láttuk, hogy a specifikáció dinamikusan ellenőrizhető az Iowa State University JML programmal. Szerencsére lehetséges a statikus ellenőrzés is az ESC/Java2 programmal.

Az ESC/Java2 (Extended Static Checker for Java2) egy olyan segédeszköz, amely ellenőrizni tudja, hogy a Java forrás megfelel-e a JML specifikációnak. Az ESC/Java2 hasonlóan a FindBugs programhoz figyelmezteti a programozót, ha null referenciára hivatkozik, vagy más gyakori programozói hibát vét. Erre akkor is képes, ha egy JML sor specifikációt se írunk. Nyilván, ha kiegészítjük a kódunkat JML specifikációval, akkor sokkal hasznosabb segédeszköz.

Az ESC/Java2 program is csak a Java 1.4 verziójáig támogatja a Java nyelvet. Ez is telepíthető Eclipse pluginként a <http://kind.ucd.ie/products/opensource/Mobius/updates/> címről.

Miután feltelepítettük, két új perspektívát kapunk, a Validation és a Verification nevűt. Az elsőben 3 új gombban bővül a menüsor alatti eszköztár. Ezek a következők: JML, JMLC, és a JMLRAC gomb, amelyek az azonos nevű segédprogramot hívják az Iowa State University JML programcsomagból.

A második perspektívában 5 új gombot kapunk. Ezek közül a legfontosabb az első, amely elindítja a ESC/Java2 ellenőrző programot. A többi gomb balról jobbra haladva a következők: jelölők (jelölőknek

nevezzük a hiba helyét jelölő piros íkszett) törlése, ugrás jelölőre, ellenőrzés engedélyezése, ellenőrzés tiltása. Ezeket nem találtuk különösebben hasznosnak. Ami hasznos volt számunkra, az az ESC/Java2 menüben található Setup menü. Itt lehet bekapcsolni az automatikus ellenőrzést, aminek hatására minden egyes mentés után lefut az ESC/Java2.

Nézzünk egy egyszerű példát, amikor JML specifikáció nélkül is hibát fedez fel a kódunkban az ESC/Java2.

```
package probe; public abstract class Decorator extends Car { Car target; public int getSpeed(){ return target.getSpeed(); }
```

Itt az ötödik sorra azt a hibát kapjuk, hogy „Possible null dereference (Null)”. Ez a figyelmeztetés teljesen jogos, segíti a programozót egy hiba kijavításában.

Nézzük meg azt a példát, amivel a FindBugs nem boldogult:

```
public static void main(String[] args){  
    Object o = null; int i = 1; if(i == 1) o = "hello";  
    System.out.println(o.toString());  
}
```

Erre az ESC/Java2 semmilyen hibát nem ad. Ez azért lehetséges, mert mögötte egy automatikus tételezbizonyító áll, ami meg tudja nézni, hogy valamely feltétel igaz vagy sem az egyes lehetséges végrehajtási utakon.

Ugyanakkor a ESC/Java2-höz adott Simplify nevű automatikus tételezbizonyító nem túl okos. Például nem tudja, hogy két pozitív szám szorzata pozitív, ezért ad hibát a következő példára:

```
public class Main {  
    //@ requires n>=0;  
    //@ ensures \result > 0;  
    public int fact(int n){ if (n==0) return 1; return n*fact(n-1); }  
}
```

Itt az ESC\Java2 hibásan a „Postcondition possibly not established (Post)” figyelmeztetést adja, pedig a függvény tökéletesen betartja az utófeltételét. Szerencsére az ESC\Java2 alatt kicserélhető az automatikus tételezbizonyító.

## 6.10. Egyéb fogalmak

### 6.10.1. DoD – Definition of Done

Definition of Done (teljesítés definíciója) a Scrum módszertanban használt kifejezés. minden sprint előtt meghatározzák, hogy mikor tekintik befejezettnék a sprintet. Általában is használható egy feladat sikeres befejezésének pontos meghatározására. Iteratív módszertanoknál a DoD gyakran az, hogy minden iteráció végén a regressziós tesztek 100%-ának sikeresnek kell lennie.

### 6.10.2. POCOK

Proof of Concept (feltevés helyességének ellenőrzése) vagy más néven POCOK egy olyan, az implementációt, de gyakran a rendszerterv készítését megelőző lépés, amikor teszteljük, hogy egy

technológia alkalmas-e egy jól körülírt probléma megoldására. A cél általában egy költséghatékony, vagy esetleg ingyenes komponens tesztelése, hogy érdemes-e használni, illetve melyiket. A POCOK eredménye vagy az, hogy érdemes a külső komponenst / technológiát használni, vagy az, hogy inkább érdemes saját magunknak kifejleszteni.

#### 6.10.3. [Feature freeze](#)

Általában a tesztelést megelőző lépés. Tesztelés alatt tilos új funkciót adni a rendszerhez, csak a megtalált hibákat szabad javítani. Ez hasznos, mert egy új funkció már működő kód részek átírását is gyakran igényli, ami váratlan hibákhoz vezethet.

#### 6.10.4. [Code freeze](#)

Általában a tesztelést vagy a rendszerátadást megelőző lépés, ami letiltja a kód változtatását. Ez alatt nem szabad változtatni a kódot, vagy csak a vezető programozónak. Nagyobb programokban hasznos, amikor mindenki a kimenő verzióhoz szükséges utómunkát végzi, például telepítőcsomagot készít.

## 7. Rendszerfejlesztés technológiája - Eszközök

Ez a fejezet az előző folytatása, az olvashatóság érdekében a rendszerfejlesztést segítő eszközöket ebben a külön fejezetben emeltük ki.

### 7.1. Verziókövetés

A verziókövetés ugyan közvetetten magába foglalja egy termék 1.2 és 1.3 verziója közötti eltérések követését is, de alapvetően nem erről szól. Erre a whatsnew.txt a megfelelő eszköz. A verziókövetés ennél lényegesen részletesebb és bonyolultabb. A verziókövető rendszerek képesek állományok tartalmi változásait követni, figyelembe véve, hogy ki és mikor módosította azokat, valamint korábbi állapotokat is képes előállítani.

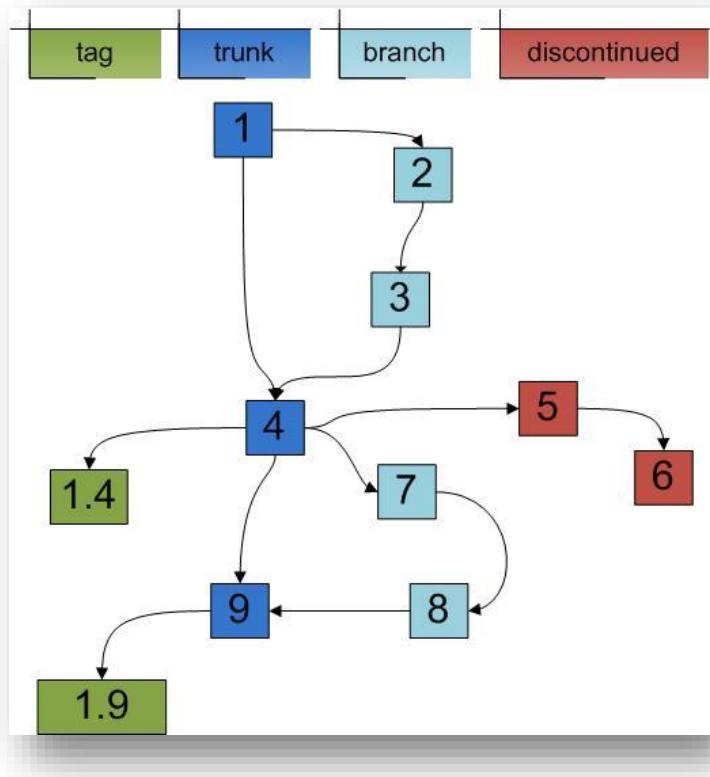
Képzeljük el, hogy dolgozunk egy nagyobb szabású munkán (nem, azért sem írom le, hogy projekt), ami számos állományból áll, amik folyamatosan változhatnak. Megkapunk egy részfeladatot, el is készítjük rögtön az első változatot, ami szívünknek felettesebb kedves, aztán javítgatunk rajta, mert rájövünk, hogy nem egészen jó még az. Később a projektmenedzser megmondja, hogy ez bizony nem azt csinálja, amit kellene, és inkább menjünk krumplit kapálni, de minimum írjuk meg azt, amit kért tölünk. Némi szenvedés után rájövünk, hogy de hiszen azt csináltuk elsőre is, csak javítottunk rajta. Ebben az esetben igen áldásos lenne, ha a forrásunk (rajzunk, zenénk, akármink) egy korábbi állapotát valahonnan elő tudnánk húzni. Az alkotói hévbe belefeledkezett programozónak szinte biztos, hogy nem jut eszébe minden korszakalkotó i++; után bezippelve, felsorszámozva és kommentezve elmenteni az aktuális állományokat, ami sok időt igénylő unalmas rabszolgamunka.

Arról nem is szólva, hogy a többiekkel is meg kellene osztani azt a kódot, ami épp működőképes. Ha mindehhez hozzávesszük, hogy nem csak egyedül dolgozunk a munkán, akkor aztán keresgélhetünk, hogy hol is van a legutóbbi stabil változat. Szóval elégé problémás lenne a folyton változó és több személy változtatta kódot felügyelni, rendszerezni. A verziókövető rendszerek ebbe a káosz és fejfájás sújtotta övezetbe hoztak hűs, árnyat adó megoldást, ahol a programozók árnyas pálmafák alatt szürcsölik a kókuszos koktélt.

Röviden összefoglalva a verziókövető rendszerek lehetővé teszik, hogy:

- A verziókövetés alá helyezett minden állomány minden korábbi változatát vissza lehet keresni.
- Egy állomány tetszőleges két változata közötti eltéréseket meg tudja jeleníteni.
- Ha ketten egyszerre módosítják ugyanazt az állományt, akkor lehetőség szerint mindkettőjük módosításai megfelelően rögzítésre kerüljenek. Amennyiben a két módosítás ütközne egymással, akkor a későbbi módosítást végző személy értesüljön a konfliktusról és döntenи tudjon arról, hogy milyen tartalom kerüljön be végül az állományba.
- Elágazásokat hozhatunk létre. Tegyük fel, hogy az alap szoftverünkbeli készítünk egy általános iskoláknak és egy középiskoláknak szánt verziót is, akkor ezeket külön ágra (angolul: branch) helyezhetjük. A későbbiekben ezeket külön-külön fejleszthetjük.

Többféle verziókövető rendszer is létezik. A teljesség igénye nélkül RCS, CVS, Subversion, SCCS, OpenCM a központosítottak közül. Az elosztott rendszerek, például: Aegis, Arch, Codeville, Monotone, Git, SVK. Ezek nyílt forráskódúak. De a „nagyok” is előálltak saját változataikkal: IBM CMVC, Microsoft SourceSafe.



31. ábra: Egy verzió fa

#### 7.1.1. Közös szókincs

Különböző rendszerek eltérő kifejezéseket használhatnak ugyanazon eszközökre, de szerencsére vannak általánosan elterjedt szakkifejezések.

- Ág (angolul: branch): A verziókövetés alá helyezett állományok egy részhalmazának fejlesztése más irányt vehet, akár több irányba is elágazhatnak. Kézenfekvő az „ág” kifejezés. Egy ágban tudunk például kísérletezni a projekttel, próbálkozni bizonyos elméletek megvalósíthatóságával, esetleg az egyes kiadott verziók hibajavításait végezhetjük itt. Végül dönthetünk úgy, hogy nem folytatjuk ebben az ágban a fejlesztést, vagy akár úgy is, hogy összefűsljük a fő vonallal az ágban történt változtatásokat.
- Lekérés (angolul: check-out): Lokális másolat készítése egy verziókövetett állományról.
- Beküldés (angolul: commit): A lokális állomány beküldése a szerverre.
- Ütközés (angolul: conflict): Ha úgy akarnak többben is megváltoztatni egy állományt, hogy a rendszer nem képes a változások önálló összefűzésére.
- Exportálás (angolul: export): Olyan lokális másolat készítése, ami nélkülözi a verziókövetéshez szükséges metaadatokat.
- Head: A legutolsó beküldött változat.
- Importálás (angolul: import): Lokálisan tárolt adathalmazt lehet beküldeni a tárolóba és verziókövetés alá helyezni.
- Összefűsülés (angolul: merge): Két változtatáslistát lehet vele összefűsülni egy közös verzióból.

- Tároló (angolul: repository): Ez általában valamelyen verziókövető szerveren található. Ide másolódnak az adataink, itt lehet megtalálni az aktuális és korábbi verzióinkat.
- Verzió (angolul: version, revision): Ez egy sorszám. Azt mutatja meg, hány beküldés történt eddig az adott állományból.
- Címke (angolul: tag, label): Ez az, amit a felhasználók általában verziószámként ismernek, de nem törvényszerű, hogy csak az lehet. Tartalmazhat bármilyen fontos rövid megjegyzést is a hozzá tartozó állománycsoport megjelölésére. A címke lehet annak a könyvtárnak a neve is, ahol pl. az alkalmazásunk hivatalosan kiadott verziója megtalálható.
- Törzs (angolul: trunk): A fejlesztés fő vonala.
- Frissítés (angolul: update): A tárolóban található új módosítások letöltése a munkamásolatba.
- Munkamásolat (angolul: working copy): A tároló állományainak másolata a lokális gépen. Itt lehet garázdálkodni (hivatalos terminológia szerint: dolgozni), és ha valamit nagyon elrontottunk, még minden lekérhetjük a tárhelyről a legutolsó változatot.

### 7.1.2. Subversion

A Subversion igen elterjedt nyílt forráskódú verziókövető rendszer, jelenleg AIX, Debian, Mac OS X, Solaris, Ubuntu és Windows rendszerekre is létezik, de ez a lista korántsem teljes és minden bizonnal bővülni fog. A dokumentációt elolvashatjuk, és az egyes változatok forrását, vagy akár a lefordított állományokat letölthetjük a <http://subversion.apache.org/> címről.

A Subversion használatához két alkalmazásra van szükségünk. Kissé sem meglepő módon az egyik a szerver, a másik pedig a kliens. A szervert többféle módon is beszerezhetjük. Például az újabb Linux disztribúciók sok esetben tartalmazzák a szervert, de ha nem, akkor is egyszerű

`apt-get install`

vagy

`rpm -i`

utasításokkal odavarázsolhatjuk a mit sem sejtő operációs rendszer alá. Windows alatt annyival egyszerűbb a helyzet, hogy az igen egyszerű Next-Next-Finish módszerrel tehetünk szert a VisualSVN szolgáltatásaira. Természetesen a telepítési útvonalat és a tárolók helyét meg kell adni, ugyanúgy, mint a portot is, amin a szerver elérhető lesz.

Mint látni fogjuk, az SVN szerver nem más, mint egy speciális Apache szerver. Akár böngészőn keresztül is elérhetjük, de az igazán hatékony alkalmazása az, ha valamelyen SVN klienst használunk. Több fejlesztő környezetnek is részévé lehet tenni, mint például az Eclipse is használhatja beépülő modulként. Windows alatt pedig a TortoiseSVN kliens beépül a keretrendszerbe, így egy könyvtáron jobbklikkel elérhetjük az összes gyakran használt SVN funkciót. A továbbiakban a VisualSVN Server (<http://www.visualsvn.com/>) és a TortoiseSVN (<http://tortoisessvn.tigris.org/>) használatát mutatjuk be Windows operációs rendszer alatt.

#### 7.1.2.1. VisualSVN szerver

A VisualSVN szerver telepítése után ugyan generál egy SSL tanúsítványt, de ez nem fogja elnyerni semmiféle SVN kliens vagy böngésző bizalmát, tehát ha lehetőségünk és szükségünk van rá, akkor állítsunk be ide ismert tanúsítvány szolgáltatótól beszerzett SSL tanúsítványt. Alapértelmezés

szerint a VisualSVN az alap hitelesítést használja, azaz felhasználónevet és jelszót kér a kliensektől a csatlakozáshoz, de ezt akár a telepítéskor, akár a későbbiekben átállíthatjuk Windows hitelesítésre. Ajánlott a HTTPS protokoll használata. Telepítésre kerül a VisualSVN Server Manager is, ami lényegesen megkönnyíti a szerveren végezhető alapvető beállításokat. A továbbiakban minden grafikus felületre történő utalás a VisualSVN Server Manager-re vonatkozik. A grafikus felületen a VisualSVN szerver alatt alap esetben három bejegyzést láthatunk: Repositories, Users, Groups. Az első a szerveren létrehozott tárházakat tartalmazza, a második a felhasználókat és a harmadik a felhasználók csoportjait. A felhasználó és csoportkezelés triviális. Adhatunk írási, olvasási jogokat, és a felhasználókat csoportokba rendezhetjük. Az SVN szerver bin könyvtárában található svnadmin.exe segítségével parancssorban is elvégezhetjük a kívánt műveleteket.

#### 7.1.2.2. Tárház létrehozása

A Repositories részen jobb gombot nyomva új tárházat hozhatunk létre. Ez lehet egy üres könyvtár, de létrehozhatjuk az ajánlott könyvtárszerkezetet is, ami a branches, tags és trunk könyvtárakból áll. Cégen, de legalábbis részlegén belül érdemes egyetlen tárházat létrehozni és azon belül elhelyezni a projekteket, miáltal kevesebb időt kell tölteni a karbantartással, és a projektek közötti adatmozgatások is könnyebbé válnak, anélkül, hogy az egyes állományok története, azaz a változási napló bejegyzései eltűnnének valami tér-idő anomáliában. Azért is érdemes egyetlen tárházat létrehozni, mert ellenkező esetben a copy, diff és merge parancsokat nem tudjuk végrehajtani. A tárház könyvtárszerkezetének kialakítása nagyobb munkák esetében jól átgondolt tervezést igényel, ehhez (és a mögöttes adatbázis megválasztásához) hasznos ötleteket találhatunk a <http://www.visualsvn.com/support/svnbook/reposadmin/planning/> címen. Miután létrejött a tárház, az elérést biztosítanunk kell a felhasználóknak. Ha készen vagyunk a jogosultságok kiosztásával, a tárházon jobb gombot nyomva a Copy URL to Clipboard menüpont segítségével könnyen megszerezhetjük azt az URL-t, amit a kliensek használnak majd a szerverhez csatlakozáshoz. Egy tárházat parancssorban is létrehozhatunk az

```
svnadmin create <útvonal>
```

parancs segítségével, ahol az útvonal a létrehozandó tárház helyét adja meg. A tárház létrejöttéről és elérhetőségről meggyőződhetünk úgy is, hogy a tárház címét (például: <https://firg.hu:1200 svn/tarhaz>) bemásoljuk egy böngésző címsorába. A név és a jelszó megadása után látnunk kell a tárház könyvtárszerkezetét. A parancsot preferálók a következő parancs kiadásával tehetik meg ugyanezt:

```
svn ls https://firg.hu:1200 svn/tarhaz
```

Ezek után túl sok dolgunk nem is lesz a szerverrel, de néha azért készítünk biztonsági mentést. Az ne töltön a hamis biztonság érzetével, hogy legalább annyi másolatunk van a tárházunkról, ahány felhasználó dolgozik rajta, mert éppen annyi működésképtelen változatunk is lehet.

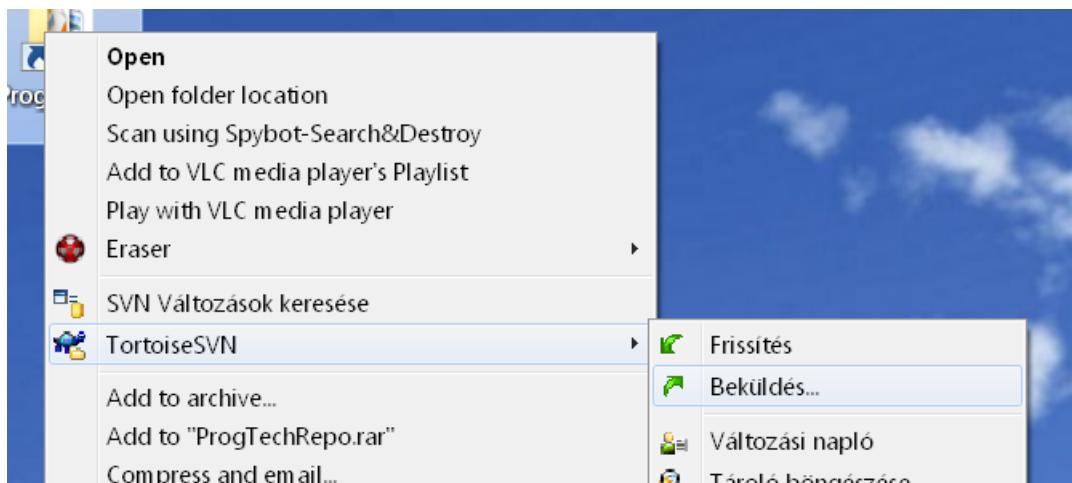
#### 7.1.2.3. VisualSVN kliens

Visual Studio beépülő modul. Fizetős. Lényegében ugyanazt a menürendszert építi be a Visual Studio-ba, mint a TortoiseSVN kliens menürendszere. Többet nem nyújt, így megkérdőjelezhető, hogy van-e értelme ezt használni a TortoiseSVN-nel szemben.

#### 7.1.2.4. Lekérés, frissítés, beküldés

Létező tárházhöz csatlakozni a következőképpen tudunk. Először is hozunk létre egy könyvtárat, amiben a munkamásolatunk foglal majd helyet. Ezen a könyvtáron jobb gombot nyomva a TortoiseSVN menüben a Lekérést (angolul: Check out) választva lehetőségünk lesz megadni a tárház URL-t, azaz példánknál maradva a <https://firg.hu:1200/svn/tarhaz> címet. Lehetőségünk nyílik a célkönyvtár módosítására. Ha minden mást alapértelmezetten hagyunk, akkor megkapjuk a tárház tartalmát. Példánkban ez kizárolag az svn üzemeléséhez szükséges szerkezetet tartalmazza, de ha már egy futó munkához csatlakozunk, akkor természetesen megkapjuk a már meglévő állományokat, az összes előzménnyel. Erre a műveletre gyakran használt kifejezés a „kihúzom a repót”, és ha már mindenéppen tapasztaltnak szeretnénk látszani, használjuk csak nyugodtan. Arra is lehetőségünk van, hogy ne a teljes tárházat töltök le a gépünkre, hanem annak csak egy részét. Természetesen csak azokat a részeket tölthetjük le, amikre jogosultságot kaptunk.

A projektünk könyvtárszerkezete kiegészül az állományok változását nyomon követő adatokkal. Ezeket az adatokat a minden egyes alkönyvtárban létrehozott .svn könyvtár tartalmazza. Ezt nem érdemes bolygatni, hagyjuk csak az SVN-re a kezelését.

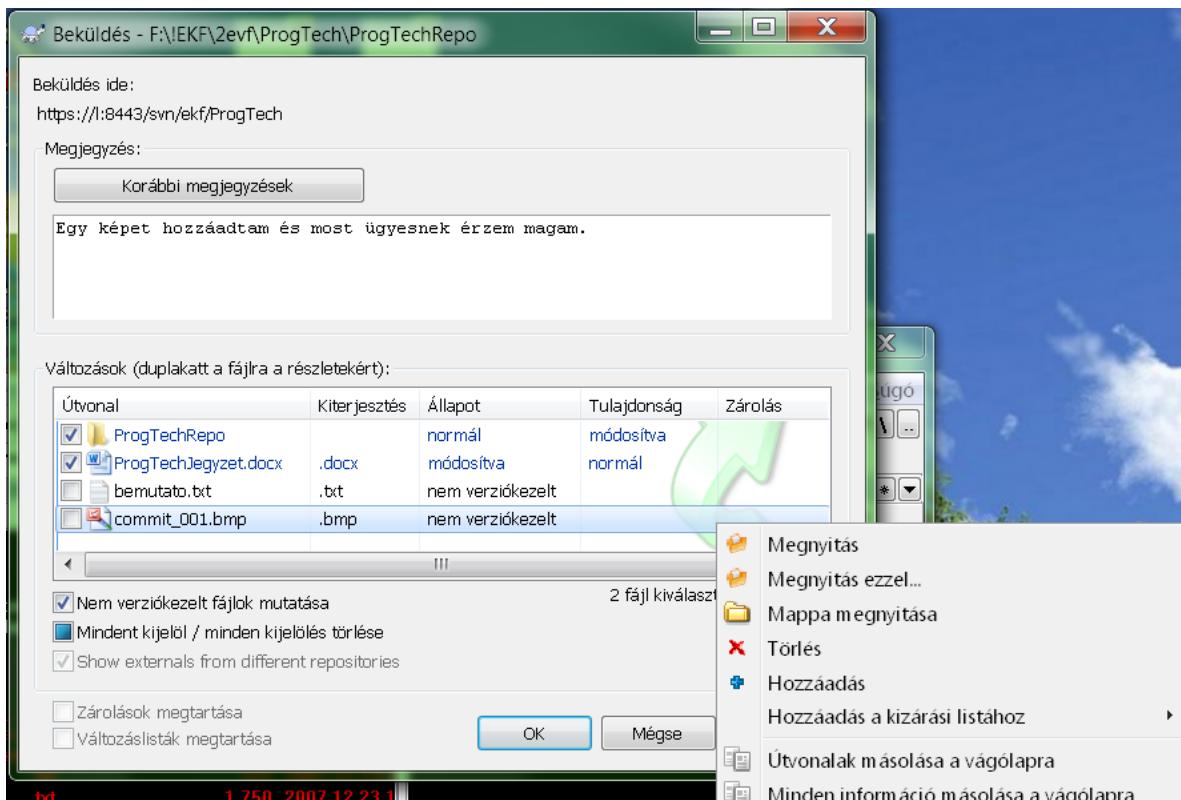


32. ábra: Beküldés TortoiseSVN segítségével

Miután módosításokat végeztünk a munkapéldányunkon, az új állományokat meg kell osztanunk másokkal is. Erre a *beküldés* funkció szolgál. A GUI rendszerbe beépült helyi menü segítségével egyszerűen elvégezhetjük a módosítások szerverre beküldését. Parancssorból a következőképp néz ki a beküldés:

```
svn commit \ProgTechRepo\bemutato3.txt
```

Természetesen nem kell egyesével beküldenünk az állományokat, megadhatunk könyvtárat is. Grafikus felületen a funkció kiválasztása után szemügyre vehetjük a legutóbbi beküldés óta módosult állományokat. Kijelölhetjük, hogy mi az, amit ténylegesen szeretnénk beküldeni. Ha új állományokat hoztunk létre, akkor azokat először hozzá kell adni a verziókövető rendszer felügyelte állományokhoz.

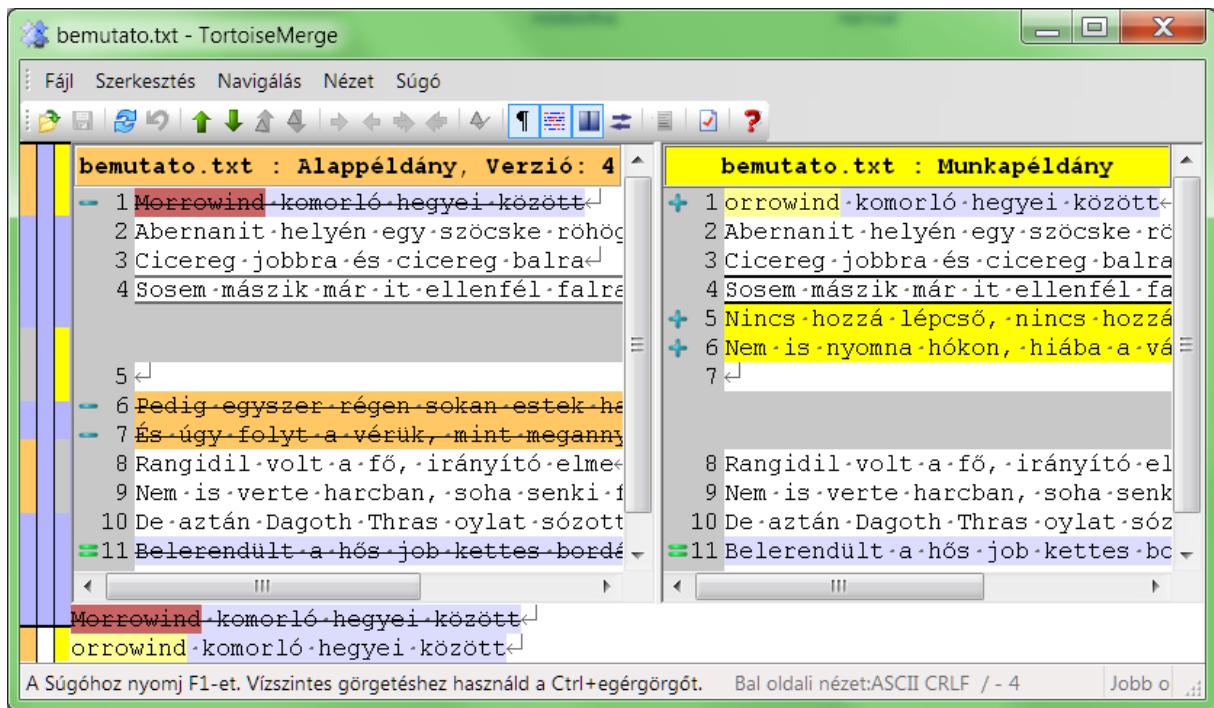


33. ábra: Hozzáadás TortoiseSVN segítségével

Ezt a *hozzáadás* művelettel lehetjük meg. Amíg ez meg nem történik, az SVN nem fog semmiféle információt tárolni az állományról. Grafikus felületen végtelenül egyszerű a művelet, de parancssorban sem kell sokáig töprengeni, íme:

```
svn add \ProgTech\ProgTechRepo\bemutato2.txt
```

Ezek után a következő beküldésnél a *bemutato2.txt* már szerepelni fog a beküldendők között. Közvetlen beküldés előtt ellenőrizhetjük, hogy milyen változtatásokat követtünk el. Ehhez egyszerűen duplán ráklikkelünk egy állományra, és az összehasonlító nézetben láthatjuk a munkamásolat és a tárházban tárolt másolat közötti eltéréseket. Különböző színekkel kiemeltek a hozzáadott és az eltávolított sorok és az is jelölést kapott, ha ütköző sorok vannak a két változatban. Ez utóbbi akkor fordulhat elő, ha többen is ugyanazt az állományt módosították és valaki már korábban beküldte az ő változatát. A mellékelt ábrán egyértelműen látszik, mi az, amit hozzáadtunk vagy eltávolítottunk az előző verzióhoz képest.

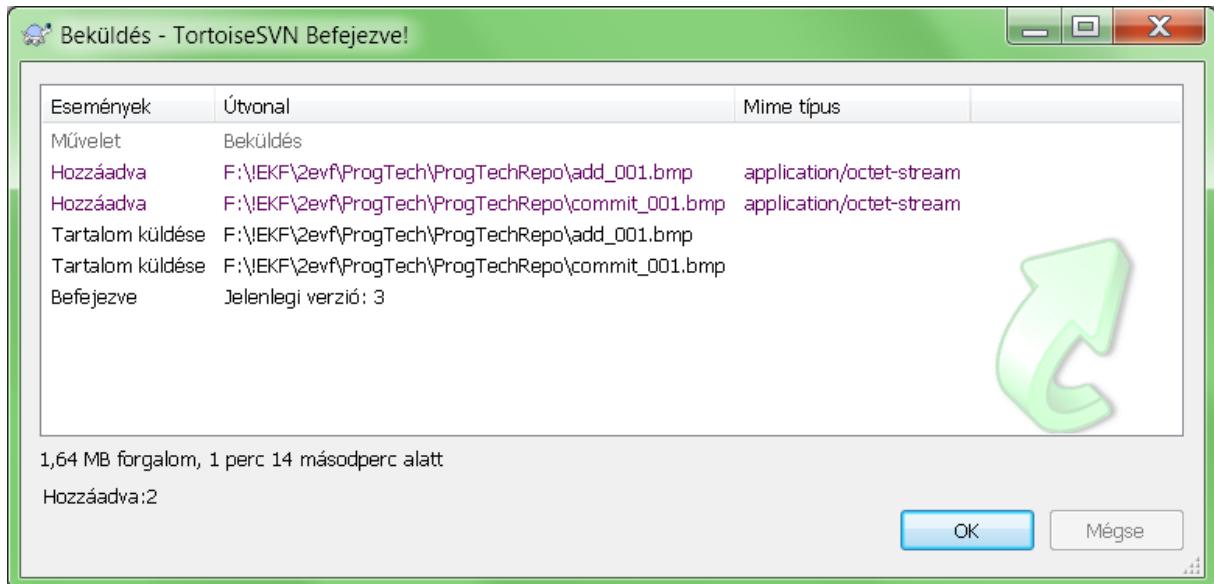


34. ábra: Összehasonlítás TortoiseSVN segítségével

Parancssorból az eltéréseket az következő parancccsal tekinthetjük meg:

```
svn diff \ProgTech\ProgTechRepo\
```

Sikeres hozzáadás és beküldés után grafikus felületen az alábbi összefoglalót kell látnunk.



35. ábra: Beküldés összefoglaló TortoiseSVN segítségével

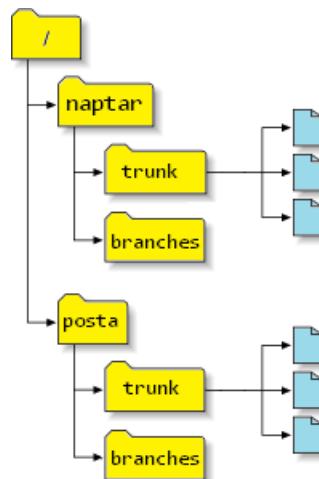
Ugyan a fenti módszerrel minden probléma nélkül be tudjuk küldeni a változtatásainkat, azért nem árt figyelembe venni azt, hogy mások is dolgoznak a rendszerben, és az is lehet, hogy nálunk korábban

beküldik változtatásait. Ha ez közvetlenül nem érinti a mi munkánkat, akkor semmi baj, ám megeshet, hogy ütközik a mi forrásainkkal, vagy tartalmaz olyan módosításokat, ami a mi kódunkban is változásokat követel meg. Ha ekkor gondolkozás nélkül beküldjük a munkánkat, akkor az nagy bosszússágot okozhat a kollégáinknak és lehet, hogy nem kapunk „pilótakekszet” a következő születésnapunkra. Ekkor megtehetnénk, hogy egyszerűen frissítjük a munkamásolatunkat, de ezzel lehet, hogy magunknak okozunk hajhullást. De mindez elkerülhetjük, ha nem közvetlenül a beküldés/frissítés parancsokat, hanem az ‘SVN Változások keresése’ (angolul: Check for modifications) menüpontot használjuk. Itt láthatjuk a saját változtatásainkat és ha a ’Tároló ellenőrzése’ gombra kattintunk, akkor láthatjuk a legutolsó frissítésünk óta a szerverre beküldött anyagokat is és azt is, hogy ki és mikor küldte be azokat. Ez alapján meg tudjuk ítélni, hogy érintettek-e bennünket érzékenyen a már beküldött módosítások. Ha gyanítjuk, hogy ilyen módosítások történtek, akkor az összehasonlító nézet alapján dönthetünk a további tennivalóról.

#### 7.1.2.5. Elágazások

Ez a fejezet a <http://svnbook.red-bean.com> alapján íródott.

A könnyebb megértés miatt tegyük fel, hogy az alábbi ábra szerinti könyvtárszerkezetben dolgozol te és a kollégád, Péter.



36. ábra: Példa alkönyvtár szerkezete

Mindkettőknek van egy munkamásolata a *naptar* projektről, és mindenketten a fő fejlesztési vonalon dolgoztok, azaz a *naptar/trunk* könyvtárban lévő elemeken. Tegyük fel, hogy azt a feladatot kapod, hogy gyökeresen változtasd meg az egész projekt szerkezetét. Biztos, hogy sokáig tart majd a fejlesztés és az is, hogy jóformán minden állományt meg kell változtatnod. A gond ott kezdődik, hogy Péter is feladatot kapott ugyanezen a projekten, méghozzá a menet közben felderített kisebb hibák javításait. Az ő munkájának alapvető feltétele, hogy a *naptar/trunk* könyvtárban mindig egy működőképes változat található. Ha te elkezded a fejlesztést és részenként beküldök a változtatásaidat, akkor biztos, hogy legalább Péter munkáját lehetetlenné fogod tenni, de nem lehetetlen, hogy másokét is.

Egyik megoldás lehet, hogy teljesen elszigetelj a magad és semmiféle információt nem adsz ki addig, amíg kész nem leszel a munkáddal. Azaz elkezded hegeszteni a munkamásolatodat, átszervezed, átírod, hozzáadsz és elveszel. De sem beküldést, sem frissítést nem végezel, amíg teljesen kész nem

vagy. Így számos problémába fogsz ütközni. Először is ez így nem biztonságos. Sokan azért is szeretik gyakran a tárházba menteni a munkájukat, ha véletlenül valami baleset történne a munkamásolattal. Másodszor pedig nagyon rugalmatlan. Ha több gépen is van munkamásolatod a *naptar/trunk* –ról, akkor kézzel kell a változtatásokat másolgatnod oda-vissza. Ugyanezért nehéz a munkádat megosztani bárki mással. Ha pedig mások nem látják a munkád, elképzelhető, hogy hetekig rossz irányba halad a fejlesztésed, mire valaki észreveszi a bajt. Az emiatt szokásos időszakos kódáttekintés (angolul: code review) sem működhet így. Végezetül, ha elkészültél a munkáddal, akkor nagyon nehéz lesz összefésülni (angolul: merge) a fejlesztés fő irányvonalával. Péter addigra már sok apró részletet megváltoztatott, ami tovább nehezíti az összefésülést, különösen akkor, ha a több hetes elszigetelődés után csuklóból kiadsz egy meggondolatlan svn update parancsot.

A helyes megoldás, hogy létrehozol egy saját ágat a tárházban. Ez lehetővé teszi, hogy beküldd a félig kész munkádat anélkül, hogy másokat akadályoznál, ugyanakkor a munkádról folyamatos képet kapnak a kollégáid.

#### 7.1.2.6. [Elágazás létrehozása](#)

Egy elágazást létrehozni felettesebb egyszerű a Subversion rendszerben, elég hozzá az

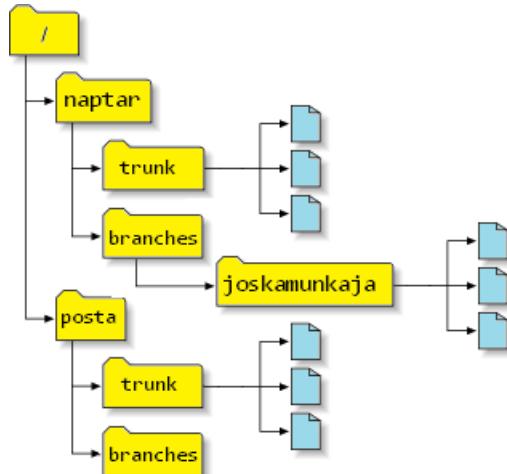
`svn copy`

parancs. Ez nem csak egyes állományokat, hanem teljes könyvtárakat is másol. Esetünkben a *naptar/trunk* könyvtárat szeretnénk másolni. Hová is? Ahová csak szeretnéd, de leginkább oda, ami megfelel a fejlesztésben meghatározott irányelveknek. Célszerű azonban a *naptar/branches/joskamunkaja* könyvtárba másolni, mert ha új fejlesztő száll be a ringbe, akkor kevesebbet kell neki magyarázni a könyvtárszerkezet felépítéséről és több idő marad kávezni. Nosza, ne habozzunk, adjuk ki az alábbi parancsot:

```
svn copy https://url.hu/svn/naptar/trunk  
https://url.com/svn/naptar/branches/joskamunkaja -m "A naptar/trunk elágazásának  
létrehozása."  
Committed revision 42.
```

Ennek több következménye is lesz. Először is létrejön a 42. verzió, amiben a joskamunkaja könyvtár létrejöttét tárjuk kedvenc munkatársaink elé. Az új könyvtár a *naptar/trunk* másolata lesz. A copy parancsnak nem véletlenül adtam URL paramétereket. Lehetséges lett volna a munkamásolaton végrehajtani a másolást, de az azzal az egész *naptar/trunk* tartalmát új helyre másoltuk volna, minimum duplájára növelte ezzel a szükséges tárhelyet, amihez hozzájön még az SVN által létrehozott .svn könyvtárak tartalma. Ezzel szemben, ha a szerveren adjuk ki a copy parancsot, akkor szinte azonnal kész a másolat, ami már sejteti azt, hogy valójában nem történt másolás és valami titok lappang az egész *naptar/trunk* könyvtár varázslatos teleportációja mögött. Nos, a másolás után valójában az új, *naptar/branches/joskamunkaja* könyvtár a régi *naptar/trunk* könyvtár tartalmát mutatja. Azonban ha bármin változtatunk, akkor a változások már az új, *naptar/branches/joskamunkaja* könyvtárban foglalnak helyet. Megjegyzendő, hogy minden egyes beküldés a tárházba ezzel, az úgynevezett „cheap copy” módszerrel történik. Természetesen a belső működése a másolásoknak és adatmegosztásoknak nem látható a külső szemlélő számára. Úgy tűnik, mintha ténylegesen létrejött volna a teljes másolata

a naptar/trunk könyvtárnak. Ugyanezt grafikus felületen is megtehetjük a *Mellékág/kiadás* menüponttal.



37. ábra: Példa alkönyvtár szerkezete másolás után

Miután létrejött az elágazás, készíts róla egy munkamásolatot akár az svn checkout parancssal, akár a grafikus felület *Lekérés* menüpontjával. Ebben eddig semmi különleges nincs, mert megkapod a tárház egyik könyvtárának másolatát. Azonban amikor beküldök a változtatásaidat, azokat Péter már nem fogja látni. Tegyük fel, hogy egy hét során a következő változtatások következnek be:

- megváltoztatod a naptar/branches/joskamunkaja/honap.aspx állományt
- megváltoztatod a naptar/branches/joskamunkaja/nevnap.aspx állományt
- Péter megváltoztatja a naptar/trunk/nevnap.aspx állományt

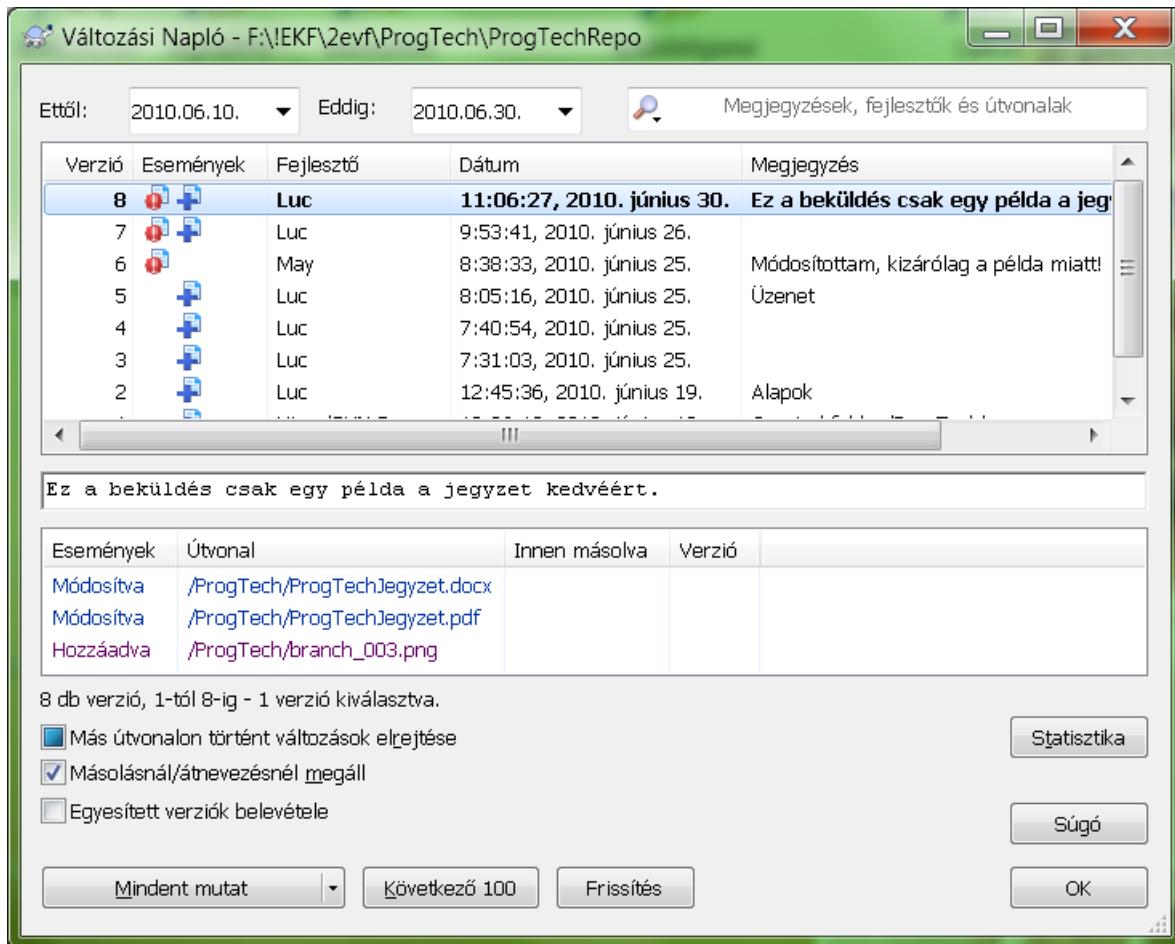
Nyilvánvaló, hogy a nevnap.aspx eredetileg ugyanaz az állomány, így aztán akár a törsben, akár az elágazásban nézzük a korábbi változtatásait, minden helyen láthatjuk a teljes történetét. Azonban az elágazás létrejötte után már csak ahhoz az ághoz tartozó változásokat látjuk, amelyik ághoz tartozó munkamásolatot használjuk éppen.

#### 7.1.2.7. Összefésülés (Merging)

Mivel a hosszú ideig tartó párhuzamos fejlesztés során nagy eltérések alakulhatnak ki, egy idő után majdnem lehetetlen lesz úgy összefésülni a két eltérő vonalát a fejlesztésnek, hogy ne lenne rengeteg konfliktus. Azonban a párhuzamos fejlesztések alatt lehetőséged nyílik megosztani Péterrel a változtatásaidnak egy bizonyos részét. Rajtad áll, hogy mit tartasz megosztásra érdemesnek. Végül a különálló fejlesztés végeztével már jóval kevesebb konfliktus adódhat.

Mielőtt tovább haladnánk, állapodjunk meg abban, hogy esetünkben mit jelent a változáslista (changeset). A Subversion esetében az N verziósáma egy fát jelöl ki a tárházban. Ugyanez az N tulajdonképpen egy változáslista neve is, mivel ha összehasonlítod az N és az N-1 verziókat, megkapod a két verzió közötti eltéréseket. Éppen ezért az N verziót könnyebb felfogni változáslistaként, mint egy faként. Hibák követő rendszerekben is hivatkozhatunk a változáslistákra, például: „Ez a hiba a 452 verzióban került javításra.” Ekkor bárki olvashat a változásokról az svn log -r 452 parancs segítségével, vagy akár a pontos beküldéseket is megnézheti a svn diff -c 452 parancssal. Mindezt lényegesen egyszerűbben megtehetjük a grafikus felület *Változási napló* menüpontjával. Bármelyik utat is

választjuk, végül birtokunkba kerül a kívánt változási lista azonosítója, amit a későbbiekben az svn merge parancsnak átadhatunk paraméterként.



38. ábra: Változási lista TortoiseSVN segítségével

Példánkat folytatva tételezzük fel, hogy egy hete már fejleszted a saját elágazásod. Az új funkció még nincs kész, de tudod, hogy Péter fontos változásokat eszközölt a naptar/trunk könyvtárban. Ezeket a változásokat a saját érdekedben be kell vezetned a naptar/branches/joskamunkaja-ba is. Valójában a bevált gyakorlat az, hogy rendszeresen szinkronban kell tartani az elágazást a törzssel, minek következtében elkerülhetők azok az előre nem látott konfliktusok az elágazásod törzsbe visszafésülésekor, amikre a „kellemetlen meglepetés” korántsem a precízebb, de talán legnyomdifestéktűrőbb kifejezés. Mielőtt a törzs változtatásait az elágazásodba fésülnéd, be kell küldened minden változtatást, amiket az elágazásodban elkövettél. Az svn status parancssal, vagy a grafikus felület *SVN változások követése* menüpontjával meggyőződhetsz arról, hogy „tiszta-e” a munkamásolatod, azaz beküldtél-e minden változtatást. Ha ez rendben van, akkor add ki a következő parancsot:

svn merge <https://url.hu/svn/naptar/trunk>

## 7.2. Hibakövető rendszerek

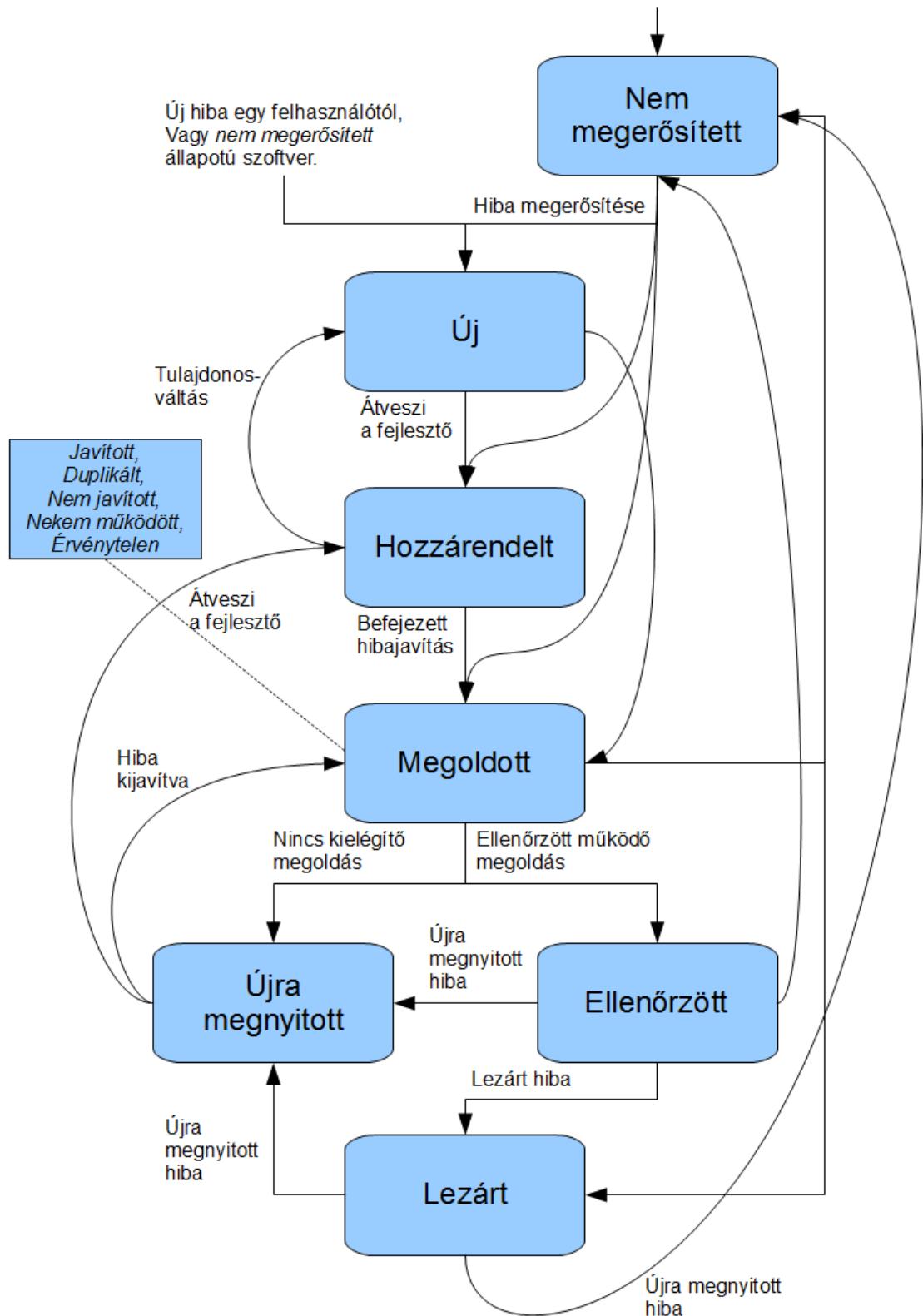
Egy szoftver készítése során és átadása után is merülhetek fel problémák a működéssel kapcsolatban. A szoftver készítése során és gyakran az átadás után is tesztelők keresnek hibákat, illetve az átadás után a felhasználók futhatnak bele egy-egy hibába. Ezeket a hibákat javítani kell, amihez a programozóknak értesülniük kell a hibáról. A hiba felfedezője és a fejlesztők között a hibakövető (angolul: bug tracking) rendszerek teremtik meg a kapcsolatot. A hibakövető rendszereket néha hívják hibabejelentő rendszereknek is. Hibakövető rendszer például:

- a JIRA,
- a BugTracker.NET,
- a Mantis,
- és a Bugzilla is.

Ebben a jegyzetben a Bugzilla és a Mantis rendszert mutatjuk be.

### 7.2.1. [Bugzilla](#)

A hibakövető rendszerek legfontosabb tulajdonsága, hogy milyen életútja lehet a hibának a rendszeren belül. Ezt a hibakövető rendszer állapotgépe írja le. Az alábbi ábrán a Bugzilla állapotgépét láthatjuk:



39. ábra: A Bugzilla állapotgépe

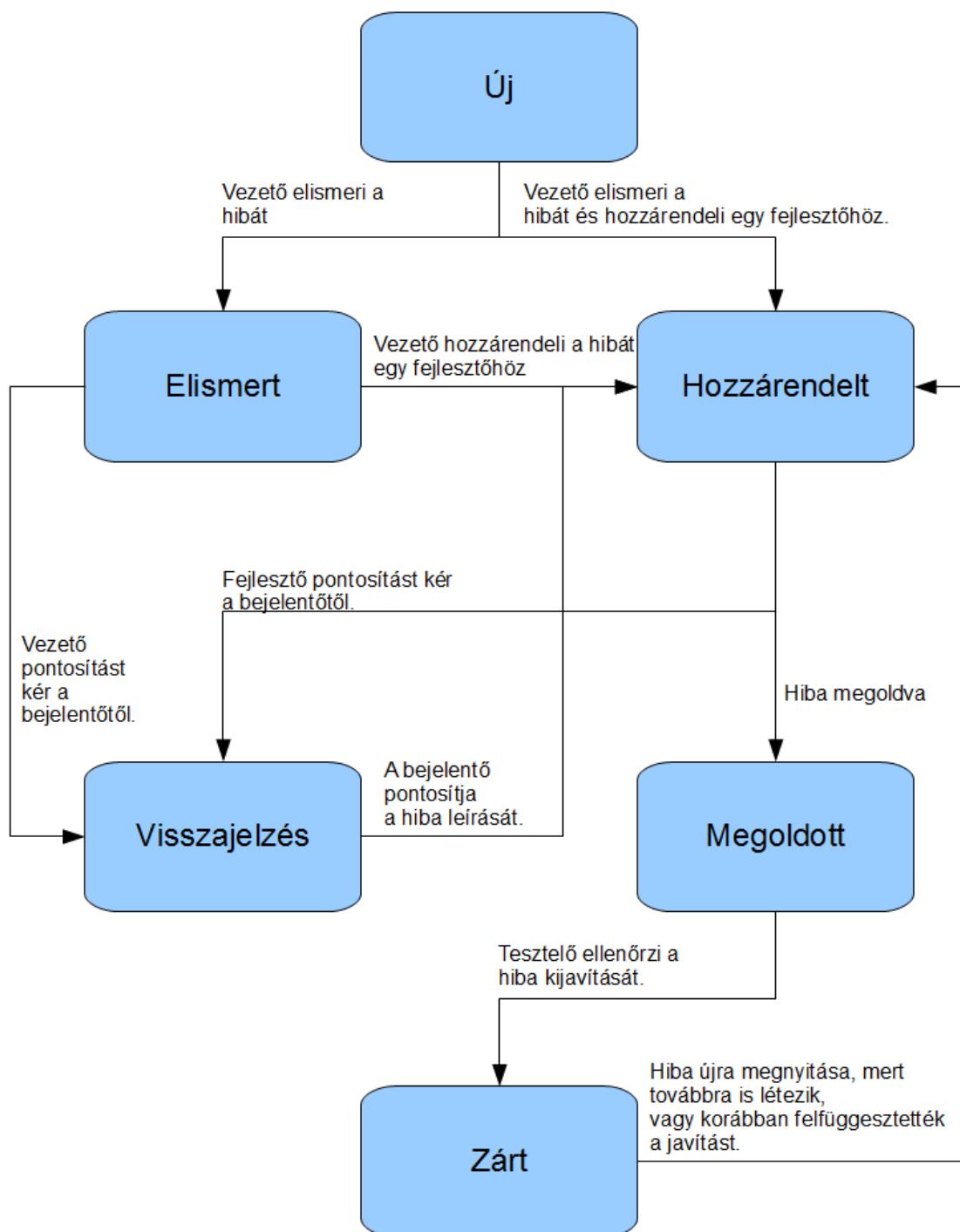
A hiba legegyeszerűbb életútja a következő:

- A hibát jelenti a tesztelő vagy a felhasználó. Fontos, hogy minél részletesebb legyen a hiba leírása, hogy reprodukálható legyen. Ekkor a hiba Új állapotú lesz.
- Az új hibákról értesítést kap a vezető fejlesztő, aki a hibát hozzárendeli az egyik fejlesztőhöz, általában ahhoz, aki a hibás funkciót fejlesztette. Ekkor a hiba Hozzárendelt állapotba kerül.
- A fejlesztő a hozzárendelt hibát megpróbálja reprodukálni. Ha ez sikerül és megtalálja a hiba okát is, akkor javítja a hibát. A javítást feltölti a verziókövető rendszerbe, majd jelzi, hogy megoldotta a hibát. Ilyenkor érdemes egy regressziós tesztet csinálni, hogy meggyőződjünk, hogy a javítás nem okoz-e más hibákat. Ekkor a hiba Megoldott állapotú lesz.
- A megoldott hiba visszakerül az azt bejelentő tesztelőhöz, vagy esetleg egy másikhoz. A tesztelő ellenőrzi, hogy tényleg megoldódott-e a hiba. Ha igen, akkor véget ér a hiba életútja, az állapota lezárt lesz.

Az optimális lefutástól sok helyen eltérhetünk. Például kiderül, hogy a hiba nem reprodukálható, vagy a megoldott hibáról kiderülhet, hogy még mindig fennáll. Ezeket a lehetőségeket minden lefedi a fenti állapotgép.

#### 7.2.2. Mantis

Ebben a fejezetben a Mantis (magyarul imádkozó sáska) ingyenes hibakövetést támogató rendszert mutatjuk be. Az alábbi ábra a Mantis állapotgépét szemlélteti. Itt láthatjuk, hogy az egyes állapotokból hogyan jut át a másikba a hiba.



40. ábra: A Mantis állapotgépe

Tekintsük át, hogyan halad a folyamat egy hiba bejelentésétől annak lezárásáig. Először is a rendszerhez hozzáféréssel kell rendelkeznie a bejelentőnek. Viszonylag egyszerű, ha belső tesztelésről van szó, mert ott többnyire adott a jogosultság a hibabejelentésre. Származhat a bejelentés a megrendelőtől is, aki kapott tesztelésre egy korai verziót, vagy ami rosszabb, már rendelkezik egy kész, kiadott verzióval. Elképzelhető olyan bejelentés is, amikor mi vesszük fel a rendszerbe a hibát, az ügyfél telefonos elmondása vagy levele alapján. A hiba bejelentője a rendszerbe bejelentkezve láthatja minimum a saját maga által bejelentett hibákat, állapotukat és a hozzájuk fűzött megjegyzéseket, valamint ő maga is további információkat fűzhet a bejelentéshez, sőt, erre gyakran meg is kérík a hibajavítók a bejelentőt.

A Mantis rendszerben vázlatosan az alábbi módon történik a hibakezelés:

- A hibát bejelentik. Ezt a hibabejelentést a hibás szoftverhez rendeljük (hiszen egy hibakövető rendszer több szoftver hibáit, illetve egy szoftver több verziójának hibáit is tartalmazhatja), elláthatjuk kategóriával, reprodukálhatósággal és súlyossággal. Kezdetben állapota Új lesz, ami a későbbiekben folyamatosan változik a hiba javítása során.
- Kategóriákat magunk adhatunk meg a tesztelt rendszer igényei szerint. A későbbiekben a hibákat bejelentő felhasználók ezekbe a kategóriákba sorolhatják a hibákat.
- A bejelentésnek lehet reprodukálhatósága:
  - Mindig: Leírásában megadott lépésekkel megismételve, a hiba mindenkor jelentkezik.
  - Néha: Nem minden esetben jelentkezik. Ez többszálú programokra jellemző.
  - Véletlenszerű: Véletlenszerűen jelentkezik. Ebben az esetben nagyon fontos, hogy megadjuk a hiba észleléssének pontos időpontját, hogy a fejlesztők a rendszernapló állományban visszakereshessék a hiba előtti állapotot, amiből rekonstruálható a hibát kiváltó események sora.
  - Nem ismételhető: Nem tudtuk megismételni. Ebben az esetben is nagyon fontos, hogy megadjuk a hiba észleléssének pontos időpontját.
- A bejelentésnek lehet súlyossága:
  - Funkció: Funkció hibás működése.
  - Nyilvánvaló: Megjelenítés, rossz szövegigazítás, egyéb szoftverergonomiai hiba.
  - Szöveg: Elírás, helyesírási hiba.
  - Nem súlyos: Kisebb, általában funkcionális hiba, aminek létezik egyszerű megkerülő megoldása.
  - Súlyos: Valamely funkció teljesen hibásan működik.
  - Összeomlás: A hiba a rendszer összeomlását eredményezi.
  - Akadály: Vagy a fejlesztés, vagy a tesztelés nem folytatható, amíg ez a hiba fennáll.
- A bejelentés állapota lehet:
  - Új: Olyan bejelentés, amihez még senki sem nyúlt.
  - Elismert: A vezető fejlesztő elismeri a hiba léttét, de még nem osztotta ki, mert például nincs elég információ a hibáról.
  - Visszajelzés: A hiba leírását pontosítani kell, a fejlesztők további információt kérnek a hibáról.
  - Hozzárendelt: A hibához fejlesztő lett rendelve.
  - Megoldott: Megoldás született a bejelentésre.

- Lezárt: A bejelentés lezárásra került.

Adja meg a hiba adatait		[ Részletes bejelentő ]
<b>Kategória</b>	admin	admin gyűlés integráció site
<b>Reprodukálhatóság</b>	mindig	mindig néha véletlenszerű nem próbáltam nem ismételhető nincs adat
<b>Súlyosság</b>	nem súlyos	funkció nyilvánvaló szöveg trükk nem súlyos súlyos összeomlás akadály
<b>*Összegzés</b>	A hiba rövid leírása	
<b>*Leírás</b>	Hiba Leírása. <b>ID</b> Platform Hibát előidéző lépések	
<b>További információ</b>	További megjegyzések	
<b>File feltöltése</b> (Maximális méret: 1,000k)	<input type="file"/> Tállízás...	
<b>Betekintés jellege</b>	<input checked="" type="radio"/> nyilvános <input type="radio"/> privát	
<b>Új bejelentő</b>	<input type="checkbox"/> (további hiba bejelentéséhez jelölje be)	
<b>* kitöltendő</b>	<input type="button" value="Bejelentő elküldése"/>	

41. ábra: Egy hibabejelentő ablak a Mantis programból

A projekt felelőse figyeli az érkező bejelentéseket és a megfelelő fejlesztőhöz rendeli a hibát. Ez úgy is történhet, hogy kiadja egy-két fejlesztőnek, hogy nézzék át a bejelentéseket és kezdjék meg a hibák javítását. Egy hibát általában hozzá lehet rendelni más hozzá és magunkhoz is.

Hibák listája (1 - 50 / 90) [ <a href="#">Hibák nyomtatása</a> ] [ <a href="#">CSV exportálása</a> ]							
	P. azonosító	#	Kategória	Súlyosság	Állapot	Módosítva	Összegzés
	0004620	6	site	összeomlás	visszajelzve (jpetto)	07-30-10	ismeretlen hiba a gyűjtés lezárásnál
	0004513	3	site	trükk	megoldva (mbeothy)	07-30-10	A .csv sablon letöltésére klikkelésnél a sablon a bongészben nyílik meg.
	0004432	3	site	funkció	visszajelzve (jpetto)	07-30-10	Törölt felhasználói név foglalt marad.
	0004504	3	gyűlés	funkció	megoldva (mbeothy)	07-30-10	0004464: Szavazási séma duplikáció
	0004505	1	site	súlyos	visszajelzve (mbeothy)	07-30-10	Lassú internettel való videóbeszélgetés szinte lehetetlen
	0004465	2	integráció	összeomlás	visszajelzve (plukacs)	07-30-10	7024-es hiba szavazás elküldésekor
	0004466	1	gyűlés	funkció	visszajelzve (plukacs)	07-30-10	Szavazási séma duplikáció
	0004442	1	gyűlés	nem súlyos	visszajelzve (plukacs)	07-30-10	zavaróan fennmaradó címke
	0004467	3	gyűlés	súlyos	megoldva (mbeothy)	07-26-10	Újraszavazásnál hibaüzenet
	0004502	1	gyűlés	nem súlyos	megoldva (mbeothy)	07-26-10	Gyűlés kezdő és vég időpontja megegyezhet.
	0004619		site	összeomlás	kioszta (plukacs)	07-26-10	szét csúszik az oldal
	0004631		site	nem súlyos	kioszta (plukacs)	07-26-10	Logo eltérés
	0004462	1	gyűlés	funkció	megoldva (mbeothy)	07-08-10	Levezető elnöki szerepkör átruházásakor hibaüzenet
	0004520		gyűlés	nem súlyos	megoldva (mbeothy)	07-07-10	Gyűlés módosításánál: Új gyűlés felirat
	0004510	3	gyűlés	trükk	megoldva (mbeothy)	07-07-10	Nagyító gomb a dokumentumoknál, nem működik.
	0004508	1	site	súlyos	kioszta (mbeothy)	07-05-10	Profilon belül, névnek HTML kódok megadása.
	0004470	3	gyűlés	súlyos	kioszta (mbeothy)	07-05-10	Gyűlés indításánál elő lehet csalni egy info újra küldést.
	0004433	1	site	nem súlyos	kioszta (felhasználó)	07-05-10	Új jelszó lehet a generált.
	0004514	2	site	súlyos	megoldva (mbeothy)	05-19-10	Vendég felülröhatja a dokumentumtár fájlait
	0004515		site	súlyos	megoldva (mbeothy)	05-19-10	a vendég tudja törölni az el nem indított gyűléseken beállított szavazásokat
	0004501	3	gyűlés	nem súlyos	megoldva (mbeothy)	05-18-10	Ha egy tag súlya nullánál nagyobb, attól még nincs szavazati joga.
	0004503	3	gyűlés	összeomlás	megoldva (mbeothy)	05-18-10	Nem tudok, sablont törölni a korábbi sablonok közül
	0004518	1	admin	nem súlyos	megoldva (mbeothy)	05-17-10	Hozzájárás elkeszítése (sokszor kattintással) hatására ismeretlen hiba szöveggel hibaüzenet jelenik meg.
	0004506	1	site	szöveg	visszajelzve (felhasználó)	05-17-10	Nem lehet új felhasználót létrehozni ha a felhasználónév tartalmaz bizonyos karaktereket, szerverhibára hivatkozva.
	0004500		site	nem súlyos	kioszta (ttompa)	05-17-10	Videó konferenciába való belépéskor hibaüzenet: An unhandled win32 exception occurred in chrome.exe [3324]&#B221;

42. ábra: Bejelentett hibák ablak a Mantis programból

A hiba bejelentője ezután megnézheti, hogy hol tart a javítás. Láthatja, hogy a hiba az Új állapotból milyen állapotba került át. Ha már kiosztva állapotú, akkor jogaitól függően láthatja azt is, hogy kihez került a hiba, és milyen megjegyzéseket fűztek eddig hozzá.

Hiba egyszerű adatai [ <a href="#">Ugrás a megjegyzésekhez</a> ]		[ << ] [ >> ]		[ <a href="#">Részletes nézet</a> ] [ <a href="#">Hiba Torténet</a> ] [ <a href="#">Nyomtatás</a> ]		
azonosító	Kategória	Súlyosság	Reprodukálhatóság	Bejelentés dátuma	Utolsó módosítás	
0004467	[együles] gyűlés	súlyos	néha	04-23-10 18:20	07-26-10 15:38	
Bejelentő		Betekintés jellege	nyilvános			
Felelős						
Prioritás	átlagos	Megoldás	javitva			
Állapot	megoldva					
Összegzés	0004467: Újraszavazásnál hibaüzenet					
Leírás	HibaID: 130012 Platform: Windows Vista, IE7 Menü: Gyűlés -> Fázisok -> 1. fázis -> szavazás -> újraszavazás Hibát megelőző lépések leírása: Hiba leírása:					
További információ	2010.04.22. 10:12-kor tapasztaltam, de már sokszor beleütköztem.					
Csatolt fileok						
	<a href="#">Hiba nyomkövetése</a> <a href="#">Hiba újra megnyitása</a>					

43. ábra: Bejelentett hiba adatai ablak a Mantis programból

A javítással megbízott programozó és a hiba bejelentője gyakran nem ugyanazt a szaknyelvet beszélik. Ebből aztán rengeteg félreérte a hibát. A programozó nem érti a bejelentést, a bejelentő nem érti, hogy mit nem ért a programozó. Így elég hosszadalmas párbeszéd alakulhat ki a hibakezelő rendszer segítségével, ám végül tisztázódik a helyzet, és vagy lezárják a hibát azzal, hogy ez nem hiba, csak az

ügyfél nem olvasta el a kézikönyvet (nagyon ritkán olvassa el), vagy pedig elkezdi a programozó a javítást.

- Ha kész a javítás, legalábbis amikor a programozó ezt gondolja, akkor a „tesztelésre vár” státussal látja el a hibát.
- Ezután többen is tesztelhetik a javítást. Erre szükség is lehet, mivel a több szem többet lát elv itt fokozottan érvényesül, sőt mi több, a hiba kijavításával esetleg más, rejtett hibákat okozhattunk, ami ugyan eléggé amatőr eset (azt mutatja, hogy nem csináltunk regressziós tesztet), de nem kizárt.
- Ha sikeres a tesztelés, akkor a hibát kijavítottnak jelölhetjük, és ha a bejelentő is megelégedett a megoldással, akkor azt a hibabejelentés megoldás mezője segítségével közli.

Megjegyzések			
(0008147) <b>ttompa</b> 05-05-10 10:56	A form submit többszöri újraküldésének tiltásával megoldható a probléma.		
(0008158) <b>plukacs</b> 05-05-10 13:57	dupla submit megoldva az meg hogy minden bázist állít be a levezető a létszámmellenőrzésnél, az az ö dolga		
Hiba Történet			
Módosítás dátuma	Felhasználónév	Mező	Változás
05-03-10 11:13	pmaiko	Új Hiba	
05-03-10 11:13	pmaiko	File Feltölve: : szavazat_szamlalas.rar	
05-05-10 10:55	ttompa	Allapot	új => kiosztva
05-05-10 10:55	ttompa	Felelős	=> plukacs
05-05-10 10:56	ttompa	Hiba Megjegyzés hozzáadva: 0008147	
05-05-10 13:57	plukacs	Allapot	kiosztva => megoldva
05-05-10 13:57	plukacs	Megoldás	nyitva => javítva
05-05-10 13:57	plukacs	Hiba Megjegyzés hozzáadva: 0008158	

44. ábra: Hibatörténet ablak a Mantis programból

A fenti módszerrel természetesen nemcsak bejelentett hibákat lehet nyomon követni, hanem akár új igények megvalósítását, vagy akár egész termékek gyártását is követhetjük vele, ezért hívják ezeket a rendszereket munkakövető (issue tracking) rendszereknek is.

Utóbb minden menedzser álmát is ki lehet nyerni a rendszerből, azaz mindenféle grafikonokat és kimutatásokat. Ezen kinyerhető adatok között találhatunk fontosakat is, mint például, hogy mennyit dolgoztunk egy munkán. Ez segít a továbbiakban megbecsülni, hogy bizonyos típusú munkákat mennyi idő alatt lehet elvégezni.

### 7.3. Modellező eszközök

Ebben a fejezetben a rendszertervezhez szükséges modellek előállítását lehetővé tevő programokat mutatjuk be. Ezek az eszközök elsősorban UML ábrák létrehozására alkalmasak.

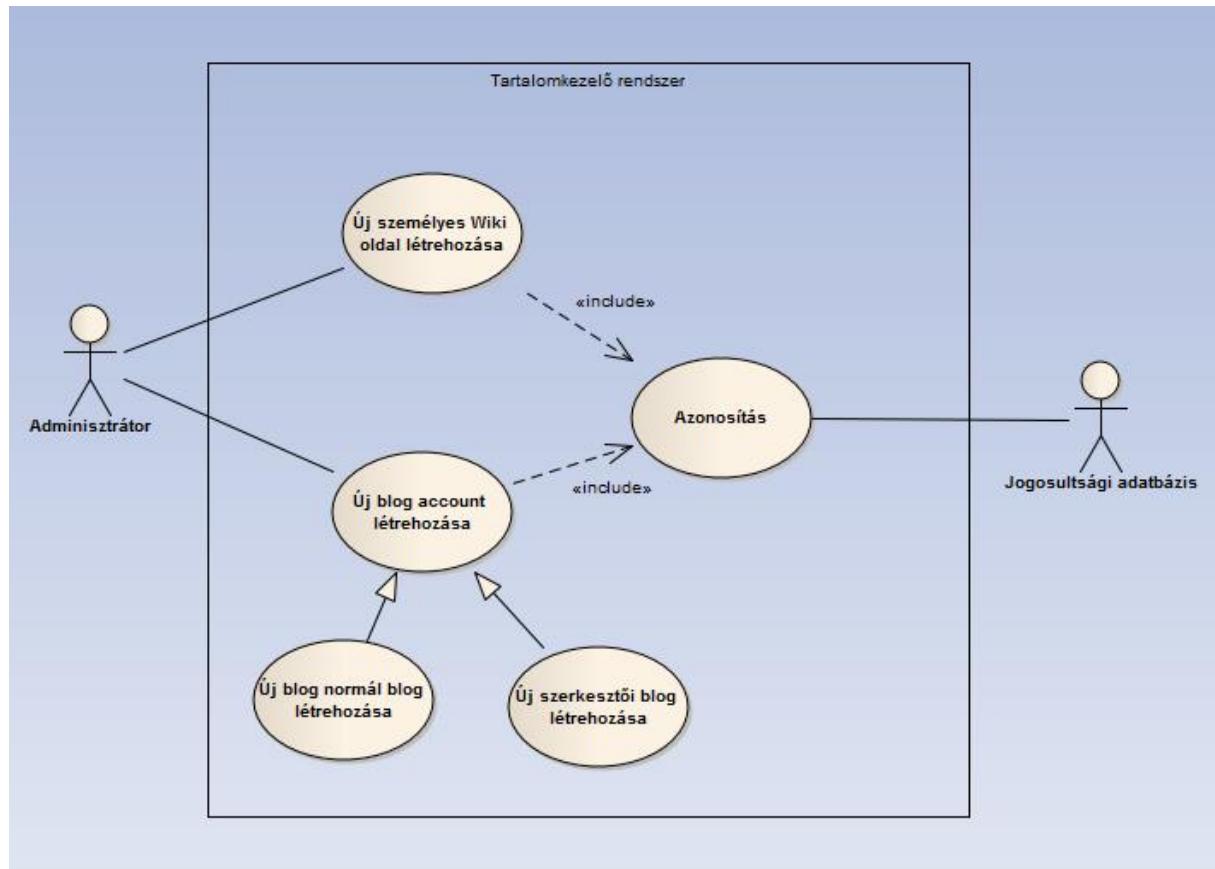
#### 7.3.1. Enterprise Architect

Az Enterprise Architect segítségével UML ábrákat készíthetünk. Sajnos ez a program nem ingyenes, szemben például a StarUML programmal. Ugyanakkor felépítése logikus, nagyban megfelel az előző fejezetekben leírt tagolásnak: üzleti folyamatok feltárása, igények elemzése, funkciók specifikációja, rendszerterv elkészítése, tesztelés, üzembe helyezés, karbantartás.

A következő alfejezetekben bemutatjuk, hogyan lehet Enterprise Architect segítségével létrehozni a főbb UML diagramokat.

### 7.3.1.1. Használati eset

Alább látható egy egyszerű használati eset (angolul: use case). Egy tartalomkezelő rendszer blog - illetve Wiki oldal létrehozását mutatja be. Az adminisztrátor nevű aktor új Wiki oldalt és blogot kíván létrehozni. Mielőtt bármit is tehetne, a jogosultsági adminisztrátor aktor ellenőrzi, hogy van-e jog a ezeket tenni. Miután a jogosultsági adminisztrátor megyőződött, hogy jogosan kíván létrehozni tartalmat, lehetősége nyílik az adminisztrátor nevű aktor-nak, hogy a kívánt tartalmat létrehozza. Személyes Wiki oldalt, vagy blogot, aminek két fajtája van: normál – és szerkesztői blog.

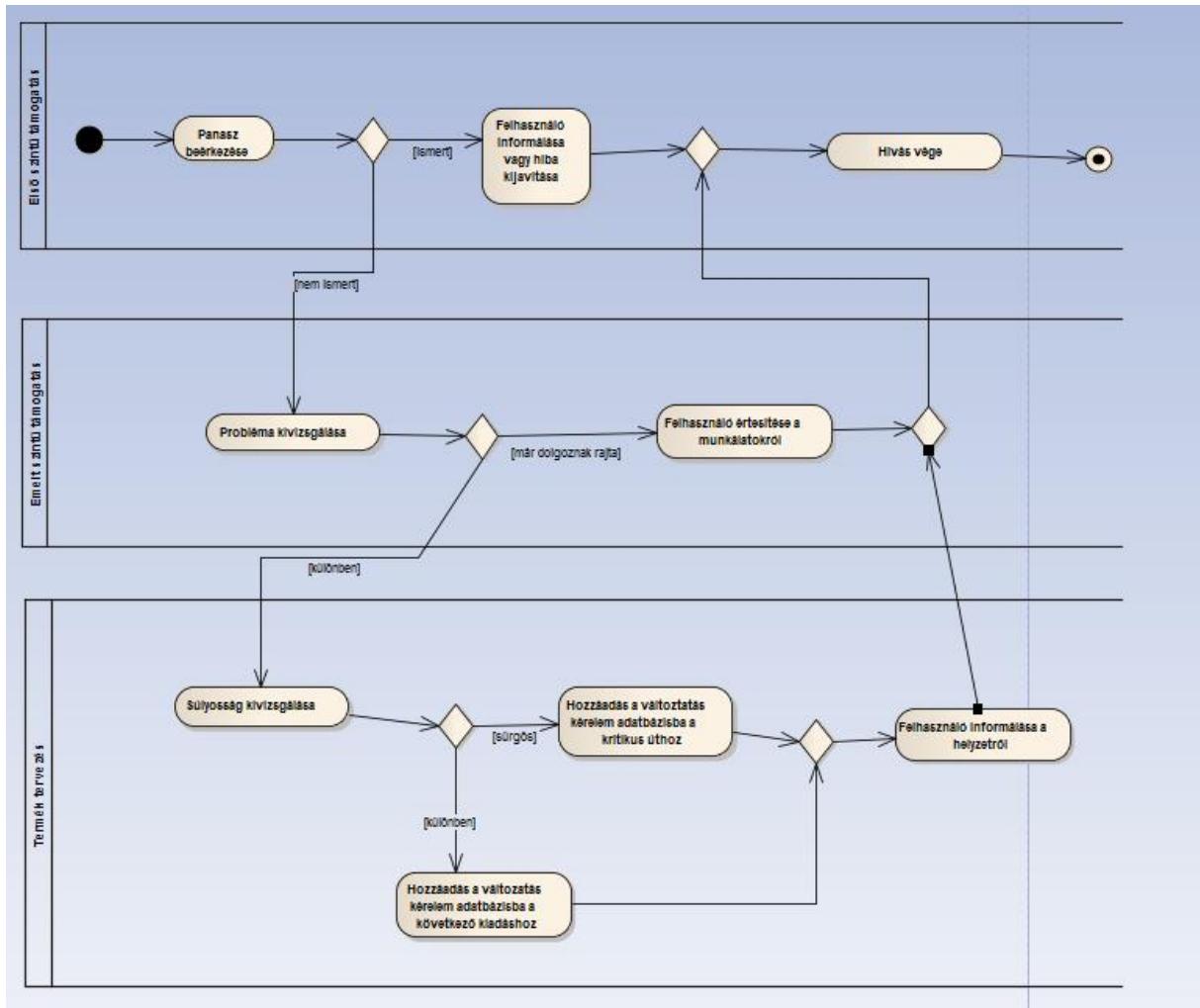


45. ábra: Egy példa UML használatieset-diagram

### 7.3.1.2. Aktiváció Diagram

Alatt látható egy aktivitási diagram. A Business Process Model-en belül kell létrehozni. Míg a használati esetek azt határozzák meg, hogy mit tudjon a rendszer, az aktivitási diagram a hogyan kérdést válaszol meg. A benne található cselekvések bármik lehetnek (számítás, valamelyen viselkedés). Ahogy a példánk is mutatja, aktivitási diagramot használhatunk, hogy modellezük a blogbejegyzés létrehozását. Az aktivitási diagramot üzleti folyamatok modellezésére használjuk.

A tevékenységek sokszor többféle résztvevőt is érinthetnek. Például különböző szerepkörű felhasználókat, vagy jogosultságok alapján kell szétválasztani az üzleti folyamatot. Ekkor használjuk a partíciókat (részeket). A megfelelő partíciók mutatják, hogy a különböző csoportok mely résztevékenységről felelősek. Az alábbi aktivitási diagram egy ilyen partícióra osztott üzleti folyamatot ábrázol.



46. ábra: Egy példa UML aktivitási diagram partíciókkal

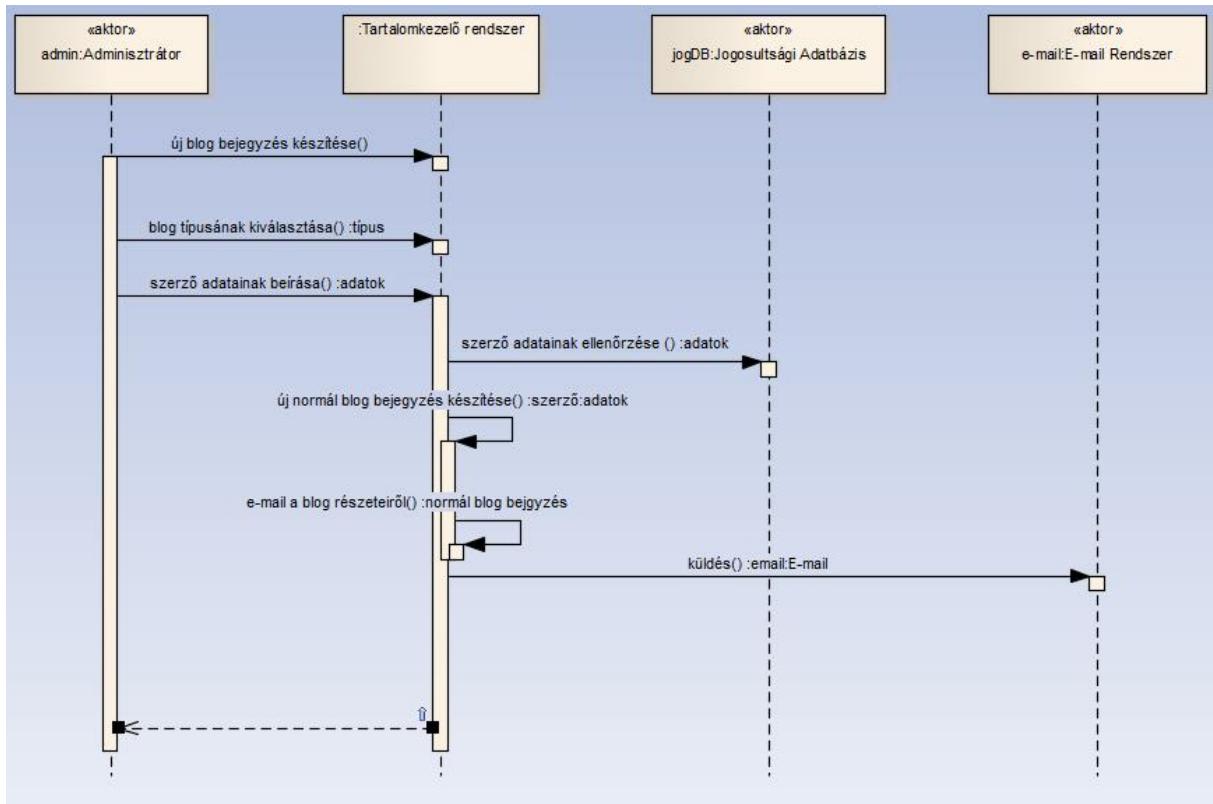
#### 7.3.1.3. Szekvenciadiagram

A rendszer részei között végbemenő interakciókat szemlélteti. Bemutatja a résztvevőket, az azok között történő üzenetküldéseket, előtérbe helyezve az időbeli sorrendiséget. Tekintsük a fentebb leírt használati eset szekvenciadiagramját.

A különböző nyilakkal az időrendiséget és az üzenetek fajtáit adhatjuk meg. A nyilak a következők:

- Szinkron: →
- Visszatérés: <-----
- Egyszerű: →
- Aszinkron: →\

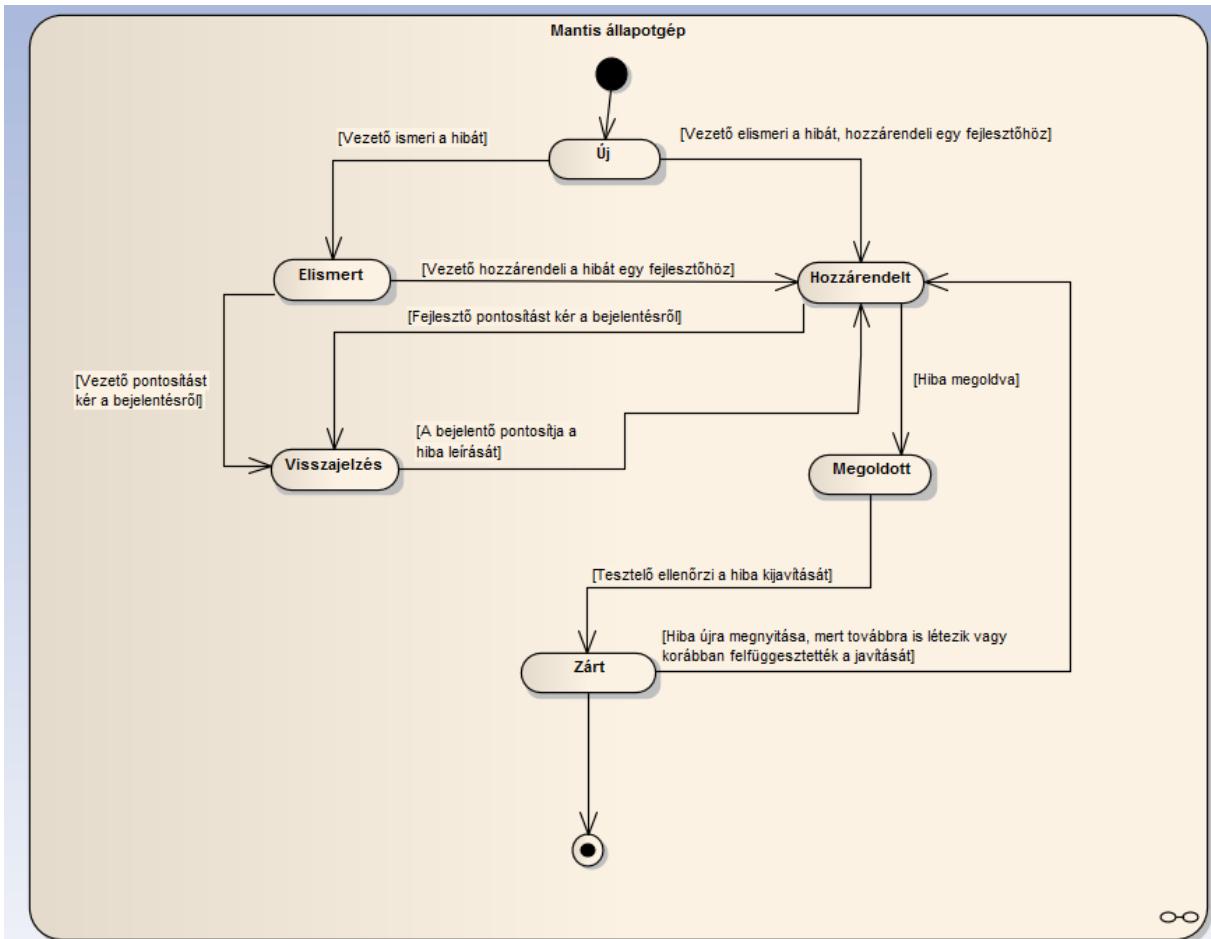
Enterprise Architect-ben a Logical View-ban kell létrehozni a szekvenciadiagramot.



47. ábra: Egy pelda UML szekvenciadiagram

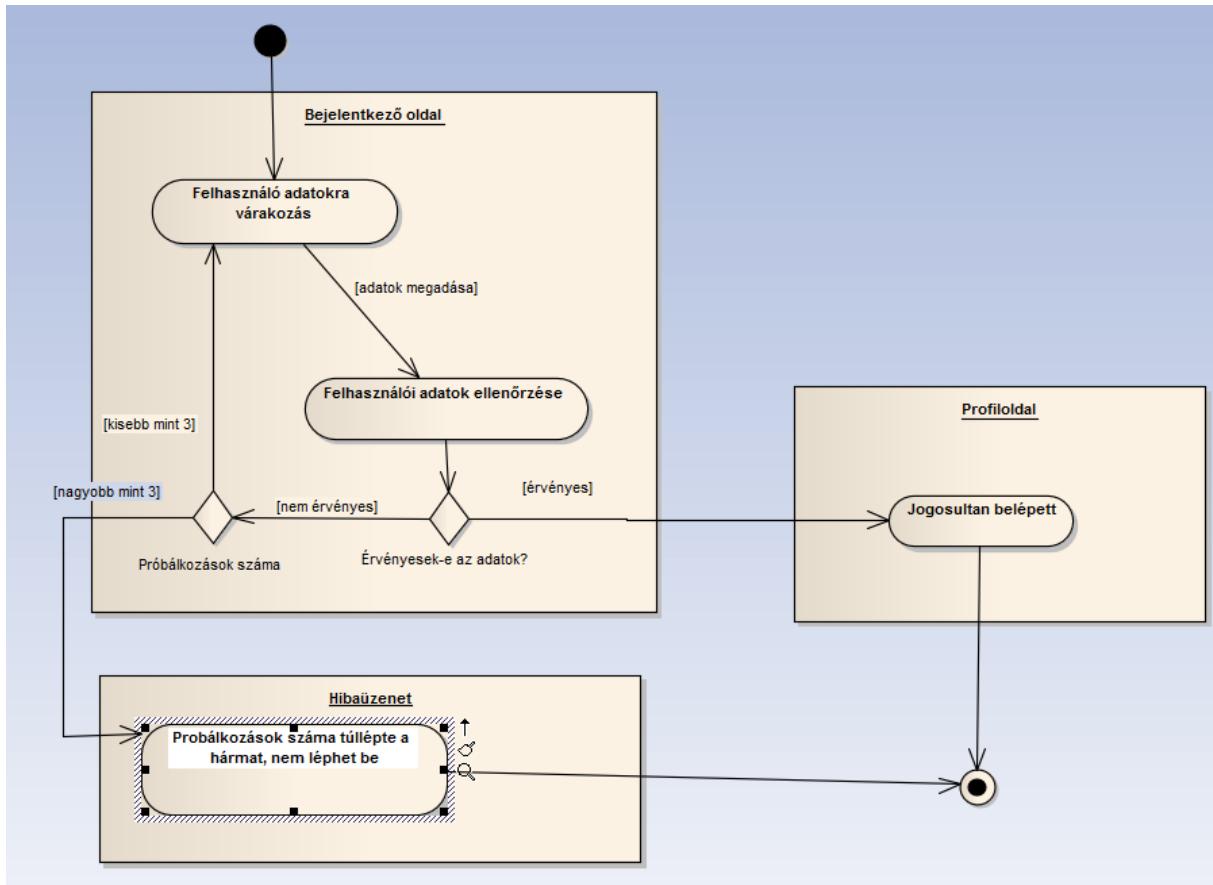
#### 7.3.1.4. Állapotgép

Az állapotgép vagy állapotdiagram segítségével szemléltethetjük, hogy egy objektum milyen állapotokon megy keresztül és milyen állapotátmeneteket alkalmaz a feladat megoldása során. Megadja, hogy egy objektum egy esemény hatására milyen állapotból milyen állapotba megy át. Két speciális állapot létezik: a kezdő- és a végállapot. E kettőnek minden állapotdiagramon szerepelnie kell.



48. ábra: Egy példa UML állapotgép diagram

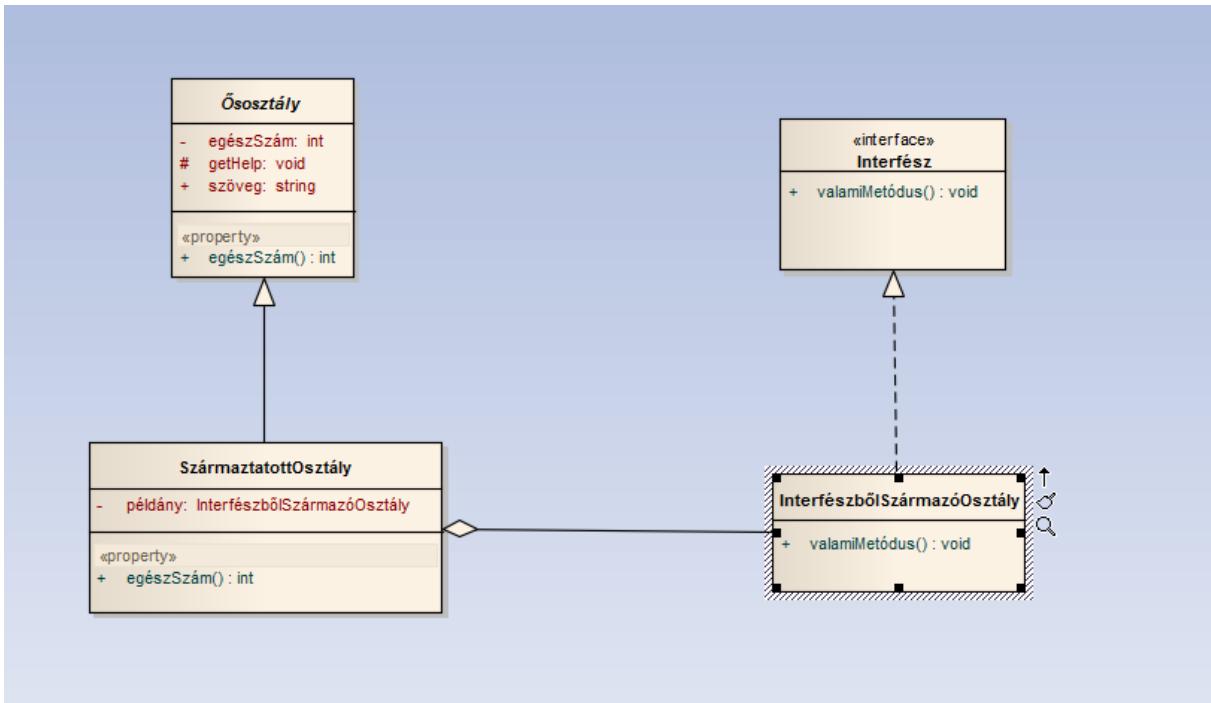
A következő ábra egy bejelentkező oldalt mutat, ahol háromszor lehet megpróbálni belépni, s ha rossz adatokat ad meg és túllépte a hármas határt, akkor egy hibaüzenetet ad, különben újra lehet próbálkozni. Ha jó adatokat adott meg a felhasználó, akkor a profiloldalára lép be.



49. ábra: Egy bejelentkező oldal UML állapotgépe

#### 7.3.1.5. Osztálydiagram

A rendszer objektumelvű leírására alkalmas. Egy statikus modell, ami a rendszerben található összes osztályt és azok közötti kapcsolatokat írja le. Ez a diagramtípus nem a felhasználók számára, hanem a fejlesztőknek készül. Általában az elkészítendő rendszer fizikai és logikai felépítésének szemléltetésére szolgál. Enterprise Architect-ben a Class Model-en belül kell létrehozni.



50. ábra: Egy példa UML osztálydiagram

### 7.3.2. StarUML

A StarUML nyílt forráskódú, ingyenes szoftver. Ez a program is UML ábrák előállítására szolgál. Jelen jegyzetben csak két lényeges tulajdonságát emeljük ki. Az első az osztálydiagramokból forráskód előállítása. A másik a tervezési minták beszúrása egy előre megadott gyűjteményből.

#### 7.3.2.1. Osztálydiagramokból forráskód előállítása

Osztálydiagramokból forráskód előállítása:

1. Indítsuk el a StarUML programot. Öt lehetőség közül választhatunk, amelyek abban különböznek, hogy milyen mappaszerkezetet kapunk az elején. Az öt lehetőség:
  - a. 4+1 Views Model
  - b. Default Approach
  - c. Rational Approach
  - d. UML Components Approach
  - e. Empty Project
2. Válasszuk a „Default Approach” nézetet. Ekkor a Model Explorerben megjelennek az alábbi mappák:
  - a. <<useCaseModel>> Use Case Model
  - b. <<analysisModel>> Analysis Model
  - c. <<designModel>> Design Model
  - d. <<implementationModel>> Implementation Model
  - e. <<deploymentModel>> Deployment Model
3. Nyissuk ki az „Design Model” mappát, azon belül klikkeljünk a Main ikonra. Erre a Toolbox ablakban megjelenik a „Class”, az „Aggregation” és a többi vizuális elem, ami szükséges egy osztálydiagram elkészítéséhez.

4. Készítsünk egy osztálydiagramot. Ha nincs ötletünk, hogy mi legyen, akkor a következő kis leírás alapján szúrunk be egy tervezési mintát.
5. Az osztálydiagramból készítsünk C# forráskódot. Ehhez a következő lépésekre van szükség:
  - a. Model menü -> Profiles... menüsor. Megjelenik a Profile Manager ablak.
  - b. C# Prifile-ra klikk, majd „Include >” gomb, majd „Close” gomb.
  - c. Tools menü -> C# menüsor -> Genarate Code... menüsor. Megjelenik a C# Code Generation ablak.
  - d. Design Model-re klikk, majd értelemszerűen Next – Next.
6. Az elkészült C# fájlokat tovább lehet fejleszteni.

Hasonlóan tudunk C++ és Java kódot is generálni az osztálydiagramokból. Ezt a módszert, amikor valamilyen tervezőszoftverből generáljuk a programunk vázát, MDA, azaz Model Driven Architecture, azaz Modell Alapú Architektúra megközelítésnek nevezzük.

Ez a megközelítés akkor teljes, ha a forráskóban történő változásokat könnyen vissza tudjuk vezetni a modellbe. Ehhez az kell, hogy a modellező eszközünk támogassa a visszamodellezést (angolul: reverse engineering), amikor is a forráskód alapján készül el az osztálydiagram és néhány más UML ábra. Erre a BOUML nevű tervezőszkőz a legalkalmasabb az internetes források szerint.

#### 7.3.2.2. Tervezési minták beszúrása

Tervezési minták beszúrása egy előre megadott gyűjteményből:

1. Indítsuk el a StarUML programot az előzőekben leírtak szerint. Klikkeljünk a Main-re a Design Model mappán belül.
2. Tools menü -> Apply Pattern... menüsor. Megjelenik az Apply pattern nevű ablak.
3. Itt nyissuk ki a „Pattern Repository”-t, ezen belül a „GoF” gyűjteményt”. Láthatóvá válnak a jól ismert tervezési minták.
4. Válasszunk egyet a listából, majd Next -> Next -> Apply.

Nagyon jó gyakorlás hibákat keresni a beépített ábrákban. Például az Egyke (Singleton) tervezési mintánál nincs jelölve, hogy az Instance() metódusnak (amit jobb lenne GetInstance() metódusnak hívni) statikusnak kell lennie. Ugyanakkor ezek a hibák javíthatók, hiszen a StarUML nyílt forráskódú.

Másik nagyon érdekes lehetőség a tervezési minták kombinálása. Például megmondhatjuk, hogy a Command minta Client osztálya legyen Singleton. Ehhez először létre kell hozni egy Command mintát, majd a Singleton minta létrehozásánál a „Pattern parameter” ablakban (tehát az első Next után) meg lehet mondani, hogy melyik osztály legyen egyke.

## 8. Összefoglalás

A programozási technológiák tárgy vezeti be a hallgatóságot az objektumorientált programozás magas szintű használatába, de ennél többet is ad, egy szemléletmódot. Ez a szemléletmód az egyszerű, de mégis a könnyen bővíthető programokat részesíti előnyben. Sokan az ilyen kódot tiszta kódnak hívják. A tiszta kód könnyen olvasható, jól karbantartott, nem okoz gondot a változtatása.

Itt a gondolatmenetben újra eljutottunk a kód változásához. A programozók legfontosabb tapasztalata, hogy a kód állandóan változik. Amikor programot fejlesztünk, akkor arra kell gondolnunk, hogy ez a kód előbb-utóbb meg is fog változni, esetleg ezeket a változásokat már egy másik programozó fogja elvégezni. Tehát, a programot nem magunknak írjuk, hanem más programozóknak, ezért olvasható kódot kell írnunk, amely követi a kiforrott megoldásokat. Ennek megfelelően érdemes tervezési mintákat, tervezési alapelveket használni, amelyeket a világon mindenhol megértenek, könnyen átlátnak.

A változásra tervezés legnagyobb fegyvere az ügyek szétválasztása (angolul: Separation of Concerns). Ez az alapelv azt mondja ki, hogy amit szét lehet választani, azt érdemes is szétválasztani. Ezért választjuk szét a stratégia tervezési mintában az osztályt és a változékony metódusát, vagy az eseményt és az esemény kezelését a megfigyelő tervezési mintában.

Az ügyek szétválasztásának egy kicsit konkrétabb megfogalmazása az SRP, amely mostanában a legelfogadottabb tervezési alapelv. Az SRP az mondja ki, hogy minden modulnak csak egy oka legyen a változásra, és ha több oka van a változásnak, akkor szét kell szedni a nagy modult (általában osztályt) kisebb részekre, aztán a részeket valamelyen módszerrel össze kell rakni.

Ennek legegyszerűbb módja az objektum-összetétel és a felelősség átadás, de vannak olyan kifinomult módszerek is, mint pl. az ágens technológia.

Ha forráskódunk rendben van, még akkor is gondolnunk kell a csapatmunkára. A kódot nem egy ember fejleszti, hanem sokan. Ezért szükség van verziókövető és hibabejelentő rendszerekre, illetve olyan dokumentációs eszközökre, mint az UML.

Ha rendben van a forráskód és a csapat is tud dolgozni, akkor még mindig kérdéses a szoftverfejlesztés folyamata. Ezt a szoftverfejlesztési módszertanok írják le, amelyek a nagyon merev vízesés modelltől a nagyon rugalmas agilis módszerekig terjednek.

Ez a jegyzet hatalmas tudásanyagot ölel fel, de így sem tér ki olyan fontos, bevált módszerekre, mint például a Kanban módszertan, vagy általánosabban a LEAN módszerekre. Nem ejt szót olyan fontos programozási technológiáról, mint az adatbázis kezelés. Nincs szó a LINQ-ról és általánosabban a név nélküli függvényekről. Fájónan kevés szó esik a keretrendszerkről, mint amilyen a Spring MVC, vagy a CodeIgniter. Ezekről ennek a jegyzetnek a folytatásában, vagy egy kibővített változatában fogunk írni.

Hiányzok tovább a GIT leírása, ami a SVN mellett egyre inkább elterjed.

Illetve nem sikerült mind a 24 tervezési mintát bemutatni, habár ez nem is volt cél, ami a GOF könyvben található. A GOF könyv után is jelentek meg remek tervezési mintákkal foglalkozó könyvek, amiket érdemes lenne feldolgozni.

Hiányoznak továbbá az úgynevezett anti tervezési minták leírása is, ugyanakkor ezek hiánya szándékos.

A sok hiány mellett örömteli kezdeményezések is vannak. A jegyzet megírása mellett sikerült megszervezni egy kis csapatot az egri Eszterházy Károly Főiskola hallgatóiból, akikkel közösen elkészítjük a tervezési minták magyar nyelvű Wikipédia weboldalát. Ebben a pillanatban a 48 ismert tervezési minta közül 27 mintának van meg a fordítása, illetve néhányat sikerült az angol eredetitől is részletsebben leírni. Ezzel a környező országok közül mindenkit lekörözünk, a német nyelvű oldallal állunk körülbelül egy szinten. A wiki oldal itt érhető el:

[https://hu.wikipedia.org/wiki/Programtervez%C3%A9si\\_minta](https://hu.wikipedia.org/wiki/Programtervez%C3%A9si_minta)

A munkához bárki csatlakozhat a Wikipédia filozófiájának megfelelően.

A fenti hiány listából egyedül azokkal a tervezési mintával foglalkozunk még röviden, amelyek kimaradtak ebből a jegyzetből, de a GOF könyvben szerepelnek. Ezek a következők:

- **Építő** (angolul: builder): Létrehozási tervezési minta, amely több lépéssben képes felépíteni egy összetett objektumot, és azt az építési folyamat végén visszaadni. Segítségével könnyen lehet olyan új építőt létrehozni, amelyben az építés lépései hasonlóak, de az építés eredménye egy kicsit más. Klasszikus példa RTF dokumentumból TeX, TXT, DOC dokumentum építése. Előnye, hogy szétválasztja az építés folyamatát és az elkészült objektum lekérdezését.
- **Híd** (angolul: bridge): Strukturális tervezési minta, amely azon a felismerésen alapszik, hogy egy osztály felülete és megvalósítása szétválasztható. Klasszikus példa a telefongyártás, az olcsó és a drága telefonba ugyanaz a lapka (megvalósítás) kerül, a drága telefonba a hibátlan, az olcsóba pedig azok, amelyek hibásak, de az alap funkciói működnek. A két telefonon más gombok (felület) vannak, a drágán több, az olcsón kevesebb. Az is könnyen elképzelhető, hogy a lapkát továbbfejlesztik, de a felület marad. Ilyenkor is működőképes telefonokat tudunk gyártani ezzel a mintával. A szétválasztás után a működőképességhoz az kell, hogy a felületet megvalósító példányba becsomagoljuk a megvalósítást megvalósító példányt. Innentől a feladat ugyanaz, mint az illesztő tervezési minta esetén.
- **Összetétel** (angolul: Composite): Strukturális tervezési minta, amely lehetővé teszi rekurzív adatszerkezetek használatát, mint amilyen a lista, a fa, vagy a gráf, azzal a céllal, hogy a kliens egységesen tudja kezelni az adatszerkezet egészét, egy részét vagy csak egy elemét. Klasszikus példa a rajzolás az Office-ban. Rajzunk néhány alakzatot és ezeket csoportba foglaljuk. Innentől fogva a csoportban lévő alakzatokat együtt forgathatjuk, nagyíthatjuk. Megvalósítása aggregációval történik, a csoport az elemeinek aggregációja. Nagyon jó példa az átlátszó becsomagolásra. A díszítő mintától csak annyiban különbözik, hogy a dísz minden részt csomagol be, az összetétel nulla vagy több részt csomagol be. A metódusait gyakran reukrúziv kell leprogramozni, hiszen egy rekurzív adatszerkezetet ábrázol. Ennek a mintának az a különlegessége, hogy nem szétválaszt, mint sok más minta, hanem egységesít, egységesíti a rész és az egész kezelésének módját.
- **Homlokzat** (angolul: Facade): Strukturális tervezési minta, amely szétválaszt egy bonyolult alrendszeret annak klienseitől úgy, hogy egy közös, könnyen érthető felületet ad az alrendszernek. Többrétegű programozás esetén minden rétegnek van homlokzata, amelyen keresztül lehet vele kommunikálni. A homlokzat aggregálja az alrendszer elemeit, a homlokzat hívásait átfordítja a megfelelő alrendszer hívássokká. Mivel a homlokzat elrejt egy komplex rendszert egy kevésbé

komplex felület mögé, ezért a szivárgó absztrakció elve miatt nehezen megérhető hibákat okozhat.

- Pehelysúlyú (angolul: Flyweight): Strukturális tervezési minta, amely egy objektum belső állapotát szedi szét tényleges belső állapotra és külső állapotra, azért hogy a pehelysúlyú objektum széles körben megosztható legyen. Klasszikus példa a szövegszerkesztők betű kezelése. Egy betűnek van kódja, mérete, betűtípusa. Ezek közül a tényleges belső állapotban elegendő csak a kódot tárolni, a méret mehet a betű környezetét leíró objektumokba, azaz a külső állapotba. Így minden 'A' betű ugyanaz az objektum lesz, nem kell minden 'A' betűhöz külön objektumot létrehozni, ami hatalmas memória megtakarítást eredményez. Ilyenkor az objektum metódusainak meg kell kapniuk a környezetüket tartalmazó objektumot, hogy el tudják látni feladatukat. A szövegszerkesztők példánál a betű KiRajzol metódusa megkapja a betű közvetlen környezetét alkotó szövegszerkesztési egységet, például a bekezdést. A környezetnek is lehet környezete, ami befolyásolhatja a kirajzolást, tehát a betű környezetet leíró objektumnak tudnia kell, azt őt becsomagoló szövegszerkesztési egységről. A betű külső állapota a környezetében található, így a betű lehet igazán pehelysúlyú és ezért kell csak 1 darab 'A' betű objektum, amelyen a szövegben lévő minden 'A' betű osztozik.
- Felelősséglánc (angolul: Chain of responsibility): Viselkedési tervezési minta, amely egy láncon felfelé haladva keresi az első olyan objektumot, amely képes ellátni egy feladatot, amit megkapott a felelősséglánc. Klasszikus példa az F1 gomb lenyomása, amellyel a felhasználó segítséget kér. Ilyenkor felelősséglánc azon a vizuális elemnél kezdődik, amely éppen aktív volt az F1 gomb lenyomásakor. Ha van hozzá segítség rendelve, akkor az megjelenik. Ha nincs, akkor ezt a vizuális elemet tartalmazó elem kapja meg a kérést. Ha képes kiszolgálni a kérést, akkor kész vagyunk, ha nem, akkor a kérést a láncon felfelé tovább kell adni. Ezt addig kell ismételni, míg vagy nem sikerül a kérést kielégíteni, vagy el nem érjük a lánc végét. A felelősségláncot nagyon könnyű megérteni, ha valaki már érti a kivétel kezelést, hiszen ott is a hívási láncban visszafelé terjed a kivétel, amíg valaki el nem kapja, vagy el nem ér legfelső szintre, ahol futási hibát okoz.
- Parancs (angolul: Command): Viselkedési tervezési minta, amely egy metódust hívást zár egységbe és azt meghívhatóvá teszi egy előre megadott felületen, általában a Végrehajt (angolul: Execute) híváson keresztül. Ezzel a módszerrel a metódus hívást üzenetté alakítjuk át ennek minden előnyével, például az üzenet feladója és címzettje könnyen szétválasztható. A megvalósításához kell egy parancs interfész, amiben csak egy metódus van, a Végrehajt. A konkrét parancs osztályok ezt az interfészt valósítják meg, a Végrehajt metódus megvalósítása adja a parancs viselkedését. Ez azt jelenti, hogy egy konkrét hívást zárunk be egy parancs osztályba. Ez hasonló, mint az egységeszt, ahol szintén egy konkrét metódus hívást tesztelünk. Ennek az az értelme, hogy az ilyen parancs osztályokból készült parancs objektumok nagy rugalmasságot adnak a „lefagyaszott” metódus hívás jobbra-balra küldésére a rendszerben, anélkül hogy a parancs készítőjén kívül bárki is tudná, hogy mi fog történni, ha valaki meghívja a Végrehajt metódust. Ezzel a tervezési mintával könnyen megvalósítható a visszavonható parancs, illetve erre épülve a tranzakció kezelés.
- Értelmező (angolul: Interpreter): Viselkedési tervezési minta, amely szakterület specifikus nyelvek készítésére (angolul: Domain Specific Language, röviden: DSL) készítésére alkalmas. Érdemes valamely DSL keretrendszert használni erre, mint amilyen az ANTLR (Another Tool For Language Recognition) Java esetén az Xtext.
- Bejáró vagy Iterátor (angolul: Iterator): Viselkedési tervezési minta, amely egy adatszerkezet bejárását támogatja. A modern nyelvek adnak iterátort a beépített adatszerkezeteiket és

támogatják saját iterátor írását. A használata gyakran nem indokolt, hiszen a beépített adatszerkezeteket nagyon kényelmesen fel lehet dolgozni például foreach ciklussal.

- Közvetítő (angolul: Mediator): Viselkedési tervezési minta, amely központosítja az objektumok kommunikációját. A közvetítő bevezetését azt teszi indokolttá, hogy ha már átláthatatlan, hogy a sok objektum hogyan kommunikál egymással. A közvetítő bevezetése után az alrendszer minden objektuma a közvetítőn keresztül kommunikál az alrendszer többi objektumával. Ahhoz, hogy ez lehetséges legyen, a közvetítőnek ismernie kell az objektumokat. Erre gyakran egy regisztrációs folyamat szolgál. Gyakran a kommunikáció nem központosítható, mert túlságosan lelassítaná a rendszert.
- Emlékeztető (angolul: Memento): Viselkedési tervezési minta, amely pillanatképet készít az objektum belső állapotáról, azt kiírja egy perzisztens tárolóba (állományba, vagy adatbázisba) és ha kell, visszatölti. Legegyszerűbb formája a szerializáció (angolul: serialization), fejlett változata az objektum-relációs leképezés (angolul: object-relational mapping, röviden: ORM) rendszerek, mint amilyen Java esetén a Hibernate, .NET esetén az NHibernate.

## 9. Melléklet

### 9.1. Jegyzőkönyvsablon

# Jegyzőkönyv

Megbeszélés témája:

Helyszín:

Dátum / Időpont:

Készítette:

Kapja:                   Minden résztvevő

Következő időpont:

Résztvevők:

Ügyfél		Szoftverfejlesztő cég	
Név	Projektbeosztás	Név	Projektbeosztás

### 9.2. Példakérdőívek irányított riorthoz

#### 9.2.1. Példa 1.

Interjúkérdezések az elektronikus közgyűlés rendszer-követelményspecifikációjának felállításához. Kérjük, lehető legjobb tudása szerint töltse ki. A kérdezőbiztos nem válaszolhat kérdéseire, hogy ne befolyásolja Önt.

Mit gondol, mit jelent az elektronikus közgyűlés?

Milyen előnyöket nyújt Ön szerint az elektronikus közgyűlés?

Milyen biztonsági szintet kell kielégítenie Ön szerint egy elektronikusközgyűlés-rendszernek?

Milyen fő funkciókat vár el egy ilyen rendszertől?

Milyen „vágylom” funkciókat látna szívesen egy ilyen rendszertől?

Kérem, írja körül, hogy jelenleg hogyan megy Önknél egy közgyűlés! Milyen nehézségek adódtak ezeken?

Kérem, soroljon előnyöket / hátrányokat a személyes közgyűlés és az online közgyűlés relációjában!

Az elektronikus közgyűlésnél milyen szintű online megjelenést vár el a résztvevőktől? Legyen lehetőség azonnali üzenetküldésre, hangkapcsolatra vagy videó kapcsolatra? Elegendő, ha egy avatar személyesíti meg a résztvevőket, vagy szeretné látni az arcukat is?

Mit gondol, a Skype, MSN vagy egyéb azonnali üzenetküldő lehet alapja egy e-közgyűlés-rendszernek?

Milyen ablakok legyenek a rendszerben? Kérem, adjon képernyőterveket!

Hallott már hasonló szolgáltatásról? Esetleg használja valamelyiket? Milyen tapasztalatai vannak?

Ön szerint szükség van egy ilyen rendszere?

Fizetne Ön egy ilyen rendszerért? Ön szerint ennek egy egyszer megveszem, aztán használom alkalmazásnak kell lennie, egy előfizetéses szolgáltatásnak vagy használat alapján fizetettnek alapdíjjal?

Milyen költségei vannak a közgyűlésekkel kapcsolatban? Ön szerint csökkennének vagy nőnének ezek a költségek egy elektronikus közgyűrésrendszer esetén?

Ön szerint mekkora az a cégméret, tulajdonosikör-méret, ami mellett már mindenkor szükséges egy ilyen rendszer?

#### 9.2.2. Példa 2.

##### Kérdőív eFilter rendszer fejlesztéséhez (orvosi)

Az eFilter rendszer célkitűzése:

- Személyes egészségügyi adatok (ételérzékenységek, allergiák, betegségek, ezeknek a foka (mennyire cukorbeteg) stb.)
- személyre szabott diéták (napi energia, fehérje, szénhidrát bevitel stb.)
- élelmiszerre vonatkozó adatok (összetevők, név, gyárt, kiszerelés stb.)

alapján a következő tevékenységek támogatása:

- fogysztható élelmiszerek listázása,
- döntés segítése, hogy egy élelmiszer fogysztható-e vagy sem.

Kérem, hogy az eFilter rendszer sikeres kifejlesztése érdekében válaszoljon az alábbi kérdésekre legjobb tudása szerint. Ha kérdése van, nyugodtan tegye fel a kérdezőbiztosnak. Előre is köszönjük együttműködését!

1. Hogyan azonosít az orvosszakma betegségeket, allergiákat, ételérzékenységeket? Pl. BNO kód.
2. Milyen más élelmiszer-bevitelt korlátozó jelenséget ismer az orvosszakma, amely se nem betegség, se nem allergia, se nem ételérzékenység? Pl. szindróma. Ha igen, miben különböznek ezek?
3. Az allergia egy speciális betegség vagy betegsékként nem fogható fel, mert annyira különböző tulajdonságokkal bír? Az ételérzékenység egy speciális betegség?
4. Akinek mogyoróallergiája van, az egyáltalán nem ehet mogyorót?
5. Hogyan írható le egy allergia? Pl. erőssége egytől öt keresztig.
6. Egységesen leírhatók-e az élelmiszer-allergiák, ételérzékenységek és az élelmiszer-bevitelt korlátozó betegségek? Hogyan?
7. Melyek az élelmiszer-bevitelkötő betegségek? Pl. magas vérnyomás, cukorbetegség stb.
8. Hogyan írható le egy élelmiszer-bevitelt korlátozó betegség? Pl. A típusos cukorbetegség.
9. Akinek mogyoró allergiája van, az ehet-e mandulát? Tehát aki allergiás valamire, az ehet ehhez hasonló élelmiszereket?

10. Mi a különbség az étrend és a diéta közt? A diéta az csak egy diétás étrend?
11. Ugyanolyan betegségre (allergiára stb.) lehet-e többféle diéta? Ezek közül hogyan választ a szakértő?
12. Aki penicillinérzékeny, az ehet penészes sajtot?
13. Mennyire lehet pontosan leírni, hogy bizonyos allergiák, ételérzékenységek, betegségekben szenvedők miből mennyit ehetsz? Vagy ez mindenkor csak személyre szabhatóan dönthető el?
14. Miből áll egy diéta? Pl. mennyiségek, időbeli megszorítások stb.... .
15. Lehet olyan egy diétában, hogy vagy csak ezt ehetek, vagy csak azt?
16. Mennyire gyakoriak és milyen szerepük van a keresztallergiáknak (se paradicsomra, se parlagfűre nem vagyok allergiás, de parlagfű szezonban nem ehetek paradicsomot)? Fel kell erre készíteni a rendszert? Ilyen előfordul gyógyszerek esetében is?
17. Egy diéta csak megszorításokból áll? Például napi fehérjebevitel maximum 100g.
18. Egy élelmiszernél okozhat gondot a csomagolása is?
19. Az élelmiszerek és a folyadékok (pl. almalé) külön kezelendők egy diéta esetén? Általában külön kezelendők-e az élelmiszerek és a folyadékok? Ha igen, milyen más kategóriák vannak még?
20. Egy betegség esetén vannak javallatok is vagy csak ellenjavallatok? Pl. magas vérnyomás esetén egyen sok gyümölcsöt.
21. Milyen más adatokat kell nyilvántartani egy élelmiszerrel az összetevőkön túl? Pl. kiszerelés, csomagolás, név, gyártó stb.
22. Ismer-e nyilvánosan hozzáérhető élelmiszer-összetevő / kész élelmiszer adatbázisokat?
23. Érdekes-e, hogy a „várvédők kedvence” étlapí ételben lévő húst 150 vagy 200 fokom süttötték? Azaz érdekes-e az élelmiszer elkészítésének módja?

#### 9.2.3. Példa 3., riportozó alrendszer kialakítása

Projekt megnevezése	
Dokumentumazonosító	
Dátum	
Dokumentum készítője	
Tagvállalat neve	

#### 9.2.3.1. Riport általános adatai

Riport megnevezése	
Terület/osztály megnevezése (kontrolling, értékesítés stb.)	
Riport formátuma (pl.: MS Excel, TSV, CSV, PDF, képernyő stb.)	

Lefuttatás módja (felület, automatikus)	
Automatikus riport esetén a lefutás időpontja (pl. minden nap 2:00) vagy a kiváltó esemény megnevezése (valamelyen rendszerbeli esemény/funkció megnevezése)	

#### 9.2.3.2. Riport paraméterei

A lekérdezés elvégzésének paraméterei. Pl. időszak kezdete, vége, ügyfélazonosító, számlaszám stb. (Paraméter lehet a hagyományos szűrőfeltételeken kívül, akár egy a riport számítási algoritmusát befolyásoló egyéb adat, pl. egy statisztikai riport esetén, az adatokat havi vagy negyedéves bontásban szeretnénk-e megkapni.)

	Paraméter megnevezése	Paraméter típusa (szöveg, szám, dátum, kiválasztandó adat stb.)
1.		
2.		
3.		

#### 9.2.3.3. Riport adatmezői

A riport végeredményeképpen előálló lista adatmezői.

	Mező megnevezése (ügyfél neve, ügyfél számlázási címe, ügyfél azonosító stb.)	Mező típusa (szöveg, szám, dátum, kiválasztandó adat stb.)
1.		
2.		
3.		

#### 9.2.3.4. Szöveges leírás

Kérem, írja le a szövegesen, mint vár a riporttól.

#### 9.2.3.5. Mellékletek listája

A mellékletek listája tartalmazza az átadott dokumentumokat. Ezek lehetnek a jelenlegi rendszerből kapott listák vagy egyéb más, az adott riporttal kapcsolatba hozható dokumentumok.

	Melléklet megnevezése	Melléklet leírása	Csatolt dokumentumok (digitális fájlok) A fájlokat a Beszúrás/Objektum funkcióval kell hozzáadni
1.			
2.			
3.			

### 9.3. Követelménylista-sablon

A felmerült követelmények listája alább látható. A V. oszlopban található szám verziószám. Azt mutatja, hogy melyik verzióban kell legkésőbb megjelenni a funkciónak. Tehát 1.0 esetén már az első verzióban is benne kell lenni, 2.0 esetén a funkció megjelenhet az 1.3 vagy 1.8-as verzióban is, de legkésőbb a 2.0-ban. A 0.1-es verzió fenntartott az architekturális követelményeknek.

A listában szerepelnek funkcionális és nemfunkcionális követelmények is. Egyes követelmények esetleg ellentmondanak egymásnak, de ilyenkor ezek sosem egy verzióra lettek tervezve. A követelménylista:

Modul	ID	Név	V.	Kifejtés
Jogosultság	K1	Beléptetés	0.8	A felhasználó a jelszavának megadásával beléphet a rendszerbe. Ha rossz a jelszó, akkor a következő üzenetet írja ki: „Rossz jelszó! Próbáld újra.” Csak maximum 3x lehet megpróbálni a jelszót, utána 1 óráig letiltásra kerül az adott felhasználó.  Ha a felhasználó új felhasználó, akkor kérhessen jelszót.

### 9.4. Példaütemterv

Funkció / Story	Feladat / Taszk	Prioritás	Becslés	Akt. becslés	Eltel t	Hátralévő
Funkcionális specifikáció		0	24	24	24	0
Form megvalósítása		1	20	20	0	20
A szűrők elkészítése	Gyártó	1	12	12	0	12
	Gyártói kód	1	12	12	0	12
	Típus	1	12	12	0	12
	Altípus	1	12	12	0	12
	Dátum	1	12	12	0	12
	Méret	1	12	12	0	12
	%-os túrés	2	12	12	0	12
	OE szám	2	12	12	0	12
	OE összekapcsoló	2	12	12	0	12

Tételek panel	Listázás	1	8	8	0	8
	Csoportosítás	2	16	16	0	16
Képek panel	Megvalósítás	1	12	12	0	12
	Összehangolás a Tételek ablakkal	2	6	6	0	6
Összekapcsolás	Megvalósítás	1	10	10	0	10
	Csoportbontás	2	8	8	0	8
	Csoportból kiszedés	2	8	8	0	8
Navision cikkszám hozzárendelés		2	4	4	0	4
Tartalék idő		3	16	16	0	16
Tesztelés		1	40	40	0	40
Projektvezetés		2	30	30	0	30
	Órák:	310	310	24	286	
	Embernap:	38,75	38,75	3	35,75	
Napidíj:	28 000 Ft					
Árajánlat:	1 085 000 Ft					

## 9.5. Tesztpéldák

### 9.5.1. Funkcionális tesztpélda

#### 9.5.1.1. Bevezetés

A Rendszerfunkcionális Teszt (RFT) a rendszer egészének - beleértve a környezeti eljárásokat is - alapos tesztelése. A rendszerfunkcionális tesztelést előre definiált tesztadatokkal, ahol lehetséges, tesztelő eszközzel kell végezni.

A tesztelések célja a rendszer és komponensei funkcionalitásának teljes körű vizsgálata, ellenőrzése, a rendszer által megvalósított üzleti szolgáltatások verifikálása.

A jelen RFT terv alapján elvégzett tesztelés folyamatát az RFT jegyzőkönyv tartalmazza. Az elkészült jegyzőkönyv alapján a Tesztelési Vezető feladata eldöntheti, hogy a tesztelés sikeresnek minősíthető, vagy meg kell ismételni részben vagy egészben a tesztet.

#### 9.5.1.2. Tesztelési eljárások

Teszteset-azonosító: DATA1.DEF (1)

A funkciót adott időre történő táblalekérdezéssel teszteljük. A tesztprogram törzsadat-lekérdező felhasználói felületén kérjük le a BNO táblát egy olyan időpontra, melyre van az adatbázisban feltöltött adat (pl. a BNO első feltöltése, illetve BNO új verzió feltöltése folyamat során belekerült táblák). A lekérés hatására megkapjuk az adott időpontban érvényes táblát a tábla verziószámával együtt. Ellenőrizzük, hogy megfelelnek-e az értékek a korábban az adatbázisba feltöltött adatoknak.

Az eljárás sikeres, ha a visszakapott adatok megfelelnek a korábban az adatbázisba importált adatoknak.

#### Teszteset-azonosító: DATA1.ALT (1)

A funkciót adott időre történő táblalekérdezéssel teszteljük. A tesztprogram törzsadat-lekérdező felhasználói felületén kérjük le a BNO táblát egy olyan időpontra, melyre nincs az adatbázisban feltöltött adat. A lekérés hatására nem szabad adatot visszakapnunk, ennek megfelelő választ kell a tesztelő programnak adnia a kérésre.

Az eljárás sikeres, ha nem kapunk vissza adatot (pl. egy más időpont adatát), csak egy olyan hibaüzenetet, amiből egyértelműen kiderül, hogy a keresés feltételeinek nincs megfelelő tábla.

#### Teszteset-azonosító: DATA2.DEF (1)

A tesztprogram törzsadat-lekérdező felhasználói felületén kérjük le a legfrissebb BNO táblát (ez úgy történik, hogy a táblaparaméterek közül egyiket sem adjuk meg). A lekérés hatására megkapott adatokat vessük össze a táblába importált törzsadattal.

Az eljárás sikeres, ha a visszakapott adatok megfelelnek a korábban (pl. a BNO első feltöltése, illetve BNO új verzió feltöltése folyamat során) az adatbázisba importált adatoknak.

#### Teszteset-azonosító: DATA2.ALT (1)

A tesztprogram törzsadat-lekérdező felhasználói felületén kíséreljünk meg olyan törzsadatot lekérni, ami nem szerepel a rendszer adatbázisában (pl. nem létező verzió). Hibajelzést kell kapnunk.

Az eljárás sikeres, ha nem kapunk vissza adatot, csak egy olyan hibaüzenetet, amiből egyértelműen kiderül, hogy a keresés feltételeinek nincs megfelelő tábla.

### 9.5.2. [Teszttervpélda](#)

#### 9.5.2.1. [Elvárások](#)

Jelen dokumentum célja, hogy az XX rendszer megvalósítása projektben a rendszerek tesztelési elvárásait ismertesse.

A projekt sikeres befejezésének eszköze a tesztelési terv és a tesztelési jegyzőkönyvek. A tesztelési jegyzőkönyvekben részletezett tulajdonságok megfelelése, valamint a Szerződésben hivatkozott dokumentációk átadása esetén kerül sor a Szerződésben szereplő végteljesítés-igazolás Végső Kedvezményezett általi ellenjegyzésre.

Összefoglalásként kiemelhető, hogy akkor tekinthető egy adott részrendszer tesztelése sikeresnek, ha

- a tesztelési jegyzőkönyvek mezői 98%-ban MEGFELELT minősítésűek, azaz
- a specifikációban elfogadott funkciók működnek (a rendszer az előre definiált eredményt adja előre definiált bemeneti adatok esetében – funkcionális teszt),

- o a rendszerfunkciók specifikált paramétereinek mért értékei az elvárásoknak megfelelő teljesítmény-határértékek között vannak (mért válaszidő vagy elvégzési idő kisebb az előre definiált válasz- vagy elvégzési időnél – terheléses teszt).

A rendszertesztesztelési terv általánosan ismerteti a tesztelés folyamatát, valamint a tesztjegyzőkönyvek minimálisan szükséges, részrendszer-specifikus adattartalmát. Mellékletekként jelennek meg a tesztjegyzőkönyvek.

#### 9.5.2.2. Tesztelés menete

Rendszerteszteszteléshez kapcsolódó határidők

Sorszám	Tevékenység részletezése	Határidő
1.	A kifejlesztett funkcionális szállító oldali tesztelésének lezárása, az átadás-átvételi folyamat megkezdése.	
2.	A rendszerek átadás-átvételi folyamatának lezárása. (Rendszerspecifikus átadás-átvételi tesztek elvégzése, eredmények ellenőrzése és ellenjegyzése.)	
3.	Az üzembe állításhoz szükséges dokumentációk elkészítése (felhasználói kézikönyv, üzemeltetési dokumentáció, módszertani útmutató).	
4.	Az üzembe állításhoz szükséges dokumentációk Megrendelőnek történő átadása (felhasználói kézikönyv, üzemeltetési dokumentáció, módszertani útmutató).	
5.	A kifejlesztett funkcionális éles üzembe állítása.	

Tesztelési folyamat leírása

A tesztelési folyamat vázát adjuk meg egy lista formájában:

- előzetes tesztek validálása, elfogadása: termék megfelelőségek vizsgálata, összevetés az adott specifikációkkal (amennyire lehet, ki kell terjednie a felhasznált eszközök szintjére),
- tesztelési folyamatok ellenőrzése, elfogadása,
- tesztek műszaki kiértékelése – a kapott eredmények megfelelősségeinek vizsgálata.
- mérőrendszerek tesztelése – tartalmi és hitelességi szempontok alapján.

A kialakított rendszerek tesztelését részben a szoftverfejlesztők, részben az erre szakosodott tesztelők végezik a tesztelési tervezek alapján. A tesztelési tervezek kiternek az elvégzendő fejlesztői, szervezői, felhasználói stb. funkcionális, teljesítmény- és egyéb tesztek folyamatára, ütemezésére, valamint a tesztadatok feltöltésére. A tesztelés funkcionális tesztelést, integrációs tesztelést és terheléses tesztelést jelent. A tesztek fő szempontja funkcionális és technikai ellenőrzés, melyeket követően a szükséges javítások elvégzésére kerül sor. A tesztek rövidített jegyzőkönyvek készülnek.

Az átadás-átvételi tesztek megkezdését kezdeményezni a sikeres, előzetesen Végső kedvezményezettnek átadott és előzetesen egyeztetett tesztelési metódus alapján végzett vállalkozói tesztelési jegyzőkönyvek birtokában lehetséges.

Az átadás-átvételi tesztek lebonyolítása Végső kedvezményezett minőségbiztosítójának a feladata, itt a Vállalkozó részéről csak a tesztekben való közreműködés és a feltárt hibák elhárítása az elvárt. A teszteket a Minőségbiztosítási tervnek megfelelően kell elvégezni.

Az átadás-átvétel akkor tekinthető sikeresnek, ha az átadás-átvételre felajánlott rendszerek a Rendszertervezekben, valamint a Minőségbiztosítási tervben foglalt feltételeknek megfelelnek.

#### 9.5.2.3. Tesztelési típusok részrendszer vonatkozásában

##### Funkciótesztek

- Általános funkcionális teszt
  - A rendszer működésének vizsgálata normál működés esetén. A teszt során ellenőrizzük, hogy a rendszer funkciói az elvártnak megfelelően működnek, a teszt során a kívánt eredményeket kapjuk.
- Szélsőérték funkcionális teszt
  - A rendszer működésének vizsgálata szélső bemeneti/kimeneti értékek esetén. A teszt során ellenőrizzük, hogy a rendszer funkciói az elvártnak megfelelően működnek, a teszt során a kívánt eredményeket kapjuk.
- Biztonsági teszt
  - A szoftver jogosultsági rendszerének tesztelése ellenőrzi, hogy a rendszer adataihoz csak az elvárt felhasználók férnek-e hozzá.
- Telepítési és rendszer-visszaállítási teszt
  - A rendszer telepítésének tesztelése a rendszerüzemeltetési leírás alapján
  - Rendszer visszaállítása (program és adat) a rendszerüzemeltetési leírás alapján

##### Teljesítménytesztek

A teljesítménytesztek során a rendszer teljesítőképességének tesztelése történik nagy mennyiségű bemenő, kimenő adat, ill. adatbázis, valamint nagy mennyiségű egyidejű felhasználó esetén.

#### 9.5.2.4. Tesztelési típusok projektrésztvevők vonatkozásában

A informatikai rendszerek fejlesztése során a tesztet végző személyek, szervezetek különböző mélységen tesztelik a rendszereket.

A Végső Kedvezményezett által végzett átvételi tesztekben a specifikációban megfogalmazott funkcióknak kell megfelelni (a rendszer az előre specifikált eredményt adja specifikált teljesítményhatárok között).

### Fejlesztői tesztek

A fejlesztői tesztelés során funkcionális és teljesítményteszteket is kell végezni. A tesztelést a rendszer szállítója végzi.

A fejlesztői tesztek akkor tekinthetők befejezettnak, ha a tesztelés során feltárt összes hiba kijavításra került, vagy csak olyan hibák maradtak, amelyek mellett az átadás-átvételi tesztek megkezdése nem okozhat problémát sem az adott alkalmazásban, sem pedig az érintett külső rendszerekben.

A fejlesztés folyamatában az egyes modulok funkcióinként is tesztelésre kerülnek.

### Átadás-átvételi tesztek

A leszállított rendszerek átvételhez szükséges teljesítménytesztjét Végső kedvezményezett minőségbiztosítója végzi, amelyhez a szükséges közreműködést Vállalkozó mind a teljesítménytesztek tervezése, minden a tesztek végrehajtása során biztosítja.

#### 9.5.2.5. Rendszerek átvételének sikerkritériuma

##### Az elfogadható működőképesség ismérvei

Tehát akkor tekinthető egy adott részrendszer tesztelése sikeresnek, ha

- a tesztelési jegyzőkönyvek mezői 98%-ban MEGFELELT minősítésűek, azaz
- a specifikációban a Felek által kölcsönösen elfogadott funkciók működnek (a rendszer az előre definiált eredményt adja előre definiált bemeneti adatok esetében – funkcionális teszt),
- a rendszerfunkciók specifikált paramétereinek mért értékei az elvárásoknak megfelelő teljesítmény-határértékek között vannak (mért válaszidő vagy elvégzési idő kisebb az előre definiált válasz- vagy elvégzési időnél – terheléses teszt).
- Felhasználói és rendszerüzemeltetési kézikönyv megfelelő minőségen átadásra kerül.

#### 9.5.3. Funkciótesztjegyzőkönyv-minta

Rendszer megnevezése	
Tesztelés várható időtartama	
Tesztelés erőforrás-szükséglete Vállalkozó oldalon	
Tesztelés erőforrás-szükséglete Végső Kedvezményezett oldalán	

Vállalkozó részéről:

Aláírás:

Név:

Beosztás:

Dátum:

Végső Kedvezményezett részéről:

Aláírás:

Név:

Beosztás:

Dátum:

Sorszám	Funkció leírása	Vizsgálat módja/eszköze	Elvárt eredmény	Eredmény	Megfelelősség státusza [Megfelelő, Pótlás határideje:]
<b>Általános funkcionális teszt</b>					
1.					
2.					
<b>Szélsőérték funkcionális teszt</b>					
1.					
2.					
<b>Biztonsági teszt</b>					
1.					
2.					
<b>Telepítési és rendszer-visszaállítási teszt</b>					
1.					

2.					
----	--	--	--	--	--

#### 9.5.4. Teljesítményteszt jegyzőkönyvmintája

Rendszer megnevezése	
Tesztelés várható időtartama	
Tesztelés erőforrás-szükséglete Vállalkozó oldalon	
Tesztelés erőforrás-szükséglete Végső Kedvezményezett oldalán	

Átvételi eljáráson résztvevők

Vállalkozó részéről:

Aláírás:

Név:

Beosztás:

Dátum:

Végső Kedvezményezett részéről:

Aláírás:

Név:

Beosztás:

Dátum:

Sorszám	Funkció leírása	Vizsgálat módja/eszköze, részletes leírása	Elvárt eredmény [válaszidő, végrehajtási idő, egységnyi idő alatt végrehajtott tranzakció]	Eredmény	Megfelelősség státusza [Megfelelő, Pótlás határideje:]
1.					

## 9.6. Kulcsfogalmak

Architektúra: A program azon része, ami nem változik az idő során, vagy ha változik, akkor az nagyon nehezen kivitelezhető.

Delegálás: Felelősség átadás, vagy felelősség delegálás alatt azt értjük, hogy egy objektum valamely metódusa meghívja az általa birtokolt egyik objektum, azaz egy becsomagolt objektum, metódusát, hogy az helyette oldja meg a feladatot részben vagy egészben.

Egységeszt vagy unit-teszt: Az egységeszt vagy más néven unit-teszt a metódusok tesztje. Egy unit-tesztben rögzítjük a metódus paramétereit, illetve megmondjuk, hogy erre a bemenetre mi az elvárt kimenet.

Felelősség beinjektálás: Felelősség beinjektálás (angolul: dependency injection) alatt azt értjük, hogy az objektum-összetétel által birtokolt objektumot az objektum kívülről, általában a konstrukturán vagy egy „setter” metóduson keresztül kapja meg.

GOF: A GOF szó az angol „Gang of Four” rövidítése, magyarul a négyek bandája, amely a Programtervezési minták könyv 4 szerzőjére utal. A GOF könyv magára a Programtervezési minták című könyvre utal.

Klónozás: A klónozás (angolul: clone) az a programozási technika, amikor a klónozandó objektummal teljesen megegyező új objektumot hozunk létre, azaz a két objektum belső állapota ugyanaz lesz. Az új objektum részben vagy teljesen független az a klónozott objektumtól. Két fajtája van, a sekély klónozás (angolul: shallow copy), és mély klónozás (angolul: deep copy).

Klónozás - Mély klónozás: A mély klónozás (angolul: deep copy) a klónozásnak az a fajtája, amikor az eredeti objektum referencia típusú mezőit is klónozzuk (a két referencia nem ugyanoda mutat), így az új klón teljesen független lesz az eredeti objektumtól. A megváltoztathatatlan (angolul: immutable) mezőket, mint például a string típusúakat, nem érdemes klónozni.

Klónozás - Sekély klónozás: A sekély klónozás (angolul: shallow copy) a klónozásnak az a fajtája, amikor az eredeti objektum referencia típusú mezőit csak másoljuk (a két referencia ugyanoda fog mutatni), így az új klón csak részben lesz független az eredeti objektumtól.

Kontrol megfordítása: A kontrol megfordítása (angolul: Inversion of Control, vagy röviden IoC) az a programozási módszer, amikor nem a kosztum kód hívja az előre megírt általános kódot, pl. egy könyvtár (angolul: library) függvényt, hanem az általános kód hívja a kosztum kódot. Egyik példája, amikor nem a gyermekosztály hívja az ősosztályt, hanem az ős a gyermeket.

Objektum: Az objektum valamely osztálynak a példánya. Az objektumnak van belső állapota, felülete és viselkedése. Belső állapotát a mezőinek értéke írja le. Felülete megegyezik osztályának a felületével. Mivel egy objektum több osztály példányaként is használható, ezért több felülete van. Viselkedése megegyezik az osztályának megvalósításával, a futó foráskódot nevezzük videlkedésnek.

Objektum-összetétel: Az objektum-összetétel azt jelenti, egy osztálynak van egy olyan példány szintű mezője, amely egy másik objektumra mutat, ennek az osztálynak az objektumai egy másik objektumot tartalmaznak. Angol elnevezése HAS-A kapcsolat, magyarul VAN-EGY kapcsolat. UML jelölése:

Rombuszból induló nyíl, a rombusz a tartalmazó osztályhoz kapcsolódik, a nyíl a tartalmazottra mutat. Az objektum-összetétel az öröklődés alternatívája, amit meglehet oldani öröklődéssel, az megoldható objektum-összetételel is, kivéve a típus kompatibilitást.

**Objektum-összetétel - Aggregáció:** Az aggregáció az objektum-összetétel egy fajtája, nem kizárolagos tulajdonlást jelent. UML jelölése: Rombuszból induló nyíl, a rombusz üres.

**Objektum-összetétel - Kompozíció:** A kompozíció az objektum-összetétel egy fajtája, kizárolagos tulajdonlást jelent. UML jelölése: Rombuszból induló nyíl, a rombusz teli fekete.

**Objektum-összetétel - Becsomagolás:** A becsomagolás (angolul: wrapping) az objektum-összetétel szinonimája, amelyet akkor használunk, ha az azt akarjuk kihangsúlyozni, hogy a becsomagolt objektum az összetétel által minőségileg jobb lett. Más megfogalmazásban, akkor, ha a fő funkcionálitást a becsomagolt objektum adja. Két fajtája az átlátszó és a nem átlátszó becsomagolás. A becsomagolt objektum általában a külvilágóból érkezik felelősségi beinjektálásával. Mivel a fő funkcionálitást a becsomagolt objektum adja, ezért a csomagoló objektum a felelősségeinek egy részét átadja a becsomagolt objektumnak.

**Objektum-összetétel - Becsomagolás - Átlátszatlan becsomagolás:** A becsomagolás átlátszatlan, ha a becsomagolt objektum szolgáltatásai, azaz publikus metódusai, nem elérhetőek a csomagoláson keresztül. Tehát a nem átlátszó becsomagolás feltétele, hogy csak HAS-A kapcsolat legyen a becsomagoló és a becsomagolt objektum között. Lásd az Illesztő tervezési mintát.

**Objektum-összetétel - Becsomagolás - Atlátszó becsomagolás:** A becsomagolás átlátszó, ha a becsomagolt objektum minden szolgáltatása, azaz publikus metódusai, elérhetőek a csomagoláson keresztül is. Ennek feltétele, hogy a csomagolásnak ugyanolyan felülete legyen, mint a becsomagolt objektumnak. Tehát az átlátszó becsomagolás feltétele, hogy HAS-A és IS-A kapcsolat is legyen a becsomagoló és a becsomagolt objektum között. Lásd a Díszítő és a Helyettes tervezési mintát.

**OOP alapelv - Egységbázárás:** Az egységbázárás (angolul: encapsulation) azt jelenti, hogy a belső állapotot leíró mezőket és a belső állapot átmenetet megvalósító metódusokat egy egységbe zártuk, amit osztálynak nevezünk, illetve az objektum belső állapotát meg kell védeni, azt csak a saját metódusai változtathatják meg. Ezt szokás információrejtésnek (angolul: information hiding) is nevezni.

**OOP alapelv - Öröklődés:** Az öröklődés (angolul: inheritance) a kód-újrahasznosítás kényelmes formája. A gyermek osztály az ős osztály minden nem privát mezőjét és metódusát megörökli. Azaz a gyermek osztály örökli az ős osztály felületét és megvalósítását. Az öröklődés a gyermek és az ős osztály között implementációs függőséget okoz, ami kerülendő. Öröklődés helyett, ha csak lehet, objektum-összetételel ajánlott használni.

**OOP alapelv – Többalakúság:** A többalakúság (angolul: polymorphism) azt jelenti, hogy egy objektum több osztály példányaként is használható. A többalakúság az öröklődés következménye. Mivel a gyermek osztály örökli az ős felületét, ezért a gyermek osztály példányai megkapják az ős típusát is. Így egy objektum több típusként, azaz több alakban is használható.

**Osztály:** Az objektumorientált programozás alapegysége az osztály. Az osztálynak van felülete és megvalósítása. Az osztály publikus rész a felülethez tartozik, így pl. a publikus metódusok feje. A nem publikus részek a megvalósításhoz tartoznak, pl. a publikus metódusok törzse is a megvalósítás része.

**Osztály - Absztrakt osztály:** Olyan osztály, amelynek van interfésze, de implementációja lehet hiányos, azaz lehetnek absztrakt metódusai, amelyek implementálását a gyermek osztályaira bízza.

**Osztály - Metódus:** Az osztályhoz implementációját megvalósító alprogramokat metódusoknak nevezünk.

**Osztály - Mező:** A mező (angolul: field) az osztály belső állapotának tárolását valósítja meg.

**Refaktorálás:** A kódszépítés, vagy refaktorálás (angolul: refactoring) az a programozói módszer, amikor egy forráskód viselkedését nem változtatjuk meg, csak az implementációját írjuk át, hogy könnyebben értető legyen. Az integrált fejlesztői környezetek sokfajta kódszépítési módszert, pl. átnevezés, metódus kiemelés, támogatnak.

**Réteg:** A program olyan jól elszeparált részét értjük, amely akár külön számítógépen futhat.

**Szerződés alapú programozás:** A szerződés alapú programozás (angolul: Design by Contract) egy programozási technika, amely során azt feltételezzük, hogy a meghívott metódus és a hívó metódus közt van egy szerződés, amely kimondja, hogy ha a hívó betartja a hívott előfeltételét, akkor a hívott garantálja, hogy az utófeltétele igaz lesz. Ebben a megközelítésben minden metódusnak van elő- és utófeltétele, esetleg invariánsa, amelyeket együtt szerződésnek nevezünk.

**TDD - Teszt vezérelt programozás:** A teszt vezérelt programozás (angolul: Test Driven Development, röviden: TDD) egy programozási technika, amely során először írjuk meg a metódushoz tartozó tesztet, csak utána magát a metódust.

**TDD - Piros-Zöld-Kék-Piros:** Amennyiben a kód refaktorálásra szorul (a kódban redundáns sorok vannak, vagy lehetne alkalmazni egy tervezési mintát, esetleg szét lehetne választani valamit, amire eddig nem volt szükség) a kék fázis következik. Tulajdonképpen ez a kódszépítési fázis. Elvileg minden zöld lépés után kék jön, ezért a TDD-t nevezhetjük Piros – Zöld – Kék – Piros tesztelésnek is.

**TDD - Piros-Zöld-Piros:** A sikertelen unit-teszt jelzése minden keretrendszerben piros, a sikeresé zöld. A TDD előírja, hogy a metódusból csak annyit írunk meg, hogy épp átmenjen a legutoljára megírt unit-teszten. Ezután írunk egy új unit-tesztet a már félkész metódushoz, ami egy eddig le ne fedett esetet tesztel, aztán írunk meg a metódusba ezt az esetet. És így tovább, amíg minden esetet lefedünk teszesettel, illetve kóddal a metódusban.

**TDD - Tesztvezérelt programozás lépései:** 1.) Unit-tesztet írunk először, csak utána metódust. 2.) minden unit-teszt az eddig nem tesztelt lehető legegyszerűbb esetet írja le. 3.) A metódusból csak annyit írunk meg, hogy épp átmenjen az összes unit-teszten.

**Tervezési alapelv:** A tervezési alapelv egy olyan programozók számára szóló ajánlás, amelyet követve a létrejövő programok rugalmasan bővíthetők és könnyebben újrafelhasználhatók.

Tervezési alapelv - A szétválasztás elve: A szétválasztás elve (angolul: separation of concerns) azt mondja ki, hogy amit szét lehet választani, azt érdemes is szétválasztani. Ez az elv az SRP tervezési alapelv még magasabb absztrakciós szinten lévő változata.

Tervezési alapelv - GOF1: A GOF1 egy tervezési alapelv, a GOF könyv 1. alapelve, amely így hangzik: Programozz felületre megvalósítás helyett.

Tervezési alapelv - GOF2: A GOF2 egy tervezési alapelv, a GOF könyv 2. alapelve, amely így hangzik: Használj objektum-összetételelő öröklődés helyett, hacsak lehet.

Tervezési alapelv - HP: A HP egy tervezési alapelv, az angol „Hollywood Principle” kifejezés rövidítése, amelyet magyarul Hollywood alapelvnek nevezünk, és amely így hangzik: Ne hívj, majd mi hívunk.

Tervezési alapelv - OCP: Az OCP egy tervezési alapelv, az angol „Open – Closed Principle” kifejezés rövidítése, amelyet magyarul nyitva–zárt alapelvnek nevezünk, és amely így hangzik: A program legyen nyitott a bővítésre, de zárt a módosításra.

Tervezési alapelv - SRP: Az SRP egy tervezési alapelv, az angol „Single Responsibility Principle” kifejezés rövidítése, amelyet magyarul egy felelősség – egy osztály alapelvnek nevezünk, és amely így hangzik: minden osztálynak csak egy oka legyen a változásra.

Tervezési minta: A tervezési minta egy gyakori programozási feladatra adott általános megoldás, amely széles körben elfogadott, szakemberek által ajánlott. A tervezési minták használatával rugalmasan bővíthető, könnyen újrahasznosítható programot kapunk.

Tervezési minta - Absztrakt gyár: Az Absztrakt gyár (angolul: Absztrakt Factory) egy létrehozási tervezési minta, amely olyan objektumok gyártására jó, amelyek képesek egymással együttműködni, ennek megfelelően több létrehozásra alkalmas metódust (angolul: create method) tartalmaz.

Tervezési minta - Díszítő: A Díszítő (angolul: Decorator) egy szerkezeti tervezési minta, amely úgy csomagol be, azaz díszít, egy objektumot, hogy az megőrzi eredeti viselkedését, illetve az bővül vagy egy új metódussal vagy valamelyik metódus viselkedése bővül. Ehhez átlátszó becsomagolást használ.

Tervezési minta - Egyke: Az Egyke (angolul: Singleton) egy létrehozási tervezési minta, amely bemutatja, hogyan kell olyan osztályt létrehozni, amiből legfeljebb csak egy példány hozható létre, és ez egy példány egy globális hozzáférési ponton keresztül érhető el.

Tervezési minta - Helyettes: A Helyettes (angolul: Proxy) egy szerkezeti tervezési minta, amely megadja, hogyan kell olyan objektumot készíteni, amely egy értékes objektum és a külvilág közé áll, a külvilág szemszögéből teljesen átlátszó, kiszolgálja a külvilágból érkező kéréseket. A Helyettes tervezési mintának sok változata van, amelyek abban különböznek, hogy a helyettes hogyan delegálja a felelősséget az értékes objektum felé. Ez a minta átlátszó becsomagolást használ.

Tervezési minta - Illesztő: Az Illesztő (angolul: Adapter) egy szerkezeti tervezési minta, amely átalakítja a becsomagolt objektum felületét a kívánt felületre. Ehhez nem átlátszó becsomagolást használ.

Tervezési minta - Létrehozási tervezési minták: A létrehozási tervezési minták olyan tervezési minták, amelyek objektumok gyártásának bevált módszereit mutatják be.

Tervezési minta - Megfigyelő - Húzó: A Húzó Megfigyelő (angolul: Observer with pull model) egy referenciát ad át a megfigyelőknek, amin keresztül lehúzhatják az eseményt.

Tervezési minta - Megfigyelő - Toló: A Toló Megfigyelő (angolul: Observer with push model) magát az eseményt adja át a megfigyelőknek paraméterként.

Tervezési minta - Megfigyelő: A Megfigyelő (angolul: Observer) egy viselkedési tervezési minta, amely egy esemény által kiváltott változékony metódust emel ki egy egy-sok kapcsolat sok oldalára, és amely műsorszórással hívja meg a kiemelt metódusokat. Ez a tervezési minta a HP tervezési alapelvet valósítja meg. A megfigyelőnek két fajtája van a húzó (angolul: pull) és a toló (angolul: push), amelyek abban különböznek, hogyan adjuk át az eseményt a megfigyelőknek.

Tervezési minta - Prototípus: A Prototípus (angolul: Prototype) egy létrehozási tervezési minta, amely egy prototípus klónozásával gyárt objektumokat. A klónok a prototípus pontos másolatai, de saját memória címük van, így a klón megváltoztatása nem változtatja meg a prototípust. Ez a tervezési minta mély klónozást (angolul: deep copy) használ.

Tervezési minta - Sablon metódus: A Sablon metódus (angolul: Template Method) egy viselkedési tervezési minta, amely egy vagy több változékony metódust emel ki egy gyermekosztályba, és amely IoC segítségével hívja meg a kiemelt metódusukat.

Tervezési minta - Stratégia - Változékony metódus: Változékony metódusról beszélünk, ha egy metódus kódját gyakran átírjuk a megváltozott megrendelői igények miatt.

Tervezési minta - Stratégia: A Stratégia (angolul: Strategy) egy viselkedési tervezési minta, amely minden változékony metódust emel ki egy osztály hierarchiába, és amely felelősséggel átadással hívja meg a kiemelt metódust.

Tervezési minta - Szerkezeti tervezési minták: A szerkezeti tervezési minták olyan tervezési minták, amelyek olyan bevált módszereket mutatnak be, amelyekkel objektumokból objektum szerkezeteket lehet létrehozni általában a becsomagolás módszerével.

Tervezési minta - Viselkedési tervezési minták: A viselkedési tervezési minták olyan tervezési minták, amelyek olyan bevált módszereket mutatnak be, amelyekkel egy osztályból kiemelhető egy másik osztályba egy vagy több változékony metódus. A szétválasztás elvének megfelelően a változékony metódusokat minden érdemes kiemelni másik objektumba.

Tiszta kód: A tiszta kód alatt olyan forráskódot értünk, amelyet más programozók is könnyen megértenek, könnyen bővítnek, és könnyen újrafelhasználnak.

Tiszta kód - A cserkész szabály: A cserkész szabály (angolul: The Boy Scout Rule) egy általános alapelvek, amely kimondja, hogy hagyd a tábot tisztábban, mint ahogy találtad. Programozási módszerként azt jelenti, ha valamik hozzájárult egy kódhoz, akkor azt tisztábban kell visszatennie a verziókötő rendszerbe, mint ahogy találta. Ennek elfogadott módszere a kódszépítés vagy más szóval refaktorálás (angolul: refactoring).

Tiszta kód - Olvasható kód: Az olvasható kód alatt olyan forráskódot értünk, amely olvasásához nem kell fel-le ugrálnunk a forráskód szövegében, hanem az fentről lefelé elolvasva megérthető. Az

olvasható kód rövid osztályokból áll, az osztály fő szolgáltatását adó metódusok vannak elől, az ezek által hívott metódusok ezek alatt. Az olvasható kód kerüli a megjegyzések használatát, hiszen ha kód csak megjegyzéssel együtt értelmezhető, akkor az nem könnyen olvasható.

Unit-teszt: Lásd egységeszt.