FEATURE FREEZ - SENKI NE SZERKESSZE!!!!!!!

Eszterházy Károly Katolikus Egyetem

Informatikai Kar

Kusper Gábor, Fazekas Csaba

Programozási Technológiák

megújított, részben lerövidített, részben kibővített

egyetemi jegyzet

Mottó

Weöres Sándor: Ének a határtalanról

Amikor még senkise voltam, / fény, tiszta fény, / a kígyózó patakokban / gyakran aludtam én.

Hogy majdnem valaki lettem, / kő, durva kő, / hegylejtőn jég-erezetten / hömpölygetett nagy erő.

És végül élni derültem, / láng, pőre láng, / a szerte határtalan űrben / mutatom valódi hazánk.

verzió: 2.2.9.1.

2024-07-09

Tartalomjegyzék

[1. Bevezetés 7](#_heading=h.gjdgxs)

[1.1. Lecke 1.: A tervezői szakma 8](#_heading=h.30j0zll)

[1.1.1. A lecke célja és tartalma 8](#_heading=h.1fob9te)

[1.1.2. Bevezetés 8](#_heading=h.3znysh7)

[1.1.3. Üzleti elemző és az üzleti folyamat fogalma 9](#_heading=h.2et92p0)

[1.1.4. Rendszerszervezés 10](#_heading=h.tyjcwt)

[1.1.5. Rendszerterv, és backlog 11](#_heading=h.3dy6vkm)

[1.1.6. Rendszerelemző, és rendszerszervező 14](#_heading=h.1t3h5sf)

[1.1.7. Folyamatelemző, és folyamatszervező 15](#_heading=h.4d34og8)

[1.1.8. A szökőkút-dilemma 18](#_heading=h.er2xu9zdsuy9)

[1.1.9. Összegzés és ellenőrző kérdések 20](#_heading=h.2s8eyo1)

[1.2. Számítógépes információs rendszer fogalma 21](#_heading=h.17dp8vu)

[1.3. Szoftverkrízis 25](#_heading=h.3rdcrjn)

[1.4. A három P betűs szó: Principles, Patterns, Practices 28](#_heading=h.26in1rg)

[1.5. A jó terv viták eredménye 30](#_heading=h.lnxbz9)

[2. A használati eset alapú tervezés és termékei 31](#_heading=h.35nkun2)

[2.1. Információs rendszerek életciklusa 32](#_heading=h.1ksv4uv)

[2.2. Lecke 2.: Bevezetés a használati eset alapú tervezésbe 34](#_heading=h.2jxsxqh)

[2.2.1. A lecke célja és tartalma 34](#_heading=h.z337ya)

[2.2.2. Bevezetés 34](#_heading=h.3j2qqm3)

[2.2.3. Használati eset diagram 35](#_heading=h.1y810tw)

[2.2.4. Rendszerhatár 36](#_heading=h.9rq844b2doea)

[2.2.5. Szereplő és üzleti szereplő 39](#_heading=h.4i7ojhp)

[2.2.6. Szereplők a bal és a jobb oldalon 46](#_heading=h.2xcytpi)

[2.2.7. Nyilak az UML használati eset diagramon 47](#_heading=h.1ci93xb)

[2.2.8. Alábontás, és visszakövethetőség 53](#_heading=h.3whwml4)

[2.2.9. Felhasználói sztori 54](#_heading=h.2bn6wsx)

[2.2.10. Egy felhasználó több szerepkörben 58](#_heading=h.qsh70q)

[2.2.11. Összegzés és ellenőrző kérdések 58](#_heading=h.3as4poj)

[2.3. Lecke 3.: Specifikáció írás 59](#_heading=h.1pxezwc)

[2.3.1. A lecke célja és tartalma 59](#_heading=h.49x2ik5)

[2.3.2. Bevezetés 59](#_heading=h.2p2csry)

[2.3.3. Olvasmányos specifikáció készítése 59](#_heading=h.147n2zr)

[2.3.4. Három olvasmányos példa 60](#_heading=h.3o7alnk)

[2.3.5. Félreértések elkerülését szolgáló módszerek 61](#_heading=h.23ckvvd)

[2.3.6. A Markdown nyelv rövid bemutatása 62](#_heading=h.ihv636)

[2.3.7. Miért Markdown? Mert a kontextus váltás káros! 63](#_heading=h.2omtkrucyaec)

[2.3.8. Összegzés és ellenőrző kérdések 65](#_heading=h.32hioqz)

[2.4. Lecke 4.: Követelményspecifikáció 65](#_heading=h.1hmsyys)

[2.4.1. A lecke célja és tartalma 65](#_heading=h.41mghml)

[2.4.2. Bevezetés 66](#_heading=h.2grqrue)

[2.4.3. Szabad riport 66](#_heading=h.vx1227)

[2.4.4. Irányított riport 67](#_heading=h.3fwokq0)

[2.4.5. Funkcionális és nemfunkcionális követelmények 68](#_heading=h.1v1yuxt)

[2.4.6. Riportok elemzése 68](#_heading=h.4f1mdlm)

[2.4.7. Szereplők és használati esetek 68](#_heading=h.2u6wntf)

[2.4.8. Követelménylista 72](#_heading=h.19c6y18)

[2.4.9. Fogalomszótár 73](#_heading=h.3tbugp1)

[2.4.10. Összegzés és ellenőrző kérdések 74](#_heading=h.28h4qwu)

[2.5. Lecke 5.: Funkcionális specifikáció 75](#_heading=h.nmf14n)

[2.5.1. A lecke célja és tartalma 75](#_heading=h.37m2jsg)

[2.5.2. Bevezetés 75](#_heading=h.1mrcu09)

[2.5.3. Követelményelemzés 76](#_heading=h.46r0co2)

[2.5.4. Képernyőtervek 76](#_heading=h.2lwamvv)

[2.5.5. Felhasználói sztori, forgatókönyv 77](#_heading=h.111kx3o)

[2.5.6. Összegzés és ellenőrző kérdések 77](#_heading=h.3l18frh)

[2.6. Ütemterv 78](#_heading=h.206ipza)

[2.6.1. Napidíj 79](#_heading=h.4k668n3)

[2.6.2. COCOMO modell 79](#_heading=h.2zbgiuw)

[2.7. Árajánlat 80](#_heading=h.1egqt2p)

[2.7.1. Projektütemezés 81](#_heading=h.3ygebqi)

[2.7.2. Kritikus út 82](#_heading=h.sqyw64)

[2.8. Megvalósíthatósági tanulmány 83](#_heading=h.3cqmetx)

[2.9. Lecke 6.: Rendszerterv 84](#_heading=h.1rvwp1q)

[2.9.1. A lecke célja és tartalma 84](#_heading=h.4bvk7pj)

[2.9.2. A rendszerterv fajtái 85](#_heading=h.2r0uhxc)

[2.9.3. A rendszerterv fejezetei 87](#_heading=h.1664s55)

[2.9.4. Határosztály-tervezés 89](#_heading=h.3q5sasy)

[2.9.5. Menühierarchia-tervezés 89](#_heading=h.25b2l0r)

[2.9.6. Storyboard 89](#_heading=h.kgcv8k)

[2.9.7. Logikai rendszerterv 91](#_heading=h.34g0dwd)

[2.9.8. Fizikai rendszerterv 92](#_heading=h.1jlao46)

[2.9.9. Összegzés és ellenőrző kérdések 92](#_heading=h.43ky6rz)

[3. Principles: Az objektumorientált tervezés alapelvei 94](#_heading=h.2iq8gzs)

[3.1. Absztrakt alapelvek 94](#_heading=h.xvir7l)

[3.1.1. A program kódja állandóan változik 94](#_heading=h.3hv69ve)

[3.1.2. Ügyek szétválasztása – Separation of Concerns 94](#_heading=h.1x0gk37)

[3.1.3. A szoftver is csak egy termék, olyan, mint egy doboz müzli 95](#_heading=h.2w5ecyt)

[3.1.4. A szoftverfejlesztésnek nincs Szent Grálja 95](#_heading=h.1baon6m)

[3.1.5. A terv legyen a lehető legegyszerűbb, és csak annyira bonyolult, amennyire muszáj 95](#_heading=h.3vac5uf)

[3.1.6. További elvek 96](#_heading=h.2afmg28)

[3.2. Az objektumorientált programozás alapelvei 98](#_heading=h.pkwqa1)

[3.2.1. Egységbezárás – Encapsulation 100](#_heading=h.39kk8xu)

[3.2.2. Öröklődés – Inheritance 101](#_heading=h.1opuj5n)

[3.2.3. Többalakúság – Polymorphism 101](#_heading=h.48pi1tg)

[3.3. Az OOP hasznos megoldásai 102](#_heading=h.2nusc19)

[3.3.1. Automatikus szemétgyűjtés 102](#_heading=h.1302m92)

[3.3.2. A mező mint lokális-globális változó 111](#_heading=h.3mzq4wv)

[3.3.3. Többalakúság használata osztály behelyettesítésre 112](#_heading=h.2250f4o)

[3.3.4. Csatoltság csökkentése objektum-összetétellel 112](#_heading=h.haapch)

[3.4. Tervezési alapelvek 118](#_heading=h.319y80a)

[3.5. Lecke 7.: A GOF könyvei alapelvei 118](#_heading=h.1gf8i83)

[3.5.1. A lecke célja és tartalma 118](#_heading=h.40ew0vw)

[3.5.2. A GOF könyv 1. alapelve – GOF1 119](#_heading=h.2fk6b3p)

[3.5.3. A GOF könyv 2. alapelve – GOF2 123](#_heading=h.upglbi)

[3.5.4. Az IS-A és a HAS-A kapcsolat 123](#_heading=h.3ep43zb)

[3.5.5. Átlátszó és átlátszatlan újrahasznosítás 125](#_heading=h.1tuee74)

[3.5.6. Aggregáció és kompozíció 125](#_heading=h.4du1wux)

[3.5.7. Átlátszó és átlátszatlan becsomagolás 127](#_heading=h.2szc72q)

[3.5.8. Rész – egész viszony 130](#_heading=h.184mhaj)

[3.5.9. Összegző példa objektum-összetételre 131](#_heading=h.3s49zyc)

[3.5.10. Az objektum-összetétel és a csatoltság 132](#_heading=h.279ka65)

[3.5.11. GOF2 a gyakorlatban 132](#_heading=h.meukdy)

[3.5.12. Összegzés és ellenőrző kérdések 132](#_heading=h.36ei31r)

[3.6. Lecke 8.: SOLID tervezési alapelvek 133](#_heading=h.1ljsd9k)

[3.6.1. A lecke célja és tartalma 133](#_heading=h.45jfvxd)

[3.6.2. Bevezetés 133](#_heading=h.2koq656)

[3.6.3. SRP (Single Responsibility Principle) – Egy felelősség - egy osztály alapelve 134](#_heading=h.zu0gcz)

[3.6.4. OCP (Open-Closed Principle) – Nyitva-zárt alapelv 134](#_heading=h.3jtnz0s)

[3.6.5. LSP (Liskov Substitution Principle) – Liskov-féle behelyettesítési alapelv 136](#_heading=h.1yyy98l)

[3.6.6. Szerződésalapú programozás – Design by Contract 137](#_heading=h.4iylrwe)

[3.6.7. ISP (Interface Segregation Principle) – Interfészszegregációs-alapelv 139](#_heading=h.2y3w247)

[3.6.8. DIP (Dependency Inversion Principle) – Függőség megfordításának alapelve 140](#_heading=h.1d96cc0)

[3.6.9. Összegzés és ellenőrző kérdések 142](#_heading=h.3x8tuzt)

[3.7. További tervezési alapelvek 142](#_heading=h.2ce457m)

[3.7.1. Hollywood alapelv – HP (Hollywood Principle) 142](#_heading=h.rjefff)

[3.7.2. Demeter törvénye / a legkisebb tudás elve 143](#_heading=h.3bj1y38)

[4. Patterns: Tervezési minták 144](#_heading=h.4anzqyu)

[4.1. Architekturális minták 144](#_heading=h.2pta16n)

[4.1.1. MVC – Model-View-Controller 144](#_heading=h.14ykbeg)

[4.1.2. Többrétegű architektúra 146](#_heading=h.338fx5o)

[4.2. Lecke 9.: Létrehozási tervezési minták 148](#_heading=h.42ddq1a)

[4.2.1. A lecke célja és tartalma 148](#_heading=h.2hio093)

[4.2.2. Bevezetés 148](#_heading=h.wnyagw)

[4.2.3. Egyke – Singleton 149](#_heading=h.3gnlt4p)

[4.2.4. Prototípus – Prototype 1](#_heading=h.2uxtw84)

[4.2.5. Gyártómetódus – Factory Method 1](#_heading=h.1a346fx)

[4.2.6. Absztrakt gyár – Absztrakt Factory 1](#_heading=h.2981zbj)

[4.2.7. Összegzés és ellenőrző kérdések 1](#_heading=h.odc9jc)

[4.3. Lecke 10.: Szerkezeti tervezési minták 1](#_heading=h.38czs75)

[4.3.1. A lecke célja és tartalma 1](#_heading=h.1nia2ey)

[4.3.2. Bevezetés 1](#_heading=h.47hxl2r)

[4.3.3. Illesztő – Adapter 1](#_heading=h.2mn7vak)

[4.3.4. Díszítő – Decorator 1](#_heading=h.11si5id)

[4.3.5. Helyettes – Proxy 1](#_heading=h.4kx3h1s)

[4.3.6. Összegzés és ellenőrző kérdések 1](#_heading=h.3z7bk57)

[4.4. Lecke 11.: Viselkedési tervezési minták 1](#_heading=h.2eclud0)

[4.4.1. A lecke célja és tartalma 1](#_heading=h.thw4kt)

[4.4.2. Bevezetés 1](#_heading=h.3dhjn8m)

[4.4.3. Állapot – State 1](#_heading=h.1smtxgf)

[4.4.4. Megfigyelő – Observer 1](#_heading=h.2rrrqc1)

[4.4.5. Sablonmetódus – Template Method 1](#_heading=h.3qwpj7n)

[4.4.6. Stratégia - Strategy 1](#_heading=h.ss8xw0w39c0t)

[4.4.7. Látogató – Visitor 1](#_heading=h.356xmb2)

[4.4.8. Dupla hivatkozásfeloldás – Double dispatch 1](#_heading=)

[4.4.9. Összegzés és ellenőrző kérdések 1](#_heading=h.44bvf6o)

[5. Practices: Jól bevált módszerek 1](#_heading=h.2jh5peh)

[5.1. Tiszta kód 1](#_heading=h.3im3ia3)

[5.1.1. Cserkészszabály 1](#_heading=h.1xrdshw)

[5.1.2. Rothadó kód 1](#_heading=h.4hr1b5p)

[5.2. Lecke 12.: Egységteszt 1](#_heading=h.2wwbldi)

[5.2.1. A lecke célja és tartalma 1](#_heading=h.1c1lvlb)

[5.2.2. Bevezetés 1](#_heading=h.3w19e94)

[5.2.3. Egységteszt készítése JUnit5 és Maven használata esetén 1](#_heading=h.2b6jogx)

[5.2.4. Egységteszt készítése Visual Studio 2013 vagy újabb esetén 1](#_heading=h.qbtyoq)

[5.2.5. Egységteszt készítése Visual Studio 2008 és 2010 esetén 1](#_heading=h.3abhhcj)

[5.2.6. Összegzés és ellenőrző kérdések 1](#_heading=h.1pgrrkc)

[5.3. Lecke 13.: Tesztvezérelt fejlesztés – Test Driven Development – TDD 1](#_heading=h.49gfa85)

[5.3.1. A lecke célja és tartalma 1](#_heading=h.2olpkfy)

[5.3.2. Bevezetés 1](#_heading=h.13qzunr)

[5.3.3. TDD és a tiszta kód 1](#_heading=h.3nqndbk)

[5.3.4. Piros – Zöld – Piros 1](#_heading=h.22vxnjd)

[5.3.5. TDD esettanulmány 1](#_heading=h.i17xr6)

[5.3.6. Piros – Zöld – Kék – Piros 1](#_heading=h.320vgez)

[5.3.7. A TDD a szoftverfejlesztés kettős könyvelése 1](#_heading=h.1h65qms)

[5.3.8. Összegzés és ellenőrző kérdések 1](#_heading=h.415t9al)

[5.4. Lecke 14.: Viselkedésvezérelt Fejlesztés – Behaviour-Driven Development – BDD 1](#_heading=h.srn77aa5qiuk)

[5.4.1. A lecke célja és tartalma 1](#_heading=h.1ps7zzu8renu)

[5.4.2. Bevezetés 1](#_heading=h.ft62wbye1yps)

[5.4.3. Mi is az a BDD? 1](#_heading=h.lk0dghhzq6un)

[5.4.4. A Gherkin nyelv bemutatása 1](#_heading=h.8hq663dr3olh)

[5.4.5. A Cucumber bemutatása 1](#_heading=h.a4smg5qtkuar)

[5.4.6. Összegzés és ellenőrző kérdések 1](#_heading=h.s38mfx83upif)

[5.5. Naplózás 1](#_heading=h.3sv78d1)

[5.6. Aspektusorientált programozás 1](#_heading=h.280hiku)

[5.7. Statikus tesztelési technikák 1](#_heading=h.n5rssn)

[5.7.1. Felülvizsgálat – Bevezetés 1](#_heading=h.375fbgg)

[5.7.2. Felülvizsgálat – Informális felülvizsgálat 1](#_heading=h.1maplo9)

[5.7.3. Felülvizsgálat – Átvizsgálás 1](#_heading=h.46ad4c2)

[5.7.4. Felülvizsgálat – Technikai felülvizsgálat 1](#_heading=h.2lfnejv)

[5.7.5. Felülvizsgálat – Inspekció 1](#_heading=h.10kxoro)

[5.7.6. Statikus elemzés - Bevezetés 1](#_heading=h.3kkl7fh)

[5.7.7. Statikus elemzés csak a forráskód alapján 1](#_heading=h.1zpvhna)

[**Ábrajegyzék 1**](#_heading=h.5w16coj4mj3a)

# Bevezetés

A jegyzet célja, hogy egy nagyon friss kihívásra reagáljon: A legújabb mesterséges intelligencia megoldások már egész jól programoznak!

Eddig az informatikai alapszakokon (programtervező informatikus, mérnökinformatikus, és gazdaságinformatikus) a képzés fő célja az volt, hogy a hallgatók megtanuljanak programozni. Ugyanakkor a nagy nyelvi modelleken (angolul: Large Language Model - LLM) alapuló mesterséges intelligenciák (magyar rövidítése: MI, angolul: artificial intelligence, angol rövidítése: AI) már egész jól programoznak. Az LLM-alapú mesterséges intelligenciákat szokás mesterséges általános intelligenciáknak (angolul: Artificial General Intelligence, angol rövidítése: AGI) is hívni, amikkel bármiről lehet beszélgetni, programozásról is. A jegyzet írásának idejekor, azaz 2024 év elején, ezek voltak a legismertebb AGI megoldások:

* OpenAI ChatGPT (előfizetés nélkül GPT-3.5, előfizetés esetén GPT-4.0 modellt használja)
* Google PaLM és Gemini
* ANTHROP\C Claude 3
* Meta LLaMA

Ráadásul már van olyan szűk MI, úgynevezett mesterséges szűk intelligencia (angolul: Artificial Narrow Intelligence, angol rövidítése: ANI), ami kifejezetten a programozók munkáját hivatott segíteni:

* Microsoft Copilot

Ez azt jelenti, hogy a mesterséges intelligencia átveheti a junior programozók munkáját. Ha ez tényleg bekövetkezik, akkor az informatika oktatásnak erre reagálni kell. Lássuk a lehetséges reakciókat:

1. Képezzünk célirányosan kérdező (angolul: prompt engineering) szakembereket! Bízhatunk abban, hogy az MI nem veszi át teljesen a junior programozók munkáját, csak segíti őket, és így az oktatás során nem csak a programozási környezetek használatát, pl. a nyomkövetést kell megtanítani, hanem azt is, hogyan kell az MI-től célirányosan kérdezni, hogy segítsen kijavítani egy hibát, megírni egy metódust.
2. Képezzünk medior programozókat! Ha junior programozókra már nincs szükség, akkor rögtön medior programozókat kell a programozó képzésben képezni, azaz lehetővé kell tenni, hogy a hallgatók specializálódjanak a piacon ismert irányokba, ezeken a területeken projekt munkában vegyenek részt és pár éves munkatapasztalatnak megfelelő tudást szerezzenek. A piacon jól ismert specializált programozói szakmák a következők:
   1. frontend programozó (magyarul: webprogramozó)
   2. backend programozó (magyarul: szerver oldali programozó)
   3. full stack (backend + frontend) programozó
   4. adattudós (angolul: data scientist)
   5. gépi tanulás szakértő (angolul: machine learning expert)
3. Képezzünk junior tervezőket! Ha junior programozókra már nincs szükség, mert lesznek olyan MI-k, amik egy tervből képesek jó minőségű kódot írni, amit néhány medior, egy-két senior programozó kicsiszol, akkor megnő az igény a tervezői munkakörökre. Ebben az esetben a cégek junior programozók helyett junior tervezőket fognak keresni. Ez a tendencia már most is megfigyelhető, az egyik legismertebb hazai banki informatikai beszállító szoftverház már most több junior tervezőt keres, mint junior programozót. Ez azt jelenti, hogy a programozó képzés keretében elmélyült tervezési ismereteket is nyújtani kell a felsőoktatásban.
4. Képezzünk kommunikatív, csapatmunkára alkalmas szakembereket! Ha junior programozókra már nincs szükség, mert az MI megfelelően jó minőségű kódot ír, még akkor is szükség lesz csapatra, ami képes együttműködni a MI-vel, illetve a tagok egymással. Ennek megfelelően a programozó képzésben nagyobb hangsúlyt kell kapnia az úgynevezett soft skill-ek (magyarul: csapatmunka készségek) fejlesztésének.
5. Bizonyára lesznek más válaszok is, amik néhány év múlva visszatekintve egyértelműek lesznek, de most nehezen láthatók.

Ennek a jegyzetnek a célja, hogy a 3-as számú választ tegye lehetővé: Képezzünk junior tervezőket! Eddig nem is létezett olyan szakma, hogy junior tervező. A munkaerő piacon elfogadott szemlélet szerint csak az lehetett tervező, akinek már néhány év tapasztalata volt programozóként, esetleg tesztelőként, hogy könnyen megértse a szót a keze alá dolgozó programozókkal. Emiatt lényegében csak medior tervezőként lehetett kezdeni a tervezői szakmát. Ennek a jegyzetnek a célja, hogy a kedves olvasó olyan tudásra tegyen szert, aminek a segítségével sikeresen elhelyezkedhet junior tervezőként.

A jegyzet használható referenciakönyvnek, amiben csak megnézzük azt a tervezési fogalmat, amire kíváncsiak vagyunk, illetve, használható egymásra épülő leckék sorozataként is. Az ide tartozó fejezeteket a lecke szóval jelöltük meg, a többi fejezet inkább referenciaként használandó.

## Lecke 1.: A tervezői szakma

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a tervezői szakmákkal.

Szükséges eszközök, források: -

Feldolgozási idő: 30 perc.

Témakörök: üzleti folyamat, üzleti elemző, rendszerelemző, folyamatelemző.

A lecke tartalma: Ebben a leckében azt a 3 tervezői szakmát nézzük át, amik hazánkban a legelterjedtebbek. Ezek az üzleti elemző, a rendszerelemző, és a folyamatelemző. Mindhárom szakma részleteit feltárjuk, az átfedéseket megmutatjuk. Az üzleti elemző és rendszerszervező kiemelkedő szerepet játszanak az üzleti folyamatok hatékonyságának növelésében és a fejlesztési folyamat előzetes tervezésében. A folyamatelemzők pedig proaktívan hozzájárulnak a vállalati folyamatok javításához és automatizálásához. Az első két szakma képviselői általában szoftverfejlesztő cégeknek dolgoznak, a harmadik szakma képviselői pedig a megrendelői oldalon.

### Bevezetés

A számítógépes információs rendszereket tervező szakma több speciális szakmát takar:

* üzleti elemző (angolul: business analyst),
* rendszerelemző, rendszerszervező (angolul: system analyst, system architect),
* folyamatelemző, folyamatszervező (angolul: process analyst, process architect),
* követelményelemző (angolul: requirement engineer),
* adatelemző (angolul: data analyst),
* változáselemző (angolul: change management analyst).

A fenti lista nyilván nem teljes körű, inkább csak ízelítő, milyen sokféle a számítógépes információs rendszert tervező szakma. A fenti listából csak az első hármat fejtjük ki. A többi munkakör nagyon-nagyon ritkán jelenik meg hazai informatikai álláshirdetésekben, és ha mégis, biztosan nem junior szinten.

Mint látható, a tervezői szakmát néha az elemző (angolul: analyst), néha a szervező (angolul: architect) szóval is jelölik. Általában az architekt egy nagyon kiemelkedő, mély tudású szakembert jelöl. Talán már csak a guru szó közvetít nagyobb tiszteletet, de nem is szokás használni ezt a szót csak a rendszergazda guru, illetve a Linux guru szókapcsolatokban.

### Üzleti elemző és az üzleti folyamat fogalma

Az üzleti elemző (angolul: business analyst) feladata, hogy

* megértse a megrendelő üzleti folyamatait (angolul: business process),
* javaslatokat tegyen az üzleti folyamatok javítására, annak érdekében, hogy a megrendelő pénzt és időt takarítson meg, nőjön a hatékonysága;
* az elfogadott javaslatok alapján követelmény specifikációt készít,
* a követelmény specifikáció alapján funkcionális specifikációt készít.

Miután proaktív módon meggyőzi a megrendelőt a változások fontosságáról, vagy a megrendelőben magától igények merülnek fel, akkor ezeket az igényeket mint követelményeket megfogalmazza. A követelményeket vagy átadja a rendszerelemzőnek, vagy ő maga készít ezekből informatikai rendszertervet egy módszertan mentén, esetleg ad-hoc módon.

Hogy az üzleti folyamat (angolul: business process) alatt pontosan mit értünk, az iskolánként eltérő lehet. Mi ezt a definíciót használjuk: Az üzleti folyamat olyan

* gyakran ismétlődő tevékenység sorozat,
* amelynek bemenete
  + nyersanyag, félkész termékek,
  + adat, információ,
  + energia;
* amelyet egy cég
  + munkatársai, megbízottjai,
  + robotjai,
  + szoftverei együttműködve végeznek; ehhez
* felhasználják a cég eszközeit, amik lehetnek
  + pénzügyi eszközök (tőke, hitel, bér),
  + termelőeszközök, szerszámok, gépsorok,
  + számítógépek, szerverek, szenzorok, munkaállomások,
  + járművek, targoncák, szállító eszközök;
* és ez értéket teremt.

Az érték lehet:

* kézzelfogható árucikk, félkész termék, részegység, csomagolás,
* szolgáltatás,
* kézzel nem fogható döntés, terv, ellenőrzés,
* ezeket dokumentáló dokumentáció, illetve
* bármilyen más dolog, ami a cég valamely üzleti céljának elérését szolgálja.

Azt lehet mondani, hogy minden folyamat üzleti folyamat, ha azt egy cégen belül végezzük, nem egyszeri, és a cég érdekét szolgálja.

Ezek mind elég intuitív feltételek, valószínűleg bárki egy kis józan paraszti ésszel így definiálná az üzleti folyamatot. Egy nem teljesen természetes kikötést adtunk: Az üzleti folyamat egy olyan folyamat, ami gyakran ismétlődik. Azaz megismételhető. Ezt sok más definíció úgy fogalmazza meg, hogy strukturált. E helyett mi a gyakran ismétlődő megfogalmazást használjuk, mert ez az a pont, ahol az informatika hatékonyan tud egy üzleti folyamatot javítani.

Az üzleti elemző legfontosabb feladata, hogy megértse a megrendelő üzleti folyamatait. Ehhez úgynevezett domain tudásra van szüksége. Itt van egy rekurzív csapda, mert a domain tudásnak része az üzleti folyamatok ismerete, az üzleti folyamatok megismeréséhez pedig szükséges a domain tudás. Ezt a csapdát úgy lehet kivédeni, ha a domain tudás összegyűjtését egy tanulási folyamatnak fogjuk fel. Először csak a megrendelő alapvető szakszavait tanuljuk meg. Aztán megismerjük az alapvető struktúrákat, döntéshozatali folyamatokat. Majd szépen, fokozatosan feltárjuk a megrendelő üzleti folyamatait. Ha valamit nem értünk, fontos, hogy kérdezzünk, mert az üzleti elemző legnagyobb ellensége a félreértés. A legjobb barátja pedig a rendszerszemlélet.

A rendszerszemlélet annak a képessége, hogy egy „sztoriról” képesek vagyunk eldönteni, hogy kerek egész, vagy bizonyos részletek még nem világosak. Itt a sztori leggyakrabban egy üzleti folyamat leírása. Például strand belépőjegyet árusítunk: Van díjmentes gyerekjegy 0-tól 5 éves korig. Van kedvezményes diákjegy, amelyhez diákigazolvány kell. Van kedvezményes családi jegy 2 felnőtt + 2 gyermek részére. Van normál felnőttjegy. És van szintén kedvezményes nyugdíjas jegy.

Kerek ez a sztori? Hiányzik valami részlet? Ennyi információ alapján implementálható ez a funkció?

Aki észreveszi, hogy a kedvezmény sehol sincs pontosan megadva, az már jó úton halad. Az sem derül ki, hogy a normál felnőttjegy mennyibe kerül. Azt már egy kicsit nehezebb észrevenni, hogy a gyerek fogalma nem olvasható ki a fenti leírásból. Ki a gyerek? Akinek diákigazolványa van? Aki a 18. életévét még nem töltötte be? Ez nem derül ki, erre rá kell kérdezni.

A félreértések nagyon gyakran abból adódnak, hogy ami a megrendelőnek egyértelmű, arra a megrendelő nem veszteget szót, hiszen egyértelmű. A jó üzleti elemzőnek fontos képessége, hogy ráérezzen, mit hallgatott el a megrendelő. A másik fontos tulajdonsága, hogy proaktív módon ajánlja a fejlesztéseket.

Mivel az üzleti folyamatok gyakran ismétlődnek, ezért általában érdemes őket automatizálni. Az viszonylag ritka, hogy egy informatikai megoldás közvetlenül hasznot termel a megrendelőnek. Persze van ilyen is, például egy webshop közvetlenül hasznot termel. Sokkal gyakoribb, hogy egy informatikai megoldás időt, más erőforrást spórol meg, a szokottnál gyorsabb és pontosabb választ ad, megmutatja a válasz mögötti összefüggéseket, és így közvetett hasznot termel. Ez úgy érhető el, ha sikerül egy-egy üzleti folyamatot automatizálni.

### Rendszerszervezés

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 (lásd: Szoftverkrízis). A szoftverkrízisre adott egyik válasz a rendszerszervezés.

A rendszerszervezés egy körfolyamatot támogat, a szoftverfejlesztés életciklusát (angolul: Software Development Life Cycle, röviden: SDLC). Ez alatt sok termék (szoftver, dokumentáció) keletkezik, amelyek elkészítésére a módszertanok tesznek ajánlást. A legfontosabb dokumentum a rendszerterv, ami vagy egy nagy összefüggő doksi, vagy rengeteg kisebb dokumentum alkotja, amelyek a backlog részei. Az SDLC fogalmát részletesen áttekintjük egy későbbi fejezetben.

Az SDLC első része a követelmények elemzése és a terv elkészítése, amit így együtt angolul "(system) analysis and design”-nak nevezünk és A&D rövidítéssel hivatkozunk rá. Később ezt a rövidítést gyakran használjuk.

A korai módszertanok nagy hangsúlyt fektettek arra, hogyan kell részletes, előretekintő 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.

### Rendszerterv, és backlog

A rendszerterv (angolul: Design Specifications, esetleg: System Design Specifications) fogalmát részletesen áttekintjük egy későbbi fejezetben. Itt most csak a fogalom bevezetésére kerül sor. A rendszerterv arra szolgál, hogy a tervező alaposan átgondolja, mit és hogyan kell elkészíteni. Azt gondolhatnánk, hogy ez teljesen felesleges, mert a programozók úgyis végigmennek ezeken a lépéseken. A követelmény specifikáció alapján átgondolják, mit és hogyan kell implementálni, elkezdik megvalósítani, és ha kiderül, hogy valamit úgy nem lehet megoldani, ahogy elsőre tervezték, akkor újra átgondolják, újból nekifutnak.

Ez egy kis projektnél, úgynevezett garázs projektnél működik is. Ugyanakkor van két fogalom, amit ismernünk kell, ez a programozás-kicsiben (angolul: programming in small) és a programozás-nagyban (angolul: programming in large). Ez utóbbi kicsit hasonlít a WALL-E mozifilmben a Buy n Large (magyarul: Vásárolj Nagyban) cég nevére.

A programozás-kicsiben fogalmat használjuk a garázs projektek megnevezésére: Pár haver saját szórakozására programot ír. Jobb, mint ha hangosan zenélnének, legalább csendben vannak.

Ezzel szemben a programozás-nagyban azt jelenti, hogy akkora kódot írunk, amit egy-két ember, akármilyen zsenik, már nem látnak át. Ekkora kód írásánál már csapatmunka szükséges, ezt a munkát már szervezni kell, a fejlesztéshez pénzt paripát fegyvert kell biztosítani.

Az első esetben megengedhetjük magunknak, hogy nem tervezünk, legfeljebb vakvágányra futunk, vagy csak demó szinting jutunk. A második esetben luxus kihagyni a tervezést. A tapasztalat azt mutatja, hogy sokkal jobb többször alaposan átgondolni egy feladatot, teleírni néhány lapot példákkal, mielőtt egy sor kódot is írnánk.

**100 Forintos Licit:** A fő gond azzal van, hogy ha valaki valamibe energiát fektet, akkor azt már nem akarja veszni hagyni, vagy csak nehezen. Erre épül a 100 Forintos Licit játék: Bocsássunk licitálásra egy 100 forintos érmét! Egy kicsit keltsük fel a közönség érdeklődését, ez egy kivételes darab, a monda szerint több híresség birtokában is volt, és most bárki megszerezheti, aki a legtöbbet ajánlja érte. Azonban van egy csavar: nemcsak a legnagyobb licit fizet, hanem a második legnagyobb is, ugyanakkor ő nem kap semmit. Próbálják ki! A tapasztalat azt mutatja, hogy a második nem akarja veszni hagyni a befektetését, akár irracionálisan nagy árat is hajlandó fizetni egy egyszerű 100 forintos érméért; de amikor rálicitál az eddigi legnagyobb licitre, akkor az lesz ugyanebben a helyzetben, ő se akarja veszni hagyni a pénzét, ezért ő is rálicitál. Lám, milyen gyarló az ember!

Ugyanez történik, ha terv nélkül vágunk bele a fejlesztésbe. Lesz egy félig-meddig jó megoldásunk, amit refaktorálással toldozunk-foldozunk, de sose lesz igazán jó. Azaz, lehetne, csak el kellene dobni az eddigi kódot, és elölről kezdeni, de ehhez túl gyarlók vagyunk. Ha pedig már annyira rothad a kód, hogy már tényleg csak ez az opció marad, az eldobás és újrakezdés, akkor valamelyik versenytárs úgyis leköröz minket, és tönkre megy a cég. Azt szokták mondani, hogy arra sosincs idő, hogy valamit rendesen megtervezzünk, de arra mindig akad, hogy toldozzuk-foldozzuk éveken át.

Tehát, programozás-nagyban esetén tervezni kell: Gondolja át a tervező még a fehér lapon mit és hogyan fejlesszünk. Ettől függetlenül biztosan lesz olyan, hogy kiderül, a terv nem jó, de a módszertanok kezelik ezt a helyzetet is.

És ezzel elértünk a módszertanok gondolatához. Ahogy a Buy n Large nem egy pozitív szereplő a WALL-E című filmben, úgy a programozás-nagyban sem feltétlenül pozitív. Ez a fogalom azt jelenti, hogy a programozás már nem programozó zsenik művészete, hanem egyfajta jól megfizetett mérnöki munka. Nem egyéni, hanem csapat munka, ahol a cél már nem a kivételes minőség, hanem az átlagosan jó minőség. Ehhez kell egy módszertan.

Ahhoz, hogy az átlagosan jó minőséget elérjük egy átlagos képességű csapattal, ahhoz szükségünk van egy szoftver főzési receptes könyvre. Ezt a receptes könyvet hívjuk szoftverfejlesztési módszertannak.

Persze felmerül a kérdés, hogy miért átlagos képességű csapattal dolgozunk? Miért nem válogatjuk össze a zseniket? Miért nem egy kimagaslóan jó képességű csapattal dolgozunk? Nos, ezt kipróbálták néhány amerikai top egyetemen. Az volt a tapasztalat, hogy a belharcok felőrölték a csapatot, nem tudott kialakulni a csapaton belüli hierarchia, mindenki főnök akart lenni, egymás fúrására ment el az energiájuk. Ebből az következik, hogy a programozás-nagyban szükségszerűen magával vonja a csapatban fejlesztést, ami szükségszerűen magával vonja, hogy átlagos képességű csapatunk legyen. Ezért kell egy módszertan, hogy a csapat munkája ne egy katyvasz, hanem egy átlagosan jó minőségű szoftver legyen.

A szoftverfejlesztési módszertanok arra a kérdésre adnak választ, hogyan bontsuk fel a bonyolult szoftverfejlesztést több egyszerűbb lépésre, hogy ennek eredményeként egy átlagosan jó minőségű szoftvert kapjunk.

Minden módszertan elfogadja azt az alapgondolatot, hogy fejlesztés előtt érdemes átgondolni, mit és hogyan fejlesztünk. A nehézsúlyú módszertanok előírják a rendszerterv készítését, míg a könnyűsúlyú módszertanok egy folyamatosan finomított backlog használatát javasolják.

A kettő ugyanazt szolgálja, a tervező átgondolja a feladatot, megoldási javaslatot ad, még mielőtt a csapat ráveti magát a feladatra. A különbség ott van, hogy a rendszerterv egy nagy dokumentum, a backlog először egy apró, pár oldalas dokumentum, ami folyamatosan bővül. A backlog sose áll össze egy nagy dokumentummá, hanem megmarad sok kis dokumentum strukturált, vagy kevésbé strukturált listájának.

A backlog szónak nincs elfogadott magyar fordítása. Azért nehéz lefordítani, mert elsődleges jelentése, hogy milyen feladatok vannak még hátra, mit kell még megtervezni, lefejleszteni. Másrészt jelenti az elkészült dokumentumok összességét is. Harmadrészt egyszerre utalhat még el nem készült, és már elkészült dokumentumokra is. Negyedrészt a backlog egy prioritásos lista, a lista tetején lévő feladatokat biztosan megcsináljuk, a lista alján lévőket aligha. Ötödrészt időben változik, bővül, finomodik. Azaz, mindenféle feladatot tartalmaz, illetve ezek alapján létrejött termékeket (terv, ábra, dokumentum), amelyek egy másik feladat bemenetei, egészen addig, amíg a feladat végeredménye forráskód nem lesz; a forráskód már nem része a backlognak. Lehetséges magyar fordítások:

* prioritásos teendőlista,
* háttérkészlet,
* feladattár.

Egy fórum beszélgetésben még ezek merültek fel, kiszűrve a komolytalan hozzászólásokat:

* hátraléklista, így egybeírva, vagy különírva: hátralék lista,
* várólista,
* hátralévő feladatok,
* feladatok,
* teendők.

Ebben a jegyzetben a backlog szót úgy használjuk, mint egy az angolból magyarba meghonosodott szakszót. Ha mégis magyarul kellene mondani, akkor a feladattár mellé tesszük le a voksunkat.

A backlog először csak egy pár oldalas vízió. Aztán a vízió alapján nagyon magas szintű eposzokat (angolul: epic) fogalmazunk meg. Majd ezeket folyamatosan alábontjuk sztorik sorozatává. A sztorikat felbontjuk feladatokra. Mivel az epik szó kezd meghonosodni nyelvünkben, ezért az eposz helyett inkább az epik szakszót használjuk ebben a jegyzetben.

Az epik egy összefüggő funkció csoport, alrendszer, vagy modul, ami egy-egy szóba, szókapcsolatba sűríthető. Epik lehet mondjuk a beléptető oldal, az eladási alrendszer, a gazda modul. A víziót nem szoktuk feladat szintnek tekinteni, így az epik a legfelsőbb hierarchia szint a feladatok listájában.

A második szint a felhasználói sztorik (angolul: user story) szintje. Felhasználói sztori helyett mondhatjuk röviden, hogy sztori. Egy epik több sztorit tartalmazhat. A felhasználói sztori egy funkció egy lefutása a felhasználó szemszögéből. Általában forgatókönyvszerű (angolul: scenario) leírás: Béla bácsi elindítja a rendszert, sikeres belépés után szeretné használni a gazda modult, ehhez megnyomja a Gazda Modul feliratú gombot a főképernyőn. Ezen belül szeretné megnézni a csapadék adatokat, de nincs megfelelő szolgáltatás előfizetve. A rendszer felajánl 4-5 lehetséges előfizetést, amik közül Béla bácsi választ. Ezután már megtekintheti a csapadék adatokat.

A harmadik szint a feladatok (angolul: task) szintje. Feladat helyett használhatjuk a taszk szót is. Mivel a feladat szó egyszerre jelölhet egy nagy átfogó epiket és egy kis fejlesztési feladatot, ezért az utóbbi értelemben inkább a taszk szót használjuk. Egy sztori több taszkot tartalmazhat. Egy sztorihoz általában tartozik egy-egy frontend-tervezési, programozási és tesztelési taszk. Ugyanígy egy-egy backend-tervezési, programozási és tesztelési taszk.

Ha egy sztori vagy egy taszk túl nagy, túl komplex, akkor finomítjuk, több kisebb sztorira, illetve taszkra bontjuk. Ezt a folyamatot az egyes módszertanok pontosan szabályozzák.

Tehát a backlog egy folyamatosan bővülő feladattár, ahol a bővülés módja:

* a finomítás, és
* az alábontás, illetve
* egy feladat eredménye lehet egy vagy több új feladat.

Azaz egy feladat feladatokat szül, egész addig, amíg forráskódot, illetve prototípust nem kapunk.

A backlog bővítésének fő folyamata:

* A rendszer vízióját finomítjuk és epikeket hozunk létre.
* Az epik alábontása a felhasználói sztorik listája.
* A sztori alábontása a taszkok listája.

A taszkok eredménye lehet:

* ábra,
* terv,
* módosítási kérelem (angolul: change request, angol rövidítése: CR),
* tesztjelentés,
* bármilyen dokumentum,
* de akár forráskód is.

Amíg ezek újabb feladatok bemenetei, addig a backlog bővítésére szolgálnak, de a kész forráskód, illetve a prototípus már nem része a backlognak.

A backlog egy-egy része lehet teljesen kifejtett. Más részei további finomításra, alábontásra váró részek. Egy szépen alábontott backlog egyszerre tartalmazza:

* a követelmény specifikációt,
* a funkcionális specifikációt,
* az ütemtervet, és
* a rendszertervet is.

Előnye, hogy folyamatosan bővül, így minden bővítés előtt lehet kommunikálni a megrendelővel:

* megmutatni, hol tart a fejlesztés,
* megmutatni az utolsó prototípust,
* megkérdezni, hogy jó irányba halad-e a terv, illetve a fejlesztés.

Akár rendszertervet készítünk, akár backlogot, ezek elkészítése, illetve gondozása a tervezők feladata.

### Rendszerelemző, és rendszerszervező

A rendszerelemző (angolul: system analyst) és az üzleti elemző (angolul: business analyst) munkakör ritkán válik szét, legalábbis az itthoni informatikai piacon. Gyakran az üzleti elemző feladata a rendszerterv elkészítése is, ami egyébként már inkább a rendszerelemző feladata lenne. Illetve gyakran a rendszerelemző feladata az üzleti igények megértése is, ami viszont egyértelműen az üzleti elemző feladata lenne. Az agilis módszertanok elterjedésével kezd szétválni a két szakma. Azok a tervezők, akik jobban tudnak kommunikálni a megrendelővel, ők az üzleti elemzők és gyakran a termék tulajdonosok is (angolul: product owner). Míg azok a tervezők, akik inkább a programozókkal, tesztelőkkel kommunikálnak sikeresen, de nem szeretnek a megrendelővel kommunikálni, azok a rendszerelemzők. Egy üzleti elemző, ha a termék tulajdonos is, akkor nem része az agilis fejlesztő csapatnak, hanem inkább 2-3 csapat tartozik alá. Egy rendszerelemző általában csak egy csapathoz tartozik.

A rendszerelemző feladatköre

* a funkcionális specifikáció alapján a logikai és a fizikai rendszerterv elkészítése,
* ha a módszertan rendszerterv helyett más dokumentum elkészítést írja elő, pl. backlog, akkor annak elkészítése, finomítása.

Mint látható, a két szakma pontos szétválasztása nehéz, inkább csak hozzávetőleges szabályok adhatók:

* Az üzleti elemző inkább a megrendelővel tartja a kapcsolatot.
* A rendszerelemző inkább a fejlesztő csapattal tartja a kapcsolatot.
* Az üzleti elemző inkább az A&D (angolul: Analysis and Design) feladatok első felében aktív:
  + elkészíti a követelmény specifikációt,
  + elkészíti a funkcionális specifikációt,
  + új epikeket vesz fel a backlogra,
  + új felhasználói sztorikat vesz fel a backlogra.
* A rendszerelemző inkább az A&D feladatok második felében aktív:
  + elkészíti a rendszertervet,
  + a backlogon lévő terveket finomítja,
  + a felhasználói sztorikat finomítja, taszkokat vesz fel a backlogra.

Míg üzleti elemző esetében nem szokás arhitekt szintről beszélni, rendszerelemző (angolul: system analyst) munkakörben általában létezik még egy fokozat, ez a rendszerszervező (angolul: system architect). Ez a szó, rendszerszervező, már kezd kikopni a szaknyelvből, habár még a 90-es évek elején gyakran találkozhattunk vele az álláshirdetésekben. A rendszerszervező szó helyett egyre gyakrabban arhitektnek hívjuk azokat a magasan képzett, nagy tapasztalattal bíró tervezőket, akiknek a feladata az architekturális kérdések eldöntése.

### Folyamatelemző, és folyamatszervező

A következő tervezői szakma a folyamatelemző (angolul: business process analyst, vagy röviden: process analyst). Míg eddig azt írtuk, hogy egy tervező gyakran egymaga látja el az üzleti elemző és a rendszerelemző munkaköröket, addig itt épp fordított a helyzet, ritka, hogy egy folyamatelemző más tervezői feladatot is ellásson. Ennek az az oka, hogy a folyamatelemzők általában a megrendelői oldalon ülnek, míg az üzleti elemzők és a rendszerelemzők a fejlesztést végző szoftverházban dolgoznak.

Míg az üzleti elemzőnél azt írtuk, hogy feladata proaktív módon javaslatokat tenni az üzleti folyamatok javítására, addig ez a folyamatelemzőre hatványozottan igaz: A folyamatelemző fő feladata az üzleti folyamatok javítása, automatizálása, mindezt a lehető legnyomulósabb, proaktív módon.

Miért hangsúlyozzuk ki ennyire a proaktivitást? Mert minden változásnak vannak vesztesei. Ha valamely folyamatot javítunk, automatizálunk, akkor Pista bácsinak és Mari néninek kevesebb lesz a munkája, és ha nem képesek továbbképezni magukat a cégen belül, akkor előbb-utóbb kirúgják őket. Általában egy vezető látja ezt, de a vezetőknek fontos, hogy tisztelet övezze őket. Épp ezért, egy vezető, legyen a neve Zolika, ritkán adja ki utasításba, hogy na Ferikém, javítani kellene F123-as folyamaton, hogy éves szinten megspóroljunk néhány milliót. Inkább az a bevett megoldás, hogy Ferike, a folyamatelemző, meglátja a javítás lehetőségét, és addig nyomul Zolikánál, amíg nem sikerül a szükséges fejlesztést jóváhagyatni. Zolika örül, mert spóroltak néhány milliót és ezért nem őt utálják. Ferike is örül, mert igazolva van a magas fizetése, még talán év végi bónuszt is kap, és hogy mindenki szerint ő az oka, hogy nyugdíj előtt kivágták Mari nénit és Pista bácsit, na az nem zavarja.

Persze nem ennyire fekete és fehér a helyzet, de minden tervezőnek tisztában kell lennie azzal, hogy minden fejlesztés egyben változás is, aminek lesznek nyertesei és vesztesei.

Még egy fontos kérdés merül fel: Hogyan tudja a folyamatelemző meggyőzni a vezetőséget egy-egy folyamat javításáról? A válasz kulcsszava a ROI (angolul: Return on Investment, magyarul: megtérülés számítás). Azt szokták mondani, hogy minden fejlesztést érdemes megtenni, aminek a megtérülési ideje legfeljebb 3 év, és semmit se, ahol a megtérülés 5 év feletti. Nyilván a fenti két szélsőérték nagyban függ a cég helyzetétől, de durva szabálynak elfogadható.

Elvileg ROI-t számolni nagyon egyszerű: ROI = A befektetés várható hozamának összege a következő X évben / (a befektetett összeg + fenntartási költségek X évre). Részletesen:

* A befektetés hozama = A cég értékének növekedése + Bevétel növekedés + Kiadás csökkenés.
* A befektetett összeg = A fejlesztés költsége.
* Fenntartási költség 1 évre = A fejlesztés üzemeltetési költsége 1 évre = éves licensz díj + éves üzemeltetési díj.

Úgy tűnik, ROI-t számolni egyszerű, csak 2 számot kell elosztani. A nehézség abból adódik, hogy ezek a számok általában csak becslések. A várható hozam általában abból adódik, hogy jobb lesz a szolgáltatás, de hogy ez hány forintot ér, mennyivel nő ettől a cég értéke, azt csak becsülni lehet. Egy kicsit egyszerűbb a helyzet, ha egy-két ember munkája kiváltható a fejlesztéssel, mert akkor az ő fizetésük kiesik, ez kiadás csökkenést, illetve termelékenység növekedését eredményez.

A másik oldalt is nehéz számolni, mert habár kapunk egy árajánlatot a fejlesztő cégtől, de általános tapasztalat, hogy minden informatikai projekt idő- és költségtúllépéstől szenved (lásd: Szoftverkrízis), amit majd a szoftvercég CR (angolul: change request, magyarul: változtatási kérelem) formájában próbál rányomni a megrendelőre.

Nézzünk egy példát: A B12-es folyamat az egyik fő szolgáltatása a cégünknek, minden részvényes imádja. Ugyanakkor a folyamat 3. lépése kiváltható egy egyszerű adatbázis lekérdezéssel, feltéve, hogy a login folyamatba még egy plusz adatot bekérünk. A fejlesztési igény alapján kapunk egy 4 MFt-os árajánlatot, éves karbantartási díj pedig 1 MFt. Becslések szerint az egyszerűbb folyamat az első évben 4 MFt-ot, második évben 5 MFt-ot, a harmadik évben 8 MFt-os cég érték növekedést fog jelenteni a mostani értékhez képest. Vigyázat ezeket a számokat nem szabad összeadni, mert a bázis év a mostani év, és nem az előző év. A 3. évre számított ROI = 8 MFt / (4 MFt + 3 \* 1 MFt) = 8/7. Mivel ez a szám X = 3 éves távlatban nagyobb, mint 1, ezért ezt a fejlesztést érdemes megrendelni.

Általában itt még nem áll meg a ROI számítás. Kockázatokkal is számolni kell. Mi van, ha 20%-kal több lesz a fejlesztés költsége? Ez sajnos nagyon valószínű. Mennyire valószínű, hogy túlértékeltük a cégérték növekedést? Ez azt jelenti, hogy több ROI értéket is számolni kell különböző konfidenciaszinttel, magyar szóval: megbízhatósági szinttel. Ennek ismeretében tud a döntéshozó megalapozott döntést hozni.

Összegezve, a folyamatelemző feladatai:

* az ad-hoc folyamatokból szabályozott folyamatokat készít,
* a meglévő folyamatokat megismeri, dokumentálja, javítja, automatizálja,
* az ehhez szükséges fejlesztések megrendelését előkészíti,
* ROI-t számol,
* kapcsolatot tart a fejlesztést végző szakemberekkel,
* mindezt proaktív módon teszi.

Az üzleti folyamatokat már megvizsgáltuk az üzleti elemző munkakör leírásánál. Itt tovább elemezzük az üzleti folyamatok fogalmát. Két szélsőséges eset van:

* ad-hoc üzleti folyamat, és
* túlszabályozott üzleti folyamat.

Akkor mondjuk, hogy egy üzleti folyamat ad-hoc, ha nem szabályozott, nincs benne az SZMSZ-ben (szervezeti és működési szabályzat), ha úgynevezett tacit tudás kell hozzá (magyar szóval: hallgatólagos tudás). Például: Mari néni tudja, hogy a nyomtatópapír beszerzés legegyszerűbb módja, hogy telefonál Gyuri bácsinak, a raktárosnak, hogy hozzon fel két csomaggal, ha jön tenni a szépet Mari néninek.

A másik véglet, amikor egy üzleti folyamat túlszabályozott, és épp ezért kevésbé hatékony, kevésbé gyors, mint lehetne. Például a nyomtatópapír rendeléshez kell egy 13B belső megrendelés, amit egy 7C összegző lapon a megrendelési osztály osztályvezetője hagy jóvá, amiről értesíti az operatív igazgatót a 23E adatlapon.

Nyilván mindkét szélsőséges eset rossz. A folyamatelemző kollégának kell meglátni azt a pontot, amikor egy ad-hoc folyamatot már érdemes szabályozni, illetve önmérsékletet kell gyakorolni, nehogy túlszabályozza a folyamatokat.

Ettől is fontosabb a folyamatok automatizálása. Ennek előfeltétele, hogy a folyamatok jól dokumentáltak legyenek. Ehhez érdemes a BPMN (Business Process Model and Notation) grafikus nyelvet használni, ami épp üzleti folyamatok modellezésére jó. Nem szabad összekeverni a BPML (Business Process Modeling Language) nyelvvel, ami szintén ugyanerre jó, de a támogatása már régen megszűnt.

Ezután fel kell tárni azokat a részfolyamatokat, amit általában informatikai megoldásokkal, pl:

* adatbázis létrehozásával,
* szövegfelismeréssel,
* szövegbányászattal,
* szoftverrobot alkalmazásával,

gyorsítani lehet, egy-két munkás munkáját ezzel megkönnyítve, esetleg kiváltva.

Külön ki kell emelni a szoftverrobotokat, hiszen az AI fejlődésével ezek a megoldások a jegyzet írásának időpontjában kezdenek olyan pontosságot elérni, mint egy átlagos alkalmazott. Például elérhető megfizethető áron olyan szoftverrobot, ami 80-85%-os pontossággal képes eldönteni egy email-ről, hogy az termék reklamáció vagy sem. Ettől jobb pontosságot csak egy jól megfizetett tapasztalt ember képes elérni.

Fontos, hogy a javítás növelje a cég versenyelőnyét, produktivitását, profit termelő képességét, egyszóval a cég érdekében álljon.

Ugyanakkor az éghajlatváltozás miatt lehet még egy oka a folyamatok változásának, ami nem javítja se a termelékenységet, se a profit termelési képességet. Ez pedig a cég társadalmi hasznosságának növelése. Erre a szakszó a CSR (Corporate Social Responsibility, magyarul: vállalati társadalmi felelősségvállalás).

Mivel az éghajlatváltozás komolyan veszélyezteti társadalmunk jólétét, sőt az egész emberiség fennmaradását, ezért a CSR egyre fontosabb. Minden folyamatelemzőnek mihamarabb bele kell terveznie CSR lépéseket az általa gondozott folyamatokba. Mik lehetnek ezek?

Mielőtt átnézzük a valódi, előremutató CSR barát megoldásokat, nézzünk rá valamire, ami nem az, aminek tűnik. Ez a greenwashing (magyarul: zöldre festés, zöldre mosás, zöldálca), amikor úgy tűnik, hogy egy cég mindent megtesz, hogy kevesebb szén-dioxidot bocsásson ki, közben pedig nem is.

Greenwashing lehet például:

* Csomagoláson feltüntetett szén-dioxid elnyelő, vagy biodiverzitást megőrző projekt támogatása, miközben ez a támogatás csak a csomagoláson létezik, a valóságban nem.
* Kosárba tehetünk a vásárlásunk környezeti lábnyomát kiegyenlítő néhány százforintos virtuális terméket, amit lényegében minden vásárló megengedhet magának, de ez a pénz a valóságban a céget gazdagítja, nem fordítják valós környezetvédelmi programokra.
* Javasolt rákeresni az interneten konkrét példákra. Ezekkel a félrevezető lépésekkel cégek kihasználják a tájékozatlan vásárló jóindulatát, környezettudatossági szándékát, de igazából csak profitra vadásznak.

Példák a valódi vállalati társadalmi felelősségvállalásra:

* papírmentes folyamatok;
* erdőültetési projektek támogatása (vigyázat, könnyen zöldálcának tűnhet);
* biodiverzitást megőrző projektet támogatása (vigyázat, könnyen zöldálcának tűnhet);
* lebomló csomagolás használata;
* szerverek felhőbe költöztetése, zöld energiát használó felhőszolgáltató választása;
* energiahatékony (angolul: Energy Efficiency) design; az “Energy Efficiency” minőségi jelző a negyedik helyen van a “Software Architecture in Practice, Len Bass, Paul Clements, Rick Kazman, 2021” szakkönyvben.
* körfolyamatok tervezése:
  + elromlott, rossz termékek visszagyűjtése, újrahasznosítása;
  + ami az egyik folyamatban keletkező melléktermék, hulladék, az egy másik folyamatban hasznosítható bemenet; például a sörfőzés mellékterméke a melegvíz, ami üvegházi dísznövény termesztést tesz lehetővé;
  + megmaradt építkezési alapanyag meghirdetése, hátha másnak jó lesz;
  + stb...

Fontos megjegyezni, hogy a CSR lépések bevezetése ugyanúgy a vállalat vezetőjének a döntése, a folyamatelemző csak javaslatot tehet rá proaktív módon. Általában fontos, hogy a cég egésze, a munkavállalók és a vezetőség is támogassa a zöld átállást. Ilyenkor is kell ROI-t számolni, de ekkor azt is bele kell számolni a megtérülésbe, hogy hány vásárló, felhasználó fordul el a cégtől, ha a cég nem lépi meg azokat a lépéseket, amiket a versenytársai, vagy amiket a közösség elvár. Ezek nehezen becsülhető tételek. A közvélemény nyomása nélkül a cégek nehezen szánják rá magukat a CSR lépésekre, ami a folyamatelemző szemszögéből nézve csak a szokásos nehézség. A folyamatelemző feladata, hogy megelőzze a kényszerű lépéseket, mihamarabb legyenek tervei a folyamatok zöldítésére, még mielőtt a közvélemény kikényszeríti azokat.

Ahogy a rendszerelemző szakma esetén, a folyamatelemző (angolul: process analyst) szakmának is van architekt foka, a folyamatszervező (angolul: process architect). A folyamatszervező feladata a folyamat architektúra megtervezése, az ehhez kapcsolódó architekturális kérdések eldöntése. Általában ez a cégen belüli szerepkörök, illetve a szerepkörökhöz tartozó jogosultságok kialakítását jelenti. Ilyesmi kérdések ezek: Melyik szerepkörben lévő munkatárs indíthat B12-es folyamatot? Kellenek-e jóváhagyások? Milyen szerepkör hagyhat jóvá? Szerepkörök vannak vagy jogosultságok? Kell-e mindkettő? Van-e szerepkör hierarchia? Van-e jogosultság hierarchia?

### A szökőkút-dilemma

A tervezők gyakran szembe találják magukat azzal a problémával, hogy a megrendelő értelmetlen, soha meg nem térülő, felesleges, de magas presztízsű fejlesztést kér. Miért baj ez? Mert a tisztelt olvasó biztosan találkozott már elhanyagolt, réges-rég kiszáradt szökőkutakkal közel a lakóhelyéhez.

Igazság szerint egy szökőkút nagyon látványos, felvidítja a szívet, azt sugározza, hogy gazdag helyen járunk. Ugyanakkor egy szökőkút könnyen elromlik, magas a karbantartási költsége, és előbb-utóbb kiderül, nincs rá pénz.

Persze ez nem azt jelenti, hogy nincs szükség a Trevi-kútra vagy a Margitszigeti Zenélő Szökőkútra. Nagyon is szükség van rájuk mint a környék fő produkcióira, informatikai nyelven: fő funkcióira. Ugyanakkor semmi szükség egy kis költségvetésű, mégis drága karbantartású szökőkútra, ami előbb-utóbb az enyészeté lesz.

Hogy lehet ezt a példázatot az információs rendszerek tervezésére lefordítani? A tervezők gyakran találkoznak olyan követelményekkel, amiknek a ROI értéke kimagaslóan rossz. Ezek általában a megrendelő becsvágyát tükrözik, a megrendelő álomvilágában elengedhetetlen fejlesztések.

A kérdés az, hogy az ilyen követelményeknek megfelelő funkciókat megtervezzük-e, vagy inkább próbáljuk lebeszélni erről a megrendelőt? Ez nagyon nehéz kérdés, hiszen mint minden megrendelés, ez is pénzt hoz a konyhára; ráadásul a vevőnek mindig igaza van. A másik oldalon a megrendelő pénzügyi helyzetét nagyban rontják ezek az értelmetlen fejlesztések, és egy szoftvercég a hosszú távú megrendelőire alapozhatja hosszú távú terveit. Ezek nélkül nehéz kitörni az alkalmi megrendelések kutya nehéz piacáról.

A fenti pénzügyi megfontolások mellett ugyanolyan fontosak az etikai megfontolások is. Etikai kötelességünk jelezni, hogy valamely megrendelés sose fog megtérülni, vagy sem? A jegyzet szerzői szerint minden esetben, ha a megtérülés öt éven belül nem reális, akkor azt jeleznünk kell a megrendelő felé. Ennek ellenére, ha a megrendelő ragaszkodik az ilyen presztízsberuházásokhoz, akkor azt teljesítenünk kell, de vannak kivételek.

Például egyre kevésbé elfogadott a kincsesláda-jutalom a játékszoftvereken belül. Nagyon sok olyan játék van, amivel ha játszunk napi 1-2 órát, akkor kapunk egy kincsesládát, amiben akár nagyon ritka jutalom is lehet. Ez egyfajta függőséget vált ki a játékosokból: a nagyon ritka jutalom reményében hónapokon keresztül képesek játszani. Ezt a fajta jutalmazást egyre inkább elítéli a közösség, és inkább elfogadott, hogy valamennyi pontot kapunk minden nap, ami pár hónap alatt összegyűlik és báváltható egy ritkaságra, de a ritkaság megnyerése nem lehetséges, mint egy szerencsejátékban. Ha kincsesláda-jutalomrendszer fejlesztésével akarnak megbízni, egész egyszerűen utasítsuk vissza.

Hasonló példa a képalbum. Képalbumot készíteni egy CMS rendszeren (angolul: Content Management System, magyarul: tartalomkezelő rendszer) belül nagyon egyszerű. Minden kapcsolódó funkció előre kész van, könnyű a képek feltöltése, címkézése, keresése, megjelenítése. Mégis, előbb-utóbb, tapasztalataink szerint, ez egy elhanyagolt része lesz a weboldalnak, de ha mégsem, akkor pedig GDPR kérdéseket vet fel. A képen lévő emberek hozzájárultak a kép feltöltéséhez? Ha Jószagú Józsi azt kéri, hogy a róla közzétett képeket töröljük, akkor van erre folyamat? És ha minden képet le kell szedni, akkor az üres funkcióra mi szüksége lenne?

Ezek miatt, bármennyire is jó ötletnek tűnik elsőre egy képalbum menüpont, beszéljük le róla a megrendelőt: Nem sok hasznot hoz, annál több fejfájást okoz! És ha ragaszkodik hozzá a megrendelő? Második körben is próbáljuk meg lebeszélni: A képalbum fenntartási költsége aránytalanul drága a remélt megtekintési számokhoz képest. És ha még mindig ragaszkodik hozzá? Hát akkor vállaljuk el tiszta lelkiismerettel, hiszen mi kétszer is megpróbáltuk lebeszélni a megrendelőt erről a felesleges pénzkidobásról.

Hogy hol van az a határ, amikor egyértelműen vissza kell utasítani egy fejlesztést, és mikor elég csak megpróbálni lebeszélni az ügyfelet egy rossz döntésről? Ez egy nehezen megválaszolható kérdés. Hagyatkozzunk a senior kollégák véleményére.

### Összegzés és ellenőrző kérdések

Ebben a leckében azt a 3 tervezői szakmát néztük át, amik hazánkban a legelterjedtebbek. Ezek az üzleti elemző, a rendszerelemző, és a folyamatelemző. Mindhárom szakma részleteit feltártuk, az átfedéseket megmutattuk. Az üzleti elemző és rendszerszervező kiemelkedő szerepet játszanak az üzleti folyamatok hatékonyságának növelésében és a fejlesztési folyamat előzetes tervezésében. A folyamatelemzők pedig proaktívan hozzájárulnak a vállalati folyamatok javításához és automatizálásához. Az első két szakma képviselő általában szoftverfejlesztő cégeknek dolgoznak, a harmadik szakma képviselő pedig a megrendelői oldalon.

Ellenőrző kérdések:

1. Kérdés: Kinek a feladata a 3 szakma közül a közvetlen kapcsolattartás a megrendelővel?

Válasz: Mindhárom szakmára jellemző, de leginkább az üzleti elemzőre.

1. Kérdés: Kinek a feladata a rendszerterv elkészítése, illetve a backlog karbantartása?

Válasz: Ha a 3 szakma szétválik, akkor az üzleti elemző feladata a rendszerterv elkészítése és a backlog karbantartása. Ha ezek a szerepkörök nem válnak szét, akkor gyakran az üzleti elemző készíti el, tartja karban ezeket a termékeket.

1. Kérdés: Mi az üzleti elemző feladata?

Válasz: Az üzleti elemző feladata az üzleti igények megértése és dokumentálása.

1. Kérdés: Milyen dokumentumokat készít az üzleti elemző?

Válasz: Követelmény- és funkcionális specifikációkat.

1. Kérdés: Mit jelent az üzleti folyamat fogalma?

Válasz: Az üzleti folyamat gyakran ismétlődő tevékenységek sorozataként definiálható, amely értéket teremt a cég számára.

1. Kérdés: Mire van szüksége az üzleti elemzőnek a megrendelő igényeinek megértéséhez?

Válasz: Az üzleti elemzőnek alapos domain tudásra van szüksége.

1. Kérdés: Miért érdemes automatizálni az üzleti folyamatokat?

Válasz: Az üzleti folyamatok automatizálása időt és erőforrásokat takaríthat meg a cég számára.

1. Kérdés: Mi az SDLC első lépése, és milyen rövidítéssel hivatkoznak rá?

Válasz: Az SDLC első lépése a követelmények elemzése és a tervezés, amit angolul "(system) analysis and design”-nak nevezünk, vagy röviden: A&D-nek.

1. Kérdés: Mi a rendszerelemző és az üzleti elemző feladata?

Válasz: Az üzleti elemző inkább a megrendelővel tartja a kapcsolatot és az A&D feladatok első felében aktív, azaz a követelmény és funkcionális specifikációt készíti el, míg a rendszerelemző inkább a fejlesztő csapattal tartja a kapcsolatot és az A&D feladatok második felében aktív, azaz a rendszertervet készíti el.

1. Kérdés: Mi a folyamatelemző (angolul: business process analyst) fő feladata?

Válasz: Az üzleti folyamatok javítása és automatizálása proaktív módon.

1. Kérdés: Mi a különbség az üzleti elemző és a folyamatelemző között?

Válasz: Míg az üzleti elemzők a fejlesztést végző szoftverházban dolgoznak, addig a folyamatelemzők a megrendelői oldalon tevékenykednek.

1. Kérdés: Milyen szerepet játszik az ROI (angolul: Return on Investment) a folyamatelemző munkájában?

Válasz: Az ROI segít meggyőzni a vezetőséget egy-egy folyamat javításáról vagy automatizálásáról, hiszen ez mutatja meg a befektetés megtérülését.

1. Kérdés: Mit jelent az ad-hoc üzleti folyamat

Válasz: Az ad-hoc üzleti folyamat olyan, amely nem szabályozott, így a munkatársak tacit tudásán alapszik.

1. Kérdés: Mi a kockázat a folyamatok automatizálásakor?

Válasz: Az informatikai projektek idő- és költségtúllépésének lehetősége, amelynek eredményeként változhat az ROI, a megtérülési idő.

1. Kérdés: Milyen példákat említ a jegyzet a valódi vállalati társadalmi felelősségvállalásra?

Válasz: Példák lehetnek például a papírmentes folyamatok, az erdőültetési projektek támogatása vagy a körfolyamatok kialakítása.

## Számítógépes információs rendszer fogalma

Az információs rendszerek (angolul: Information Systems), egészen pontosan a számítógépes információs rendszerek (angolul: Computer Information Systems, angol rövidítése: CIS), vagy más szóval informatikai rendszerek, fogalmi rendszere a magyar nyelvű szakirodalomban szerencsére jól körbejárt kérdés. A régi rendszerszervezők jól ismerték Halassy Béla úgynevezett „sárga könyvét”, ami hasonló becenév, mint Programtervezési minták könyv beceneve, amit mindenki csak „GOF könyvnek” hív. A harmadik ilyen becenévvel rendelkező könyv a „kék könyv”. A hivatkozott könyvek:

* „A sárga könyv”: Halassy Béla, Az adatbázis-tervezés alapjai és titkai, IDG Magyarország Lapkiadó Kft., 1994.
* „A kék könyv”: Halassy Béla, Ember – információ – rendszer, IDG Magyarország Lapkiadó Kft., 1996.
* „A GOF könyv”: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz, Kiskapu Kiadó, 2004.

A GOF könyv onnan kapta nevét, hogy 4 szerzője van, és őket így négyen úgy hívták, hogy a négyek bandája (angolul: Gang of Four, angol rövidítése: GoF, vagy GOF), és ennek angol rövidítéséből jött a GOF könyv név.

A sárga és a kék könyv nevének eredetét könnyű kitalálni, a sárga könyv borítója jellegzetesen sárga, a kék könyvé pedig kék. Szerencsére mindkét könyv könnyen elérhető a Magyar Elektronikus Könyvtáron keresztül: [www.mek.oszk.hu](http://www.mek.oszk.hu).

Az alábbiakban a kék könyvben található definíciókat idézzük szó szerint, még akkor is, ha nem mindegyikkel értünk egyet, akkor is, ha az idézett szöveg helyesírása a mai szabályok szerint nem jó. Érdemes elolvasni Halassy úr eredeti gondolatait a kék könyvből, de némi magyarázattal mi is szolgálunk:

* Adat: „Az adat értelmezhető, de nem értelmezett ismeret.”
* Információ: „Az információ értelmezett adat.”
* Rendszer: „A rendszer egymással összefüggő tényezők szervezett együttese.”
* Informatikai, informatikai szakember: „Az informatika az ismeretek megismerésének, azok célszerű elrendezésének és kezelésének a tudománya. Az informatikus az a szakember, aki ebben a tudományban jártas.”
* Tevékenység: „Tevékenységnek az adatok kezelését és előállítását célzó illetve az előbbieket vezérlő műveletek szervezett egységét tekintjük.”
* Esemény: „Eseménynek az információs tevékenységet kiváltó illetve az azt lezáró momentumot nevezzük.”
* Erőforrás: „Erőforrásnak az IR fejlesztéséhez és működtetéséhez szükséges időt, pénzt és technikai eszközöket - hardver illetve szoftver - tekintjük.”
* Felhasználó: „Felhasználó az ismeretekkel kapcsolatban álló ember(csoport).”
* Szabvány: „A szabvány az IR valamilyen tényezőjére vonatkozó megegyezés.”
* Információs Rendszer (IR): „Az információs rendszer
  + adatoknak (információknak);
  + a velük kapcsolatos információs eseményeknek;
  + a rajtuk végrehajtott információs tevékenységeknek;
  + az előzőekkel kapcsolatos erőforrásoknak;
  + az információk felhasználóinak;
  + ill. a fentieket szabályozó szabványoknak és eljárásoknak

a szervezett együttese.”

* Fizikai adatterv: „Fizikai szintű - röviden: fizikai - adattervnek az ismereteknek az ábrázolását és tárolóeszközökön való elhelyezését rögzítő tervet nevezzük.”
* Fogalmi adatterv: „Fogalmi szintű - röviden: fogalmi - adattervnek a valóságnak a kompromisszumoktól mentes képét rögzítő tervet nevezzük.”
* Logikai adatterv: „Logikai szintű - röviden: logikai - adattervnek az alkalmazási környezet korlátainak megfelelően átalakított kompromisszumos tartalmi (fogalmi) tervet nevezzük.”
* Tervezési folyamat: „Tervezési-folyamatnak [design-process], azaz tervezésnek hívjuk azt az időben lezajló tevékenység-sorozatot, amelynek során az információs rendszer adott vetületekbe és szintekbe sorolt elemeit meghatározzuk illetve azokat tudatosan összehangoljuk.”
* Terv: „Terv-terméknek [design-product], azaz tervnek a tervezési-folyamat végeredményét, produktumát nevezzük.”

A fenti definíciók mind a kék könyvből származnak. Majd mindegyik definíció szerintünk is helyes, egyedül az „ismeret” szó használatával nem értünk egyet. A fenti definíciókban az ismeret fogalma nem kerül definiálásra, így ez zavarra adhat okot. E helyett mi a szimbólum fogalmára építjük fel a definícióinkat. E miatt az adat fogalmára saját definíciót adunk:

**Definíció (Jel):** Azt mondjuk, hogy a jel érzékszerveinkkel felfogható szimbólum, általában írott, vagy kimondott.

**Definíció (Adat)**: Azt mondjuk, hogy az adat értelmezhető, de nem értelmezett jelsorozat.

A többi definíció jó, kivéve az Informatikai, informatikai szakember fogalmát, ami teljesen az ismeret nem definiált fogalmára épül.

Érdekes, hogy rendszerterv szó helyett az adatterv szót használja Halassy úr, de ez abból fakad, hogy ő az adatbázis központú rendszerszervezők táborába tartozik.

Hasonlítsuk össze az információs rendszer és az üzleti folyamat definícióját:

| Az üzleti folyamat olyan  - gyakran ismétlődő tevékenység sorozat,  - amelynek bemenete  --- nyersanyag, félkész termékek,  --- adat, információ,  --- energia;  - amelyet egy cég  --- munkatársai, megbízottjai,  --- robotjai,  --- szoftveri együttműködve végeznek; ehhez  - felhasználják a cég eszközeit, amik lehetnek  --- pénzügyi eszközök (tőke, hitel, bér),  --- termelőeszközök, szerszámok, gépsorok,  --- számítógépek, szerverek, szenzorok, munkaállomások,  --- járművek, targoncák, szállító eszközök;  és ez értéket teremt. | Információs Rendszer (IR): „Az információs rendszer  - adatoknak (információknak);  - a velük kapcsolatos információs eseményeknek;  - a rajtuk végrehajtott információs tevékenységeknek;  - az előzőekkel kapcsolatos erőforrásoknak;  - az információk felhasználóinak;  - ill. a fentieket szabályozó szabványoknak és eljárásoknak  a szervezett együttese.” |
| --- | --- |

Látható, hogy a két fogalom egyes részei könnyen megfeleltethetők egymásnak:

* adat, információ = adat (információ),
* a cég eszközei = erőforrások,
* a cég munkatársai = felhasználók,
* üzleti folyamat = eljárás.

Illetve, van egy-két fogalom, ami nagyjából megfelel egymásnak. Például:

* tevékenység = információs tevékenység,
* ismételt = szervezett.

Illetve egy-két fogalom, ami csak az egyik oldalon található meg. Például:

* az információs esemény csak az IR oldalon van,
* a nyersanyag, a félkész termékek, csak üzleti folyamat oldalon található meg.

Levonható a következtetés, hogy a két fogalom nem ugyanaz. Sőt, mivel a Halassy-féle eljárás fogalom leginkább az üzleti folyamatnak felel meg, ezért ezek egymásra épülő fogalmak. Ennek a megfigyelésnek a szellemében fogalmazzuk át az információs rendszer fogalmát, illetve az erre épülő fogalmakat:

**Definíció (Szervezett Rendszer (SZR))**: Azt mondjuk, hogy egy szervezett rendszer (SZR) egymással összefüggő elemek szervezett együttese, amely működéséhez a környezetéből adatot, információt, tőkét, nyersanyagot, energiát vesz fel, salakanyagait és termékeit a környezete felé közvetíti.

**Definíció (Információs Rendszer (IR)):** Azt mondjuk, hogy egy információs rendszer (IR) olyan SZR, amely adatokat gyűjt, azokat tisztítja, rendszerezi, tárolja, feldolgozza, és az így előálló információk áramlását segíti a rendszer egyes elemei és a rendszer környezete felé.

**Definíció (Számítógépes Információs Rendszer (SZIR)):** Azt mondjuk, hogy egy számítógépes információs rendszer (SZIR) olyan IR, amely feladatai elvégzésére és automatizálására számítógépeket és azokon futó szoftvereket, mesterséges intelligenciákat használ.

**Definíció (Vállalati Információs Rendszer (VIR)):** Azt mondjuk, hogy egy vállalati információs rendszer (VIR) olyan IR, amely egy vállalat üzleti folyamataihoz szükséges információáramlást biztosítja.

**Definíció (Számítógépes Vállalati Információs Rendszer (SZVIR)):** Azt mondjuk, hogy egy számítógépes vállalati információs rendszer (SZVIR) olyan VIR, amely a vállalat üzleti folyamataihoz szükséges információáramlást számítógépekkel, azokon futó szoftverekkel és mesterséges intelligenciákkal segíti.

A fenti definíciókat így foglalhatjuk össze: Az SZR fogalmát részben Halassy úr rendszer definícióját foglalja magába, részben a fenntartható fejlődésből ismert önszerveződő rendszer fogalmát. Az önszerveződő rendszer olyan „élő” rendszer, amely környezetéből szerzi a működéséhez szükséges anyagokat, energiát, salakanyagait a környezetébe üríti és alkalmazkodik a változásokhoz. Ebből lett végül a SZR definíciója, amely egyrészt használja a „szervezett” szót, mint a Halassy-féle definíció, másrészt használja a „környezet” szót, mint az önszerveződő rendszer definíciója. Ezek után az egyes definíciók mindig az előzőre épülnek, így csak a lényeges pontokat kellett megadni, ami rövid, velős megfogalmazásokhoz vezetett. Az IR olyan SZR, amely adatokat gyűjt, információkat állít elő, ez a terméke, ezt adja el a külvilág felé, illetve ezzel segíti a belső működést. A VIR olyan IR, ahol üzleti folyamatokhoz szükséges információ áramláson van a hangsúly. Itt már nem az információ a termék, de a termék előállításához kell információ is. A SZVIR olyan VIR, ahol a folyamatokat számítógép segíti.

A VIR rövidítés egy kicsit veszélyes, mert egy másik elfogadott fogalomnak, a Vállalatirányítási Információs Rendszernek is ez a rövidítése, és ez a fogalom az elterjedtebb a szakirodalomban. Mindenesetre a jegyzetben csak az „információs rendszer” vagy a „számítógépes információs rendszer” szókapcsolatot használjuk és ez alatt az SZVIR fogalmát értjük. Ha ez félrevezető lenne, akkor a fenti fogalmak egyikét fogjuk használni a rövidítésével együtt. Továbbá, a VIR rövidítést önmagában nem fogjuk használni a fenti ok miatt.

Ezt a fejezetet lehetne folytatni a különböző számítógépes vállalati információs rendszerek (SZVIR) típusainak felsorolásával, azok jellemzésével, de itt csak a korábban említett számítógépes információs rendszer tervező szakma alapjait szerettük volna kibővíteni. Ugyanakkor nekünk, a szerzőknek, nem ez a szakterületünk. Mi, a szerzők, a programozás, a használati eset alapú tervezés, a programtervezési minták, az objektumorientált tervezési alapelvek és a jól bevált módszerek területén vagyunk szakértők, így elsősorban ezekre a területekre koncentrálunk.

## Szoftverkrízis

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

* Vagy a tervezettnél drágábban készül el (angolul: over budget),
* Vagy a tervezettnél hosszabb idő alatt (angolul: 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 lassan, de csökken. 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 szoftverkrízisnek több ismert oka is van:

* Elégtelen hatékonyság: A szoftvercégek nem elég hatékonyak, azaz adott idő alatt kevesebb jó minőségű kódot fejlesztenek, mint az elvárható lenne.
* Művészlelkű programozók: A programozók „programozóművészeknek” tekintik magukat, akik a programozást öncélú megvalósítási formának tekintik, amiért jól fizetnek.
* Félreértés: A szoftvercégek nem ismerik azt a szakterületet (angolul: domain), ahonnan a megrendelő jön és így nem értik szaknyelvét. Ez félreértéseket 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 szoftverkrízisre adott főbb válaszokat is:

* A rendszerszervezés legfőbb válasza a tervezés. Mielőtt nekilátunk a programozásnak, alaposan végiggondoljuk, hogy mit és hogyan fogunk megoldani. Ezeket a megoldásokat a tervezés dokumentumaiban, más szóval, a tervezés termékeiben (angolul: A&D project deliverables) rögzítjük. Sok év alatt az a tapasztalat alakult ki, hogy a tervezés középpontjába érdemes a használati eseteket a (angolul: use case) állítani, ami a legegyszerűbb eszköz a funkcionális követelmények rögzítésére.
* A rendszerszervezés másik válasza a módszertanok bevezetése. A módszertanok szigorúan vagy kevésbé szigorúan, de előírják a szoftverfejlesztés lépéseinek, azon belül a tervezés lépéseinek 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 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 félreértések elkerülését a két fél közt.
* 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 mnemonik 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 egységbe. A modularitás nagy áldása az, hogy megengedi, hogy több programozó készítse a programot. Illetve az aspektusok, amik annyira nem terjedtek el, de amik lehetővé teszik, hogy az olyan funkciókat zárják egységbe, amit sok osztálynak implementálnia kell, pl. naplózás, jogosultság ellenőrzés. 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) keretrendszerek, 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 szakterület-specifikus technológiák közül ezek a legismertebbek:
  + Modellvezérelt architektúra, vagy másik nevén modellvezérelt programozás (angolul: Model Driven Architecture / Development, röviden: MDA / MDD),
  + 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 csoportmunkát 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 hívniuk kell egymást. Ide tartozik minden olyan szoftver, ami több programozó együttes munkáját segíti. A fő csapatmunkát segítő technológiák:
  + Verziókövetés
  + Hibakövetés
  + Modellező eszközök
  + Függőségek leírást és feloldását segítő „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 írjunk egységtesztet, csak utána a tesztelendő metódust.
* A szoftverfejlesztés technológiájának harmadik válasza a konténer technológia megjelenése. Egy-egy szoftvernek annyi függősége van, a felhasznált applikáció szerver beállításai annyira bonyolultak, hogy a szoftvert ezekkel együtt, sőt az operációs rendszerrel együtt, bezárják egy konténerbe, ami operációs rendszer szintű virtualizációt valósít meg, és így bármilyen gépen lehet futtatni. Ebből adódik a technológia fő előnye: Nagyon könnyen lehet prototípust készíteni és azt eljuttatni a megrendelőhöz. Ez az alapja a folyamatos integrációnak (angolul: Continuous Improvement – CI), amikor nagyon sűrűn, akár sprintenként készül új prototípus.
* A szoftverkrízisre adott legújabb válasz: A mesterséges intelligencia megtanítjuk programozni. Miután a GIT verziókövető rendszer népszerű lett, megjelent a GitHub, ahol nagyon kényelmesen, és ami a fő, ingyenesen lehetett, és még most is lehet, szoftver projekteket tárolni. Rengeteg kisebb-nagyobb projekt költözött a GitHub-ra. Mindez pár év múlva nyert értelmet, amikor kiderült, hogy az itt elérhető rengeteg programmal, sok millió programsorral tanítanak fel egy mesterséges intelligenciát, a Copilot nevű MI-t. És mint kiderült, a jelenlegi AGI-k egész jól programoznak. Sok programozóban merül fel a kérdés: „Az rendben, hogy az MI-k munkahelyeket tesznek tönkre. Ez tudtuk, erre számítottunk, de mi programozók, miért a programozó szakmát nyírjuk ki először?” Nyilván, ez egy sarkítás, de jól érzékelteti, hogy milyen bizonytalanság uralkodik a szakmában, és miért kell a tervezés oktatására nagyobb hangsúlyt fektetni.

Ebben a jegyzetben főleg a tervezést, a használati eset alapú tervezést, és a használat esetek alábontására alkalmas tervezési lépéseket ismertetjük, de nem a szokásos UML központú leírást használva, bár rengeteg UML ábra található a jegyzetben, hanem inkább egy programozói szemléletet tükröző leírást használva. Azt mutatjuk be, hogy ha egy programozó átáll a tervezői oldalra, akkor hogyan látja ezt az oldalt.

## A három P betűs szó: Principles, Patterns, Practices

Ennek a jegyzetnek a középpontjában a tervezés, mint a szoftverkrízisre adott fő válasz áll; a tervezésen belül is a használati eset (angolul: use case) alapú tervezés. A használati eset alapú tervezés sajátja, hogy a magas szintű használati eset ábrákat alá kell bontani, finomítani kell. Ahhoz, hogy rugalmas, könnyen bővíthető architektúrát tudjunk tervezni, elengedhetetlen a címben szereplő három P betűs szó (röviden: 3P) ismerete. A 3P ezeket az angol szavakat, illetve fogalmakat jelölik:

* Principles, hosszabban: Design Principles, magyarul: tervezési alapelvek;
* Patterns, hosszabban: Design Patterns, magyarul: tervezési minták;
* Practices, hosszabban: Best Practices, magyarul: jól bevált módszerek.

A 3P központi szerepet játszik ebben a jegyzetben. Miután körbejárjuk a user case alapú tervezést, és már átlátjuk a tervezett rendszer funkcióit, ezután következik a funkciók kibontása, a kapcsolódó adatbázis és osztály tervek elkészítése.

Amikor megtervezzük az osztályokat, akkor alapvető kérdés az osztályok felbontása. Az egyik extrém eset, amikor csupán egy osztályunk van és az old meg minden feladatot. A másik extrém, hogy minden osztályunk nagyon kicsi, csak egy-egy részfeladat megoldását látja el. Nyilván mindkét megoldás rossz. Az első megoldás rossz, mert az osztályon belül a mező globális változó és ha egy nagy, mondjuk 10 ezer soros kódot írunk, abban a globális változók használata már megtalálhatatlan hibákat eredményez. Nyilván a másik megoldás is rossz, mert a megoldás sok-sok kicsi osztálya szükségszerűen maga után vonzza a hosszú hívási láncokat: történik valami és csak az 5-6 hívás után lesz végre egy for ciklus, ami csinál is valamit.

A két megoldás közt nagyon sok átmenet lehetséges és ez igazán tervezői kérdés, hogy melyik feladatot melyik osztály zárja egységbe. A sok lehetséges megoldás közt megtalálni egy könnyen bővíthető, könnyen újrafelhasználható megoldást, ebben segít a 3P elsajátítása.

Az alapelvek (angolul: principles) az alapvető kérdést segítenek eldönteni: Mit zárjunk egységbe egy osztályban? Azok felé milyen absztrakciós szinteket tegyünk?

Ha jól használtuk az alapelveket, akkor valószínűleg valamilyen tervezési mintához közel álló megoldást találtunk. A terven belül érdemes nevet adni a gyermeknek: itt egy stratégiát alkalmazunk, itt pedig valamilyen megfigyelő tervezési mintát. Ezek a szavak a minták jól ismert nevei. Jól ismertek a programozók körében, így könnyebben megértik ezek segítségével a tervet.

Persze úgy is megközelíthető a terv, hogy nem az alapelvek betartására törekszünk, hanem próbáljuk megtalálni azt a tervezési mintát, amit használni érdemes. és ha tervezési mintát használunk, akkor az alapelveket is betartjuk, hiszen a tervezési minták jól bevált, időtálló receptek, ami abból következik, hogy betartják a tervezési alapelveket.

Ezeken túl már csak a jól bevált módszerek alkalmazása marad, amik már a konkrét fejlesztésre adnak módszereket: használjunk sok unit-tesztet, nézzük át egymás kódját!

Ugyanakkor a jól bevált módszerek oldaláról is indulhat a terv alábontása, hiszen vannak olyan módszerek, ahol először a teszteket, vagy nagyon konkrét forgatókönyveket kell a kód előtt elkészíteni. Lásd a TDD és a BDD módszereket.

Tehát a 3P a használati esetalapú tervek alábontásában segít. Hogy ezt melyik P betű felől kezdjük, az már a tervezőn, a tervező körüli csapaton múlik, és talán a tervezés területén tapasztalható divat hullámokon is.

Van, aki a RUP idején kezdett el tervezni és ezért az UML alapú eszközöket részesíti előnyben. Van, aki a Sarum korai korszakában, ezért a tervezési mintákat részesíti előnyben, hogy könnyen bővíthető kódot tervezzen. Van, aki már a tiszta kód időszakában kezdett tervezni, ezért a TDD módszerre helyezi a hangsúlyt. Mostanában a Scrum érett korszakában járunk, ahol a Scrum-ot illik kombinálni Kanban vagy más módszertanokkal. Ezek a tervezők már jól eséllyel a SOLID elvek betartására törekszenek, illetve nem ismeretlen előttük a BDD, ami szép lassan felváltja, kiegészíti a TDD módszert.

Mit hoz a jövő? Mindenképp előtérbe kerülnek a nagy nyelvi modelleken alapuló általános mesterséges intelligenciák (AGI-k), illetve majd megjelennek az egyes területekre kifejlesztett speciális modellek is (ANI-k). Hogy ezeket hogyan fogjuk a szoftverfejlesztésre, számítógépes információs rendszerek tervezésére használni, azt pontosan még nem látható. Annyi tűnik biztosnak, hogy érdemes lesz ezeket használni, hogy növeljék a szoftverfejlesztő cégek termelékenységét.

Illetve az látszik még, hogy az inga kezd visszalengi. A módszertanok hajnalán sok-sok dokumentációval dolgoztunk, majd egyre kevesebbel, egészen az extrém programozásig (angolul: eXtreme Programming, röviden: XP), ahol lényegében nem írtunk dokumentációt, csak rengeteg unit-tesztet. Az XP fő elve, hogy ha valamit csinálunk, akkor azt csináljuk extrém jól, ha valamit nem csinálunk, akkor extrém módon ne csináljuk: ne csináljunk egyáltalán dokumentációt, de csináljunk nagyon sok unit-tesztet. Aztán ez a tendencia megfordult, újra elkezdtünk írni felhasználói sztorikat, mostanában újra kezd elterjedni a rendszerterv meglétének az igénye, ami az jelenti, hogy az inga visszafelé leng, megint egyre több dokumentációt használunk.

Valószínűleg a terjedelmes dokumentációból majd mesterséges intelligencia segítségével fejlesztünk szoftvert, ahol a kérésünkbe (angolul: prompt) bele kell majd fogalmazni a 3P-re vonatkozó tudásunkat: „Kedves mesterséges intelligencia, kérlek, a következő terv alapján készíts Java programot a Megfigyelő tervezési minta használatával és a SOLID elveket betartásával.”

Az is sejthető, hogy az egyik legújabb hype a craftmanship lesz, annak a ténynek az elfogadása, hogy az informatikusoknak maguktól kell továbbképezniük magukat, hogy ne maradjanak le az informatika gyorsvonatjáról. A craftmanship (magyarul: jómunkásembernek lenni) azt jelenti, hogy nem várhatjuk el az állam bácsitól, a cégtől, ahol dolgozunk, hogy gondoskodjon a továbbképzésünkről, hanem erről saját magunknak kell gondoskodnunk. Úgy tűnik, szép lassan átlépünk a craftmanship korszakába. Hogy ez jelenti-e egy új módszertan megjelenését, vagy marad a Scrum valamilyen változata, az még nem látszik.

Hogy a kedves olvasó milyen módszereket fog használni a használati eset alapú terv finomítására, az az egyéni tapasztalatokon túl ezen a jegyzeten is múlik, ezért igyekszünk egy alapos munkát a kedves olvasó kezébe adni! Ennek megfelelően a 3P kifejtését egy-egy fejezetben mutatjuk be. Kivétel képez a DIP (angolul: Dependency Inversion Principle, magyarul: Függőség megfordításának alapelve), amelyet még további 2 fejezetben fejtünk ki.

## A jó terv viták eredménye

Sir Tony Hoare-tól származó híres idézet szerint: „premature optimization is the root of all evil” (szó szerinti fordítása: a korai optimalizáció minden gonosz forrása; szabad fordításban: minden fejfájásunk fő oka a korai optimalizálás), ami alatt mi programozók azt értjük, hogy nem érdemes az optimalizálással foglalkozni, amíg nincs visszajelzés, hogy valamelyik része a szoftvernek lassú. Csináljunk egy mezítlábas megoldást és majd később optimalizálunk, ha kell.

A fenti idézetet Donald Knuth így magyarázza: „Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

Magyarul: „A programozók rengeteg időt vesztegetnek azzal, hogy a programjaik nem kritikus részeinek sebességén gondolkodnak vagy ezen aggódnak, és ezek a hatékonysági megfontolások valójában erős negatív hatással vannak a hibakeresésre és a karbantartásra. El kell felejtenünk a kis hatékonyságú optimalizálást, mondjuk az esetek 97%-ában: a korai optimalizáció minden gonosz forrása. Mégsem szabad elszalasztani az optimalizálási lehetőségeinket ebben a kritikus 3%-ban.”

Hoare úr gondolata sok változatban él tovább, például az extrém programozás (XP) szerint törekedjünk a legegyszerűbb megoldásra, és ha az nem elég rugalmas, akkor majd refaktoráljuk a kódot. Ehhez hasonló az iteratív módszertanok, azon belül is a Scrum gondolata, minden iterációban, ahogy kiderülnek az eddig rejtett követelmények, finomítjuk a megoldást. Illetve van egy XP alapelv, hogy „úgyse lesz rá szükség” (lásd: YAGNI), amit érthetünk az optimalizációra is: úgyse lesz szükség gyors, optimális kódra.

Ezzel szemben áll a tervezői gondolkodás, ami röviden így jellemezhető: A terv legyen rugalmas, könnyen bővíthető, az egyes modulok könnyen újrahasznosíthatók, hiszen így tudunk alkalmazkodni a folyton változó megrendelői igényekhez, illetve az újrafelhasználás a szoftvercég sikerének fő kulcsa.

Sajnos ez a kettő gyakran szemben áll. A rugalmas tervben, szükségszerűen plusz absztrakciós szintek vannak (lásd: DIP), ami egyfajta korai optimalizáció, és tényleg nehézkessé teszi a hibakeresést.

A programozói gondolkodásban is létezik a korai optimalizáció elfogadása, amit, mint láttuk, Donald Knuth is elfogad az esetek 3%-ban. Habár kevésbé népszerű, de elfogadott a 3F gondolata: Function, Form, Fitness; magyarul: Funkcionalitás, Forma, Fitnesz. Ez a 3 szó a válasz arra kérdésre: Milyen a jó a szoftver?

* Funkcionalitás (angolul: Function): A jó szoftver az igényeknek megfelelő funkcionalitással bír.
* Forma (angolul: Form): A jó szoftver könnyen karbantartható.
* Fitnesz (angolul: Fitness): A jó szoftver gyors és fit.

Az eredeti angol forrás itt tekinthető meg: <https://wiki.c2.com/?DesignForPerformance>.

Tervezői körökben elfogadott továbbá az a nézet, hogy ami egyszerű, az jó (lásd: KISS). Ez a gondolat a hadi mérnököktől származik, ahol nagyon fontos, hogy a fegyverek könnyen használhatók legyenek. Később ez a gondolat annyira elfogadott lett, hogy manapság ez már az egyik fő mérnöki alapelv.

Tehát a tervezői gondolkodást az általános megoldásra való törekvés jellemzi, a programozói gondolkodást pedig az egyszerűségre való törekvés jellemzi. Ez gyakran ellentétes, de nem kibékíthetetlenül ellentétes, hiszen a programozók is elfogadják, hogy néhol szükség van az általánosításra, az optimalizációra és a tervezők is elismerik, hogy ami egyszerű, az jó.

Ha a terv elkészítése csak a tervező feladata, akkor az valószínűleg túlzottan általános lesz, sok-sok felesleges absztrakciós szinttel. Ha a tervet egy programozó készíti, akkor az valószínűleg túl egyszerű lesz, nem elég rugalmas, nem elég újrafelhasználható, csak néhány refaktorálási kör után lesz elérhető a megfelelő rugalmasság.

A jó terv a két álláspont vitájából születik, a tervező és a vezető programozó vitájából, hogy megtalálják, mely absztrakciós szintekre van tényleg szükség, hol kell valóban optimalizálni, és mi az a rész, ahol elég egy mezítlábas megoldás, és mi az, amit akár egy mesterséges intelligencia is megírhat.

# A használati eset alapú tervezés és termékei

A számítógépes rendszerek tervezése egyidős a számítógéppel. A megfizethető számítógépek 1960-as évek elején jelentek meg. Ezzel együtt megjelent az igény is, hogy a számítógépeket felhasználják a vállalkozások. Ezzel együtt megjelent a szoftverkrízis. Erre az egyik legfontosabb válasz a tervezés.

Az első időkben a tervek középpontjában az adatfolyam állt. Ez teljesen jogos gondolat, hiszen a számítógépes program végső soron egy automata, amely a bemeneti adatokat egy többlépéses folyamat során (algoritmus) kimeneti adatokká alakítja. Az adatfolyam leírás ezt helyezi középpontba, azt vizsgálja, hogy merre kanyarog, milyen alrendszereken keresztül haladva, milyen események és tevékenységek hatására lesz a bemenetből kimenet.

Ezután nagyon sok különböző tervező iskola terjedt el. Volt olyan megközelítés, ahol az adatbázis kezelő került a központba. Itt a logikát beágyazott SQL eljárások tartalmazták, de akár a megjelenítést is. Másik iskola szerint a kicsi, együttműködő programokat kell tervezi, amelyek egy egyszerű batch-ben pipeline-nal összeköthetők. Megint másik iskola szerint a tervezés középpontjába a felhasználói felület áll, annak kell gyorsnak, kényelmesnek, egyszóval gazdagnak lennie. Az Internet megjelenésével a hangsúly átkerült a weboldalak tervezésére, majd a mobil telefonok elterjedésével a weben és a mobilon is jól kinéző, úgynevezett könnyen átméretezhető (angolul: responsive) megoldások tervezésére.

Ez a sok-sok iskola és tapasztalat végül abban forrta ki magát, hogy a RUP (Rational Unified Process) módszertan, illetve az UML megjelenésével, elfogadottá vált, hogy a tervezés középpontjában a használati esetek (angolul: use case) leírásának kell állnia. Ha így teszünk, akkor onnan folytathatjuk bármely iskola tanítását betartva.

A használati eset alapú leírása egy rendszernek végtelen egyszerű: Megmondjuk, hogy melyik szereplő (angolul: actor), melyik funkciókat használja, használhatja. Nem mondjuk meg, hogyan működnek az egyes funkciók. Nem írjuk le, hogy a felhasználók milyen céllal használják a funkciókat. Egyszerűen csak azt írjuk le, ki és mit használhat. Persze egy használati eset is túlbonyolítható, lásd később.

Mi volt az a tapasztalt, ami arra sarkalta a tervezőket, hogy egy ilyen egyszerű leíró eszközt, mint a használati eset, állítsanak a tervezés központjába?

A tervezők közös tapasztalata az, hogy **a megrendelőt nagyon könnyű félreérteni**, illetve nehéz kitalálni a gondolatait. **A megrendelő a saját szakkifejezéseit használja,** és lehet, hogy forgalomirányítás alatt teljesen mást ért, mint egy informatikus. Illetve, **ami a megrendelő számára egyértelmű,** arra a megrendelő nem pazarol szavakat, **azt egyszerűen nem mondja el**.

Mit lehet itt tenni? Minden félreértés, minden fel nem derített követelmény súlyos hónapokat jelenthet fejlesztési időben, és súlyos milliókat fejlesztési költségében. A félreértések elkerülésének egyik legegyszerűbb módja a használati eset diagramok használata.

Ezek annyira egyszerű ábrák, hogy bármelyik megrendelő megérti, képes végiggondolni, hogy minden funkció és minden elképzelhető szerepkörű felhasználó fent van-e az ábrán, és ha nincs, vagy rosszul van, akkor még azt idejekorán jelezni tudja.

**A használati eset alapú tervezés ereje a használati esetek egyszerűségében van!** Ez az a közös nyelv, amit a megrendelő és a tervező is ért. Működik akkor is, ha a megrendelő nem akar specifikációt olvasni; és akkor is működik, ha a tervező unalmasabbnál unalmasabb specifikációkat ír.

## Információs rendszerek életciklusa

A számítógépes információs rendszereket nem elég megtervezni, az csak a szoftverfejlesztés életciklusának (angolul: Software Development Life Cycle, röviden: SDLC) első nagy lépése, ami több kisebb lépésből áll, és amit több nagyobb lépés követ. Ebben a fejezetben a szoftverfejlesztés életciklusát vizsgáljuk. Ez 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 a szoftverfejlesztés életciklusának.

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

A szoftverfejlesztés életciklusa (kerek zárójelben a legfontosabb elkészítendő termékek) [szögletes 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 (tesztterv, tesztesetek, 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].



*1. 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 elöregszik. 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 hardverelemeket nem lehet pótolni. Az ilyen számítógépes információs rendszereket hívjuk kiöregedett, hagyaték rendszernek (angolul: legacy system). Valahol itt van vége a szoftverfejlesztés életciklusának. Az életciklus egyes lépéseit részletesebben is kifejtjük. A lépéseket 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”, röviden: A&D. Ez azért 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.

## Lecke 2.: Bevezetés a használati eset alapú tervezésbe

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a használati eset alapú tervezés alapjaival.

Szükséges eszközök, források: UML, PlantUML.com, PlantText.com.

Feldolgozási idő: 90 perc.

Témakörök: Használati eset alapú tervezés, UML használati eset diagram, alábontás, visszakövethetőség.

A lecke tartalma: Ebben a leckében áttekintjük a használati eset (angolul: use case) alapú tervezés alapjait. A használati esetek központi szerepet játszanak a rendszer funkcionális leírásában, melyet leggyakrabban UML diagramok segítségével modellezünk. Ezek a diagramok egyszerűek és átláthatóak, így könnyen értelmezhetők mind a tervezők, mind a felhasználók számára, ezzel segítik a két fél kommunikációját. A használati eset diagramokban megtalálhatók a szereplők (angolul: actor), a használati esetek, a rendszerhatár, valamint a kapcsolatok a szereplők és használati esetek között. A tervezés során a használati eseteket kifejtjük felhasználói sztorik (angolul: user story), illetve egyéb UML diagramok formájában. Ezt nevezzük alábontásnak. Az alábontás ellentéte a visszakövethetőség. Az alábontás során keletkező dokumentumokba és UML ábrákba fel kell tüntetni, hogy melyik használati esethez tartoznak. Így, ha lesz egy változás, akkor a tervező könnyen megtalálja, mely dokumentumokba, ábrákba kell a változásokat átvezetnie.

### Bevezetés

A használati eset (angolul: use case) fogalmát először Ivar Jacobson használta egy 1987-es cikkben. Eredetileg a felhasználási forgatókönyvek (angolul: usage scenarios) és a felhasználási eset (angolul: usage case) kifejezéseket használta, majd végül a jobban hangzó használati eset (angolul: use case) mellett döntött. A használati eset alapú tervezésről (angolul: use case driven modeling) megjelent első könyv is Ivar Jacobson nevéhez köthető, amit több más kiváló könyv is követett:

* Ivar Jacobson: Object-Oriented Software Engineering: A Use Case Driven Approach, 1992.
* Kurt Bittner, Ian Spence: Use Case Modeling, 2002.
* Craig Larman: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 2004.
* Gunnar Overgaard, Karin Palmkvist: Use Cases: Patterns and Blueprints, 2004.
* Brett D. McLaughlin, Gary Pollice, Dave West: Head First Object-Oriented Analysis and Design, 2007.
* Don Rosenberg, Matt Stephens: Use Case Driven Object Modeling with UML, Theory and Practice, 2007.
* Hassan Gomaa: Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures, 2011.
* Daryl Kulak, Eamonn Guiney: Use Cases: Requirements in Context, 2004.
* Scott Mackay: Lean Use Cases, 2019.
* Jonathan Reeve Price: Write a Use Case: Gathering Requirements that Users Understand, 2020.

A használati eset leírás eredetileg egy szöveges leírás, amely a követelmény specifikáció része, és általában ebből a 7 pontból áll:

1. Cím: Használati eset címe.
2. Cél: Ki, Mit, milyen célból akar használni?
3. Ki: A felhasználó/szereplő (angolul: actor) leírása. A felhasználó lehet személy, szerepkör, másik rendszer, vagy a vizsgált rendszer valamely alrendszere.
4. Előfeltétel (angolul: precondition): azon feltételek, amelyek szükségesek ahhoz, hogy a használati eset elindulhasson.
5. Mit: A használati eset leírása, általában lépések sorozataként megadva.
6. Alternatív lefutások: A szokásostól eltérő esetek leírása.
7. Utófeltétel (angolul: postcondition): A használati eset lefutása után milyen állapotba kerül a rendszer, milyen feltételek maradnak igazak.

Ebben a formájában már ritkán használjuk a használati eset leírást, mert a követelmények leírásán túl a folyamatok leírását is tartalmazza és az új módszertanok ezeket már általában szétválasztják. Először csak a követelményeket gyűjtjük össze és csak azután írjuk le a megoldás lépéseit.

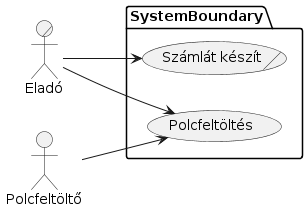
### Használati eset diagram

A legelterjedtebb, információs rendszert tervező jelölőnyelv az UML (angolul: Unified Modelling Language, magyarul: Egységes Modellező Nyelv). Ennek a jegyzetnek nem célja, hogy részletesen bemutassa az UML-t, ugyanakkor, azokat az elemeket, amiket használunk, azokat röviden bemutatjuk.

A használati eset UML diagram a következő vizuális elemeket használja:

* Szereplő (angolul: actor): pálcikaember.
* Üzleti szereplő (angolul: business actor): pálcikaember jobb alsó részben áthúzott fejjel.
* Használati eset (angolul: use case): ellipszis.
* Üzleti használati eset (angolul: business use case): ellipszis jobb alsó részben áthúzva.
* Rendszerhatár (angolul: system boundary): egy téglalap, a használati esetek mindig a rendszerhatáron belül vannak, a szereplők mindig kívül. Használata opcionális.
* Sima nyíl: Az egyes vizuális elemeket különböző típusú nyilak kötik össze. Egyelőre a legegyszerűbb nyilat mutatjuk be, a sima nyilat, aminek a jelentése egy szereplő egy használati esetet használ.

A fenti vizuális elemeket elhelyeztük egy ábrán, ami azt fejezi ki, hogy az Eladó adhat számlát is és feltöltheti a polcot is, de a Polcfeltöltő nem adhat számlát, csak a polcot töltheti fel. Itt az Eladó üzleti szereplő, a Polcfeltöltő sima szereplő. A „Számlát készít” egy üzleti használati eset. A „Polcfeltöltés” pedig sima használati eset. Ezekről bővebben a következő alfejezetekben lehet olvasni.



2. ábra: Példa használati eset diagramra

PlantUML szkript:

@startuml

left to right direction

package SystemBoundary {

usecase/ "Számlát készít" as Sz

usecase "Polcfeltöltés" as P

}

:Eladó:/ --> Sz

:Eladó:/ --> P

:Polcfeltöltő: --> P

@enduml

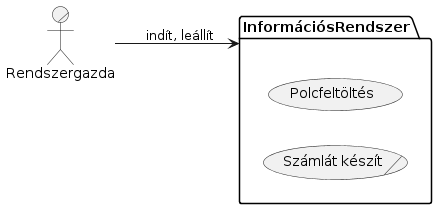
Mint látjuk, egy használati eset diagram esetén éppen olyan fontos, hogy hova mutat nyíl, és hova nem. Talán a fenti ábrán éppen ez a legfontosabb információ: Egy polcfeltöltő, aki általában egy külső munkatárs, nem adhat számlát.

### Rendszerhatár

Egy gondolat erejéig térjünk vissza a Rendszerhatárhoz (angolul: system boundary). Fentebb azt mondtuk, hogy “ez egy téglalap, a használati esetek mindig a rendszerhatáron belül vannak, a szereplők mindig kívül. Használata opcionális”. Ez az opcionálisság amiatt van, mivel a használati esetek implicit módon megadják a rendszer határait, hiszen ami nincs definiálva, az nincs is a rendszeren belül. Ugyanakkor több rendszer interakciója, vagy sok felmerült funkció kiválogatásához célszerű használni, hiszen szeparálni lehet az egyes (al)rendszerek felelősségét. Tehát nem csak a tervezés során megadott rendszerelemeket lehet általa egységbe zárni, hanem ki is tudjuk emelni azt, amiről szó volt a tervezés során. Ez nem képezi az adott rendszer részét, így a megrendelő nem keresheti a későbbiekben rajtunk, még ha emlékszik is, hogy beszéltünk róla.

Nézzünk egy-két példát:

A rendszergazda elindíthatja, illetve leállíthatja az egész információs rendszert.



*3. ábra: Használati eset rendszerhatár, első példa*

PlantUML szkript:

@startuml

left to right direction

package InformációsRendszer {

usecase/ "Számlát készít" as Sz

usecase "Polcfeltöltés" as P

}

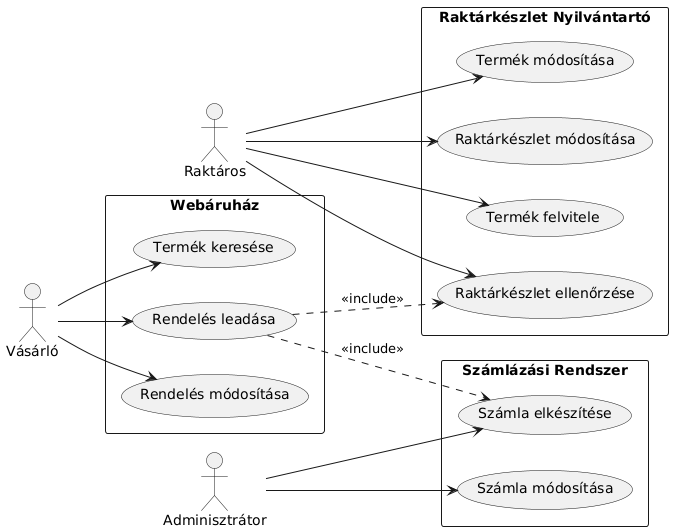
:Rendszergazda:/ --> InformációsRendszer : indít, leállít

@enduml

Nézzük az ábra tanulságait:

1. A rendszerhatárt nem muszáj rendszerhatárnak hívni, hívhatjuk akármi másnak.
2. Egy aktor mutathat a rendszerhatárra is. Ugyanakkor ennek a jelentése nem teljesen egyértelmű. Jelentheti azt, hogy az aktor az egész rendszerrel kezd valamit, mint ahogy ezen az ábrán is. Illetve jelentheti azt is, hogy a rendszer minden használati esetét használhatja.

Vagy nézzünk egy olyan példát amikor a funkciókat szeretnénk jól körülhatárolni, hogy be lehessen mutatni melyik egység feladata mi lesz, illetve mi az, amit jelen munkánkban meg kell valósítani. Vegyünk alapul egy olyan Webshop-ot, amihez korábban már elkészült a raktárkészlet nyilvántartó és számlázó rendszer.



*4. ábra: Használati eset - Rendszerhatár, második példa: egységek feladatainak körülhatárolása*

PlantUML szkript:

@startuml

left to right direction

skinparam packageStyle rectangle

actor "Vásárló" as a1

actor "Raktáros" as a2

actor "Adminisztrátor" as a3

rectangle "Webáruház" {

a1 --> (Termék keresése)

a1 --> (Rendelés leadása)

a1 --> (Rendelés módosítása)

}

rectangle "Raktárkészlet Nyilvántartó" {

a2 --> (Termék felvitele)

a2 --> (Termék módosítása)

a2 --> (Raktárkészlet módosítása)

a2 --> (Raktárkészlet ellenőrzése)

(Rendelés leadása) ..> (Raktárkészlet ellenőrzése) : <<include>>

}

rectangle "Számlázási Rendszer" {

a3 --> (Számla elkészítése)

a3 --> (Számla módosítása)

(Rendelés leadása) ..> (Számla elkészítése) : <<include>>

}

@enduml

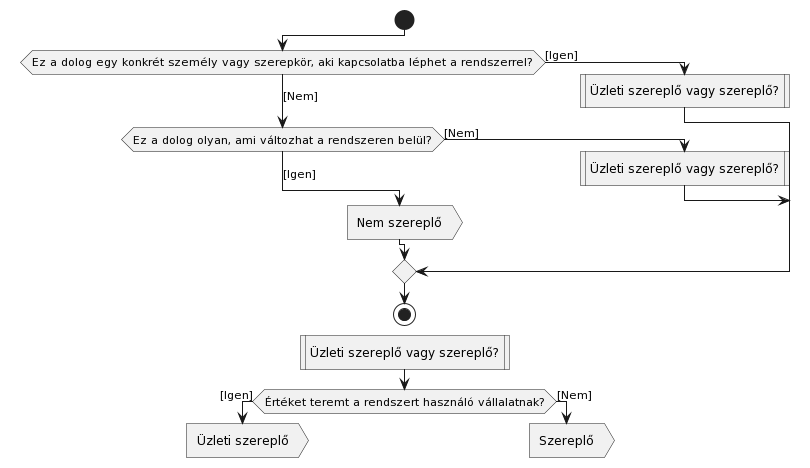
Látható, hogy a Webshop funkcióit csak a vásárlók felé akarjuk delegálni és például annak eldöntése, hogy egy termékből van-e még raktáron, nem ennek a rendszernek a dolga, csakúgy, mint egy számla készítése sem. Persze a rendszer tervezése során ezek a funkciók felmerülnek, de a rendszerhatár (angolul: System Boundary) segít a tervezés mederben tartásában a feladatok megfelelő delegálásával.

A Rendszerhatár kapcsán nem kell messzire mennünk példákért, elég csak egy amőbára, vagy egy fagylaltra, vagy az emberre gondolni. Már eleve az, hogy egy rendszernek van neve az meghatározza gondolatainkban, hogy mit tartalmazhat, de természetesen mindezekhez tanulmányaink, tapasztalataink, beszélgetéseink adnak olyan definíciókat amik miatt egyértelműnek érezzük tartalmukat. Természetesen ha csak fagylaltot emlegetünk, akkor mindenki fejében a kedvenc íze jelenik meg. Azt, hogy ugyanarról a fagylalt “rendszerről” beszélünk, pontosítani kell; meg kell adnunk az ízét, színét, összetételét, a gombóc méretét vagy a tölcsér típusát. Szoftverfejlesztés során ez a definiálás mindenképp elengedhetetlen, mert a megrendelő és a fejlesztő nem feltétlen rendelkezik ugyanazon ismeretanyagokkal, így mindenképp pontosítani szükséges mindezeket. A használati esetek készítése során el lehet kalandozni, és olyan funkciókat is beletervezhetünk, amik nem is annak a rendszernek a részét képezik. Semmi gond, kell egy másik rendszerhatárt is felvenni, és oda csoportosítani mindezeket.

### Szereplő és üzleti szereplő

A tervezés első lépése általában egy szabad riport (lásd: követelmény specifikáció, szabad riport), amikor is a tervező szóról-szóra rögzíti a megrendelő válaszát erre a kérdésre: Mik az új / megújuló rendszer jellemzői? Az így előálló szövegben szokás megkeresni az igéket, főleg a műveltető igéket. Ezek lehetnek a rendszer használati esetei, illetve a forráskód metódusai. Szokás megkeresni a főneveket is, főleg a tevékenységekhez kapcsolódó főneveket, foglalkozásra utaló főneveket, illetve a személyneveket. Ezek lehetnek a rendszer szereplői (angolul: actor), illetve a forráskód osztályai, esetleg az adatbázis táblái, az információs rendszer egyéb entitásai.

Az alábbi algoritmus segít eldönteni, hogy egy így azonosított főnév vajon szereplő-e vagy sem:



*5. ábra: Folyamatábra egy főnév szereplői körének eldöntésére*

PlantUML szkript:

@startuml

!pragma useVerticalIf on

start

if (Ez a dolog egy konkrét személy vagy szerepkör, aki kapcsolatba léphet a rendszerrel?) then ([Igen])

:Üzleti szereplő vagy szereplő?|

([Nem]) elseif (Ez a dolog olyan, ami változhat a rendszeren belül?) then ([Nem])

:Üzleti szereplő vagy szereplő?|

else ([Igen])

:Nem szereplő>

endif

stop

:Üzleti szereplő vagy szereplő?|

if (Értéket teremt a rendszert használó vállalatnak?) then ([Igen])

:Üzleti szereplő>

kill

else ([Nem])

:Szereplő>

kill

endif

@enduml

A szereplőre rögtön adunk két definíciót is:

* A szereplő olyan személy, vagy másik rendszer, esetleg, alrendszer, ami használja a rendszert.
* A szereplő olyan felhasználó, aki adatokat, illetve információkat ad vagy kap a rendszertől.

A szereplő lehet sima szereplő vagy üzleti szereplő, aszerint, hogy a rendszert használó cégnek, vállalatnak értéket teremt-e? Ha igen, akkor üzleti szereplő, ha nem, akkor sima szereplő.

Gyakorlásképpen elemezzük az alábbi szöveget: „Éva és Peti elmennek kocsit venni. Az eladó, Zoli, megmutat nekik két típust. Éva egy piros sportautó mellett dönt, azt foglalóval lefoglalózza.”

Vizsgáljuk meg, hogy kik, illetve mik a szereplők, az üzleti szereplő, és a nem szereplők? Nézzük a főneveket és rögtön nézzük, meg, hogy ezek a főnevek szereplők-e:

* Éva: Szereplő, hiszen interakcióba lép az autó eladási rendszerrel, de a szereplő neve nem Éva lesz, hanem: Vásárló, mert ebben a szerepkörben használja a rendszert.
* Peti: Ez az eset érdekes, hiszen nem lép semmilyen interakcióba a rendszerrel. Lehet, hogy csak elkíséri Évát, de az is lehet, hogy ő is egy potenciális vásárló, azaz Érdeklődő. Talán érdemes felvenni a rendszerbe egy „Érdeklődő adatainak rögzítése” használati esetet. Mivel ez valószínűleg nem lesz része az MVP-nek (angolul: Minimum Viable Product, magyarul: Minimálisan életképes termék), ezért, ha fel is vesszük, akkor is jelölni kell, hogy ez egy későbbi cél. Ekkor Peti megjelenhet, mint szereplő, mint az Érdeklődő nevű szereplő, hiszen ebben a szerepkörben lép kapcsolatba a rendszerrel.
* Kocsi: Ez egyértelműen nem szereplő, inkább az adatbázisnak lesz egy táblája, amiben az üzletben található autókat rögzíthetjük; illetve a forráskódban, mint osztály könnyen megjelenhet.
* Eladó: Üzleti szereplő. Szigorúan tekintve az információs rendszernek részei az információs rendszert üzemeltető emberek is, és akár még változó résznek is tekinthetjük őket, hiszen továbbképzéssel egy eladóból lehet üzletvezető. Ennek ellenére erősebb az az érv, hogy ez egy ember szerepköre, ezért szereplő. Mivel az üzleti tevékenységet segíti, azaz értéket teremt a cég számára, ezért üzleti szereplő. Ez a szereplő fogja rögzíteni a vásárló adatait, a foglalót. Ha ez így van, akkor mit csinál Éva, a vásárló? A vásárló kiválasztja a neki megfelelő autót. Lehet, hogy ehhez nincs egy funkció, gomb, vagy menüpont sem, de ez akkor is használati eset. Lásd később a használati eset, és az üzleti használati eset közti különbséget.
* Zoli: Üzleti szereplő, mégpedig Eladó szerepkörben.
* Típus: Nem szereplő, az adatbázis autó táblájának lesz valószínűleg egy oszlopa.
* Sportautó: Nem szereplő, az autó egy altípusa.
* Foglaló: Nem szereplő, egy nagyon fontos erőforrás, ami nélkül nincs üzlet, ez a pénz!

A fenti elemzésnek megfelelő ábrát a következő alfejezetben ismertetjük.

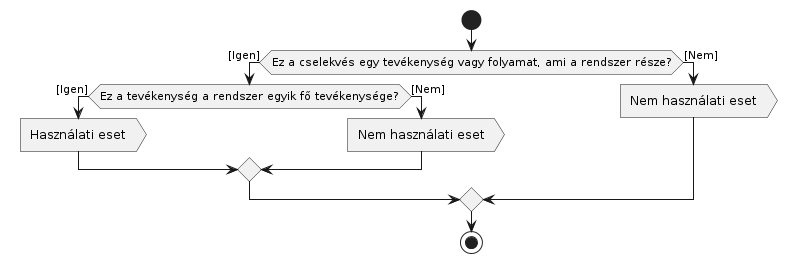
A használati eset (angolul: use case) a vizsgált rendszeren belül valamilyen esemény hatására végbemenő tevékenység, vagy folyamat. Az üzleti használati eset (angolul: business use case) egy olyan használati eset, ami értéket teremt. Az üzleti használati esethez mindig kapcsolódik egy gomb, egy menü, vagy valamilyen funkció, amit a szereplő (angolul: actor) használ. Nem üzleti használati esetnél, azaz sima használati esetnél nem feltétlenül van gomb, menü vagy funkció, amit a szereplő használhat.

A használati esetek elsősorban a funkcionális követelmények rögzítésére jó, de szöveges leírásban kibővíthetők nemfunkcionális követelményekkel is.

A használati eseteken keresztül érti meg a tervező a megrendelő üzleti folyamatait, a megrendelő igényeit. A használati esetek feltárása az első vagy a második lépés a tervezés folyamatában. Ha nem ez az első lépés, akkor általában csak a szabad riport előzi meg, aminek során a tervező szó szerint leírja, hogy mik az igényei a megrendelőnek.

Ha van ilyen riport, akkor a riportban szereplő igék, főleg a műveltető igék, a cselekvést, a használatot kifejező igék lehetnek használati esetek, vagy funkciók, persze az is lehet, hogy ezek közül egyik sem.

A következő UML aktivitási diagram segít eldönteni, hogy egy cselekvés használati eset-e vagy sem:



*6. ábra: Folyamatábra egy cselekedet meghatározására (használati eset-e)*

PlantUML szkript:

@startuml

start

if (Ez a cselekvés egy tevékenység vagy folyamat, ami a rendszer része?) then ([Igen])

if (Ez a tevékenység a rendszer egyik fő tevékenysége?) then ([Igen])

:Használati eset>

else ([Nem])

:Nem használati eset>

endif

else ([Nem])

:Nem használati eset>

endif

stop

@enduml

Egy műveltető igét magába foglaló mondatrész, egy cselekvést kifejező mondatrész a fenti ábra szerint akkor használati eset, ha a rendszer egyik fő funkcióját, tevékenységét, folyamatát írja le. Felmerül a kérdés, hogy minden cselekvés valamilyen tevékenység, akkor minden cselekvés egy használati eset? Nem, csak a fő tevékenységek! Egy tevékenység lehet ezen túl előkészítő tevékenység, előfeltétel, döntés, alternatív lépés, mellékfolyamat, egy lezáró tevékenység, egy utófeltétel, és még sok más is.

Persze akár ezeket is felvehetjük az ábránkra, de a használati eset ábráknak nem ez a lényege. Egy használati eset ábrának csak azt kell rögzítenie, hogy ki és mit csinálhat a rendszer segítségével, minden más részlet helye a használati eseteket kifejtő tervben van.

Mik a rendszer fő tevékenységei? Azok, amik a rendszer célját valósítják meg. Ha a rendszernek csak egy célja van, akkor csak egy használati eset van. Jó példa ezekre a Linux apró, pipeline-ba köthető parancsai. A wc parancs egyetlen egy dolgot csinál, megszámolja a standard input-on a szavak számát és ezt kiírja a standard output-ra. Persze, minden ilyen apró parancsnak rengeteg paramétere van, ami nem feltétlenül dönti meg az elgondolást, hogy csak egy céljuk van. A sok paraméter a cél elérésének különböző lefutásai közt választanak, lásd felhasználói sztorik.

Az is nagyon gyakori, hogy egy rendszernek több célja van. Egy autószalon nem csak értékesít, hanem szerviz szolgáltatásokat is nyújt. Ezen túl, mint minden rendes cég, kifizeti az alkalmazottakat, fizeti az adókat. Továbbá, tisztán tartja a belső tereit, sőt, a kert és az utcai rész tisztasága is fontos egy autószalonnál. Ezekhez a célokhoz mind-mind tartozik egy használati eset, ami egy fő tevékenységgel jellemezhető. Általában minden fő tevékenységnek vannak előkészítő, mellék, és lezáró tevékenységei, amik viszont nem használati esetek.

Összegezve, egy tevékenység lehet:

* fő tevékenység, ami a rendszer egy célját valósítja meg, ezáltal használati eset,
* nem fő tevékenység, hanem kiegészítő tevékenység (előkészítés, mellék szál, befejezés), vagy alternatív lefutás, ezáltal nem használati eset.

A következő kérdés, hogy egy használati eset csupán egy sima használati eset, vagy egy üzleti használati eset. Ennek eldöntésére a legegyszerűbb kérdés, hogy a felhasználói felületen kapcsolódik-e hozzá gomb, legördülő lista, menü, esetleg más funkció, vagy sem. Ha kapcsolódik, akkor üzleti használati eset, egyébként sima használati eset. Ugyanakkor ezt a megkülönböztetést nem fogjuk használni a későbbi ábrákon pont azért, hogy az ábrák minél egyszerűbbek legyenek.

Vizsgáljuk meg ugyanazt a rövid leírást, amit már láttunk:

„Éva és Peti elmennek kocsit venni. Az eladó, Zoli, megmutat nekik két típust. Éva egy piros sportautó mellett dönt, azt foglalóval lefoglalózza.”

Nézzük az igéket, a cselekvéseket és rögtön nézzük, meg, hogy ezek használati esetek-e:

* Elmegy kocsit venni: Ez nyilván használati eset, hiszen a tervezett rendszernek ez az egyik fő célja, hogy a vásárló tudjon autót venni az autó boltban. Ebből 4 használati eset készíthető:
  + Autó választás: Mivel nem egy webshopban teszi be a kosárba a vevő a kiválasztott autót, hanem közli az eladóval, hogy melyik autót szeretné, aki ezt a folyamatot el is indítja a vásárló adatainak rögzítésével, ezért csak egy sima használati eset. Ugyanakkor az autó vásárlás adatfolyam és az autó vásárlás üzleti folyamat is innen indul, tehát lehet, hogy mégis üzleti használati eset lesz egy következő verzióban.
  + Vásárló adatainak rögzítése: Üzleti használati eset, mert van hozzá funkció a rendszerben.
  + Érdeklődő adatainak rögzítése: Üzleti használati eset, mert van hozzá funkció a rendszerben.
  + Érdeklődés: Sima használati eset, hiszen nincs olyan funkció, ami ezt lehetővé tenné.
* Megmutat két típust: Ehhez sincs közvetlen funkció a rendszerben, ettől még lehetne használati eset, de még csak az sem, mert ez az autó választás előfeltétele. Valószínűleg a végső rendszerben az eladó bónusza függeni fog attól, hogy hány autót mutatott be, ebből hány bemutató lett sikeres, de egyelőre ez nincs a követelmények közt.
* Dönt egy piros sportautó mellett: Általában a döntés nem elég, pénzmozgással is biztosítanunk kell a döntésünket. Ezért ez csak a lefoglalóz használati eset előfeltétele.
* Lefoglalóz: Üzleti használati eset lesz belőle, mégpedig ez: Foglaló adatainak rögzítése. Érdekes, hogy a szöveg szerint Éva adja a foglalót, ez igaz is, de ennek az adatait már Zoli, az eladó rögzíti.

Ezután az elemzés után lássuk a fenti leírása alapján elkészült használati eset diagramokat:

| PlantUML diagram | A Vásárló az Autó boltban tud Autót választani.  PlantUML szkript:  @startuml  left to right direction  actor Vásárló as V  package AutóBolt {    usecase UC1 as "Autó választás                    ..célverzió..                    v.1.0"  }  V --> UC1  @enduml |
| --- | --- |
| PlantUML diagram | Az Érdeklődő tud érdeklődni.  PlantUML szkript:  @startuml  left to right direction  actor Érdeklődő as Ér  package AutóBolt {    usecase UC5 as "Érdeklődés                    ..célverzió..                    v.1.1"  }  Ér --> UC5  @enduml |
|  | Az Eladó tudja rögzíteni az érdeklődő adatait, de ez csak v.1.1-nek lesz a része. Eladó tudja rögzíteni a vásárló adatait, illetve a foglaló adatait.  PlantUML szkriptje:  @startuml  left to right direction  actor/ Eladó as E  package AutóBolt {    usecase/ UC2 as "Vásárló adatainak rögzítése                     ..célverzió..                     v.1.0"    usecase/ UC3 as "Foglaló adatainak rögzítése                     ..célverzió..                     v.1.0"    usecase/ UC4 as "Érdeklődő adatainak rögzítése                     ..célverzió..                     v.1.1"  }  E --> UC2  E --> UC3  E --> UC4  @enduml |

*7-9. ábrák: Példák sima, illetve üzleti használati esetekre*

Mint látható, betettünk a használati esetekbe egy kis plusz információt, hogy melyik használati eset melyik verzióban lesz elérhető, azaz mi a célverzió. Ez nem szokás. Inkább csak az a szokás, hogy az MVP-hez (angolul: Minimum Viable Product), azaz a v.1.0. rendszerhez tartozó használati eseteket rögzítjük.

Ugyanakkor fontos tudni, hogy a használati eseteken belül az egyes ellipszisekben használhatunk horizontális vonalakat akár felirattal, mint most a „célverzió”, akár felirat nélkül. Ezekkel a vonalakkal adhatunk meg a megnevezésen túli részleteket a használati esetekhez.

A másik, amit észrevehetünk, hogy a használati esetek közé is lehetne nyilakat tenni, hiszen a vásárló adatait csak akkor rögzítjük, ha már kiválasztotta az autót. Azaz az Autó választás használati esetből mehetne egy nyíl a Vásárló adatainak rögzítése használati esetbe.

Ez teljesen igaz, de ez nem egy adatfolyam ábra, hanem egy használati eset ábra, amin csak egy fontos információ van: Ki és mit használhat, azaz, melyik szereplő melyik használati esetet használhatja.

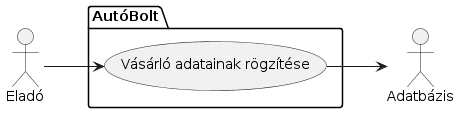
Ami még zavaró lehet, az az üzleti szereplők és a szereplők, illetve az üzleti használati esetek és a használati esetek megkülönböztetése. A jegyzet írásakor úgy döntöttünk, hogy ezeket megkülönböztetjük itt a fogalmak bevezetésénél, de később nem használjuk a megkülönböztetést, mert a gyakorlatban se szoktuk.

Persze más terv is születhetett volna ebből a fenti rövid leírásból, amit a fenti elemzés is visszatükröz. Ezeket a bizonytalanságokat érdemes sok-sok kommunikációval, pl. irányított riportok alkalmazásával megszüntetni. A tervező kérdezzen rá, hogy a megrendelő pontosan mit akar.

Habár már tárgyaltuk a két fő elemet, a szereplőket és a használati eseteket, de még nem vagyunk kész a használati eset diagramok megértésével, hiszen a lehetséges kapcsolatokat nem tártuk fel. Ezekből, mint látni fogjuk, van egy pár, de nem érdemes a használati eset diagramokat túlbonyolítani. Az a jó, ha ezeket az ábrákat meghagyjuk egyszerűnek és majd a terv kibontása során egyéb ábrákkal tárjuk fel a rejtett komplexitást.

### Szereplők a bal és a jobb oldalon

A használati eset (angolul: use case) bal oldalán gyakran emberek, emberi felhasználók, emberi szerepkörök állnak, mint szereplő (angolul: actor). A használati eset jobb oldalán gyakran alrendszerek, más informatikai rendszerek, adatbázisok, kimeneti eszközök állnak, mint szereplő. Mint látjuk, mindkét mondat végén ez áll: „mint szereplő”. Lássunk egy egyszerű ábrát:



*10. ábra: Használati eset példa bal- és jobboldali szereplőkre*

PlantUML szkript:

@startuml

left to right direction

actor Eladó as E

actor Adatbázis as A

package AutóBolt {

usecase UC1 as "Vásárló adatainak rögzítése"

}

E --> UC1

UC1 --> A

@enduml

A bal oldali szereplő, az eladó, megfelel az eddigi ismereteinknek: Az eladó egy rendszeren kívüli, azt használó személyt jelöl. Viszont a jobb oldali szereplő nem felel meg az eddig használt definíciónak, hiszen az adatbázis az informatikai rendszer része, a rendszeren belül megváltoztatható elem. Mégis, mint szereplőt használjuk. Ez egy kivétel! A használati esetek bal oldalán, és a konvenció szerint csak a bal oldalon, állhat olyan alrendszer, ami a folyamat eredményét tárolja, dolgozza fel, továbbítja. Tehát:

* A bal oldalon (szereplőből indul a nyíl) a folyamatokat indító, kezdeményező, az eseményeket kiváltó szereplő áll. A bal oldalon nem állhat a rendszer alrendszere.
* A jobb oldalon (szereplőbe vezet a nyíl) a folyamatok eredményét tároló, feldolgozó, továbbító szereplő áll, ami lehet akár a rendszer alrendszere is.

### Nyilak az UML használati eset diagramon

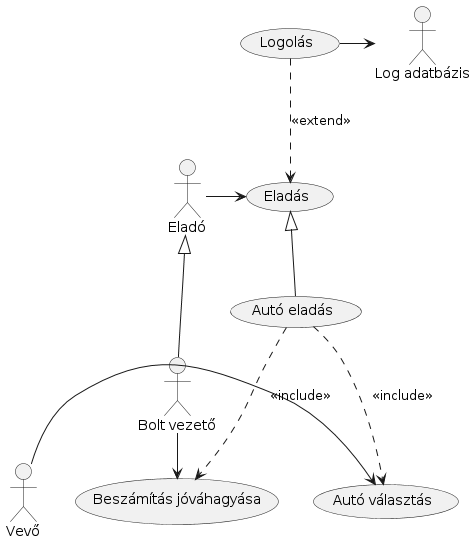
Az UML használati eset diagram (angolul: use case diagram) két szereplő (angolul: actor) megengedi a generalizáció-specializáció nyilat, ami egy háromszögbe végződő nyíl, és ami az általános felé mutat.

Ugyanezt megengedi két használati eset közt is. Illetve két használati eset közt lehet <<extend>> és <<include>> nyíl is. Illetve sztereotípia jelöléssel, azaz << és >> közt bármilyen kifejezést használhatunk.

Ez <<include>> nyilat akkor használjuk, ha két tevékenységnek van közös része, és ezt érdemes kiemelni. Ilyenkor az <<include>> nyíl a kiemelt részre mutat.

Az <<extend>> nyíl szolgál az opcionális részek kiemelésére. Ilyenkor az opcionális rész felől megy a nyíl a fő tevékenység felé. Szereplőből nem mehet nyíl opcionális tevékenység felé.

Lássunk egy példát, amiben mindezek megtalálhatók:



*11. ábra: Nyílfajták demonstrációja a használati eset diagramokban*

PlantUML szkript:

@startuml

:Vevő: as V

:Eladó: as E

:Bolt vezető: as BV

:Log adatbázis: as Log

(Autó választás) as F1

(Beszámítás jóváhagyása) as F2

(Eladás) as US1

(Autó eladás) as US2

(Logolás) as E1

E <|-- BV

US1 <|-- US2

V -right-> F1

E -right-> US1

BV -right-> F2

US2 ..> F1 : <<include>>

US2 ..> F2 : <<include>>

E1 ..> US1 : <<extend>>

E1-right-> Log

@enduml

Csúnya lett? Igen, elég csúnya és áttekinthetetlen. Csak pár perces nézegetés után lehet leolvasni az ábra tartalmát: Az Eladó végezheti az Eladás tevékenységét. Ennek speciális változata az Autó eladás, aminek része az Autó választás, ami természetesen a Vevő dolga. Ha vevő kér beszámítást (ez nincs rajta az ábrán, hogy ezt a Vevő kezdeményezi), akkor azt a Bolt vezető hagyja jóvá. A Bolt vezető egyébként egy speciális Eladó. Végül az Eladás folyamatáról a Logolás folyamata egy Log adatbázisba ír napló adatokat.

Egy használati eset ábrára nem szabad ilyen sok információt rátenni. Sokkal jobb, ha az ilyen ábrákat több kisebb ábrára szedjük szét:

|  | Az Eladó dolga az Eladás. Egy speciális eladó szerepkör a Bolt vezető, így ő is Eladhat.  PlantUML szkriptje:  @startuml  :Eladó: as E  :Bolt vezető: as BV  (Eladás) as US1  E <|-- BV  E -right-> US1  @enduml |
| --- | --- |
|  | Az Eladás egy speciális esete az Autó eladás.  PlantUML szkriptje:  @startuml  (Eladás) as US1  (Autó eladás) as US2  US1 <|-- US2  @enduml |
| PlantUML diagram | Az Autó eladás részei:  Autó választás  Beszámítási kérelem  Beszámítás jóváhagyása  PlantUML szkript:  @startuml  (Autó választás) as F1  (Beszámítási kérelem) as F3  (Beszámítás jóváhagyása) as F2  (Autó eladás) as US2  US2 .up.> F1 : <<include>>  US2 ..> F2 : <<include>>  US2 ..> F3 : <<include>>  @enduml |
| PlantUML diagram | A Vevő választja ki az autót és indítja a beszámítási kérelmet.  PlantUML szkript:  @startuml  :Vevő: as V  (Autó választás) as F1  (Beszámítási kérelem) as F3  V --> F1  V --> F3  @enduml |
| PlantUML diagram | A Beszámítási kérelem után jön a Beszámítás jóváhagyása, ami a Bolt vezető feladata.  PlantUML szkript:  @startuml  :Bolt vezető: as BV  (Beszámítás jóváhagyása) as F2  (Beszámítási kérelem) as F3  F3 ..> F2 : <<include>>  BV -right-> F2  @enduml |
| PlantUML diagram | Az Eladásnak egy technikai kiegészítése a Logolás, ami a Log adatbázisba ír.  PlantUML szkript:  @startuml  :Log adatbázis: as Log  (Eladás) as US1  (Logolás) as E1  E1 ..> US1 : <<extend>>  E1-right-> Log  @enduml |

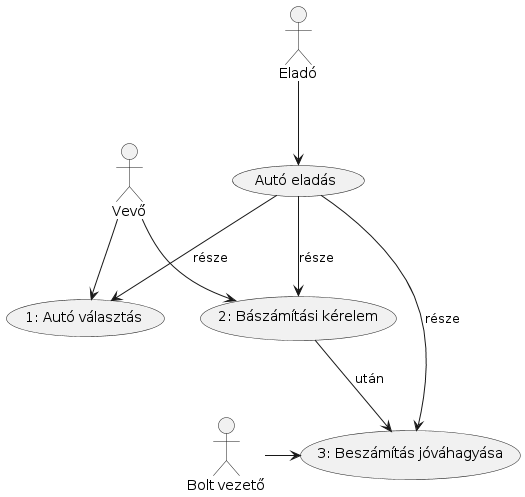
*12-17. ábra: A 11. ábra kisebb diagramokra való bontása*

Mint látható, így szétszedve, sokkal könnyebben átlátható az ábra, még akkor is, ha olyan a megrendelő által nehezen érthető nyilakat használtunk, mint az include, az extend és a generalizáció-specializáció.

Amit még észrevehetünk, az az, hogy a legelső túlzsúfolt, nehezen átlátható ábrára sehogy sem fért rá, hogy a Vevő kérheti a régi autójának a beszámítását. A szétszedés után már ez könnyen megjeleníthető.

A szétszedésnek csak egy hátránya van, egy-egy változás, vagy félreértés tisztázása során nem elég, mondjuk a Bolt vezető szereplőt egy helyen módosítani, hanem több ábrán is módosítani kell majd. Ez egy úgynevezett valamit-valamiért (angolul: trade-off) szituáció: A szétszedett ábrákat könnyebb megérteni, de nehezebb karbantartani.

Mivel a használati eset ábrák fő feladata, hogy könnyen érthetőek legyenek, ezért azt tanácsoljuk, hogy a tervező ne ragaszkodjon a fent használt angol szavakhoz. Nyugodtan írhatunk a nyilainkra magyar szöveget, eltérhetünk az UML szabványtól. A fenti példa egy egyszerűbb változata magyar nyelvű nyílfeliratokkal:



PlantUML szkript:

@startuml

:Vevő: as V

:Eladó: as E

:Bolt vezető: as BV

(1: Autó választás) as F1

(2: Beszámítási kérelem) as F3

(3: Beszámítás jóváhagyása) as F2

(Autó eladás) as US2

V --> F1

V --> F3

E --> US2

BV -right-> F2

US2 --> F1 : része

US2 --> F2 : része

US2 --> F3 : része

F3 --> F2 : után

@enduml

Pár egyszerűsítést hoztunk. Például elhagytuk az Eladás tevékenységét, aminek itt igazán nem volt szerepe, csak annyi, hogy minden eladást naplózni kell, de a naplózást is kivettük, mint egy a megrendelőt kevéssé érdeklő technikai részletet. Az az információ, hogy naplózni kell, az inkább a fejlesztőket érdeklő részlet és így egy későbbi tervnek a része.

Elhagytuk továbbá, hogy a Bolt vezető egy speciális eladó. Habár ez egy igazán fontos részlet, és ezt a részletet a megrendelő tudja visszaigazolni, vagy megcáfolni, így valamely használati eset ábrán ezt szerepeltetni kell. Ugyanakkor ez egy olyan részlet, ami ebben az ábrában nem fontos. Inkább ez egy másik, kiemelt ábrának a témája. Ezen az ábrán az autóvásárlás részei állnak a központban, ezért lehetett elhagyni az Eladó és a Bolt vezető közti kapcsolatot.

A másik újdonság az ábrán az, hogy vettük a bátorságot, és az Autó vásárlás részleteit beszámoztuk, ahogy valószínűleg követik egymást:

1. Autó választás;
2. Beszámítási kérelem;
3. Beszámítás jóváhagyása.

Ez egyáltalán nem tilos, hacsak a szoftverfejlesztő cég vezető tervezője másképp nem látja ezt a kérdést. Ugyanakkor ellene megy az elvnek, hogy a használati eset nem folyamatábra. Minden vezető tervezőnek döntenie kell a tapasztalatai alapján, hogy érdemes-e számozást bevezetni az olvashatóság érdekében.

Az utolsó észrevétel, hogy elhagytuk a szaggatott nyilakat, illetve az angol nyelvű feliratokat. Ezzel egyértelműen eltértünk a szabványos UML nyelvtől. Ráadásul az egyik <<include>> nyíl helyén a „része” szót, a másik helyén az „után” szót használtuk a jobb olvashatóság kedvéért. Megint csak a vezető tervező feladata, hogy enged-e ilyesmit a cégen belül.

Felmerülhet a kérdés, hogy miért nem azt írjuk, hogy a fenti döntéseket a cég vezetésének kell meghoznia, és miért azt írjuk, hogy a vezető tervezőnek? Ennek az az egyértelmű oka, hogy tapasztalatunk szerint a cégvezetési és a szakmai döntéseket előnyös szétválasztani egy cégen belül.

### Alábontás, és visszakövethetőség

A következő fejezetekben bemutatjuk az egyes tervezés során előálló termékeket, dokumentumokat. Ezek a használati eset központú tervezés szempontjából a használati esetek alábontása, kifejtése, a rejtett komplexitás kifejtése. Van olyan tervező eszköz, mint pl. az Enterprise Architect, ami megengedi, hogy egy UML ábrához hozzárendeljünk további UML ábrákat, mint magyarázat. Van olyan tervező eszköz, ahol ez nehézkes, vagy nem lehetséges, pl. draw.io.

Akárhogy is, a terv növekszik, előbb-utóbb elkészül a funkcionális terv, ahol már használati eseteket kifejtő felhasználói sztorikat (angolul: user story) tervezzük meg, azok felhasználói felületével. Majd jön a rendszerterv, ahol a megvalósításhoz szükséges osztályokat és algoritmusokat tervezzük meg. A tervezés ideje alatt félreértésekre derülhet fény; az igények módosulhatnak, új igények merülhetnek fel. Az Agilis módszertanok fő elve szerint ezeket a változásokat muszáj elfogadnunk, sőt proaktívan megelőzni: ha látunk egy jobbítási lehetőséget, akkor azt javasolni kell a megrendelőnek.

Tehát a tervben előremegyünk, kifejtünk részleteket, aztán hátra, hogy javítsunk egy félreértést, aztán megint előre, hogy a javításokat végigvezessük a terven, aztán még előrébb, hogy új részleteket adjunk hozzá, aztán vissza a legelejére, mert előkerült egy új követelmény. Azaz a tervben nem csak előre, hanem visszafelé is kell mozogni.

Ez csak akkor lehetséges, ha a tervünk egy fa, megvalósul az alábontás és a visszakövethetőség is:

* Alábontás: Tudjuk, hogy mely ábrának melyek a gyermek ábrái, amik a részleteket fejtik ki.
* Visszakövethetőség: Tudjuk, hogy mely ábrának melyik a szülő ábrája, ami esetleg módosul, ha a gyermek módosul.

Minden ilyen fa tetején egy használati eset áll, illetve, ha csak egy fát szeretnénk, akkor a használati esetek felé teszünk még egy víziónak nevezett magas szintű leírását a rendszernek, amik alá szervezzük be a használati eseteket. Még ettől is tipikusabb megoldás, hogy a vízió alatt van az epik-ek, mint fő funkciók szintje és csak alatta a használati esetek szintje. A használati esetek (angolul: use case) alatti szint általában a felhasználói sztorik (angolul: user story) szintje.

Ha az általunk használt tervezőeszköz támogatja az alábontást és a visszakövetést, akkor hagyatkozhatunk arra. Ha nem, akkor az egyes tervek szöveges leírásába kell beírni, hogy mely a szülő terv és melyek a gyermek tervek.

Az iparban a legelterjedtebb eszköz a Confluence, ami nem is egy tervező eszköz, hanem csak egy egyszerű dokumentumkezelő, ahol egyszerre többen is dolgozhatnak egy-egy dokumentumon. Mivel nem tervező eszköz, ezért nem is támogatja UML ábrák elkészítését, de természetesen a dokumentumokba beszúrhatjuk az UML ábráinkat. Amit viszont támogat, az a fa szerkezet, így könnyen fa hierarchiába szervezhetjük segítségével a terveinket. Ha nem is használunk ilyen eszközt, akkor is érdemes a fa hierarchiába szervezést megtenni.

### Felhasználói sztori

Manapság két nagy tervezési módszertan család terjedt el. Az Agilis és a Lean (magyarul: karcsú). Az Agilis módszertanok fő elve, hogy elfogadjuk, hogy a megrendelő újabb és újabb igényekkel, követelményekkel áll elő, a meglévő követelményeket pedig a fejlesztés során változtatja. Azért lehet ilyen nagyvonalú az Agilis módszertanok tábora, mert ezek iteratív módszertanok. Szemben a vízesés módszertannal, amikor is a szoftverfejlesztés életciklusán csak egyszer pörgünk végig, az iteratív módszertanok minden iterációjában van tervezés, fejlesztés, tesztelés és átadás. A legelterjedtebb Agilis módszertan a Scrum, amin belül az iterációt sprint-nek nevezzük. Egy-egy sprintben általában egy-két felhasználói sztorit (angolul: user story) valósít meg a csapat a tervezéstől az átadásig.

Amíg az Agilis módszertanok a szoftverfejlesztés szakmai berkein belül születtek, addig a Lean módszertanokat a termelő gyáraknak köszönhetjük. Legismertebb elve a húzó elv, ami szerint csak akkor kell legyártani valamit, ha arra van igény. Ezen túl ismert még a just in time (magyarul: épp időben) elv, ami szerint minden alkatrész épp akkor készüljön el, amikor szükség van rá, így nincs szükség raktározásra, a raktározás költségei megspórolhatók. Persze több Lean elv is van, de talán ez a kettő a legismertebb.

Szoftverfejlesztés területén a legismertebb Lean módszertan a Kanban. A Kanban egy japán szó, jelentése kártya. A Kanban módszertan esetén használjuk a Kanban táblát, amiben a gyártás különböző fázisai az oszlopok, a kártyák, legalábbis szoftverfejlesztés esetén, a felhasználói sztorik (angolul: user story) vagy azok feladatai, és minden oszlopra értelmezett egy WIP szám (angolul: Work in Progress), ami azt mondja meg, hány feladat lehet maximum egy oszlopban. Erre azért van szükség, mert egy felhasználói sztorit könnyű elkezdeni megtervezni, de nehéz lefejleszteni, a termékbe integrálni. A WIP számok nélkül elképzelhető lenne az, hogy csak tervezünk-tervezünk, de sose fejlesztünk.

A Scrum arra optimalizál, hogy a fejlesztő csapat állandó sebességgel haladjon, minden sprintben kb. ugyanannyi feladatot végezzen el. A Kanban arra optimalizál, hogy egy-egy termék gyártási ideje, azaz a gyár áteresztési sebessége, minél kisebb legyen.

Manapság ennek a két módszertannak a keveréke a legelterjedtebb. Még az extrém programozásból szokás használni elemeket, mint pl. a kód átnézés, vagy a unit tesztek intenzív alkalmazása, de a lényeg, ezeknek a módszertanoknak a középpontjába a felhasználói sztorik, illetve a feladatok állnak. Emiatt meg kell vizsgálnunk, hogy milyen kapcsolat van a használati esetek és a felhasználói sztorik közt.

A használati eset egy olyan tevékenység, ami a rendszer egyik céljának az elérését szolgálja. A felhasználói sztori pedig egy lehetséges lefutása egy használati esetnek. Egy használati esethez több felhasználói sztori tartozik. Objektumorientált szóhasználattal, a felhasználói sztori egy példánya (angolul: instance) a használati esetnek.

Nézzünk egy egyszerű példát: „Zoli, az eladó, elad Évának egy piros sport autót.” Itt a használati eset, mint az előző nagyobb példában is tárgyaltuk, az Autó eladás. Ez a használati eset többféleképpen is lefuthat, ezek mindegyike egy-egy felhasználói sztori:

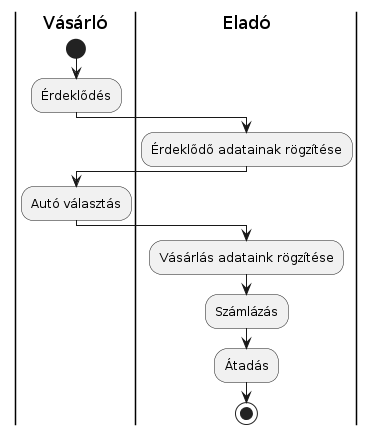
* Érdeklődő adatainak rögzítése, Vásárlás adatainak rögzítése, Számlázás, Átadás.
* Érdeklődő adatainak rögzítése, Beszámítási kérelem, Beszámítás jóváhagyása, Vásárlás adatainak rögzítése, Számlázás, Átadás.
* Érdeklődő adatainak rögzítése, Célzott reklám eljuttatása az érdeklődőknek, Vásárlás adatainak rögzítése, közte annak rögzítése, hogyan jutott el a reklám a vásárláshoz, Számlázás, Átadás.

Mint látható, mindhárom esetben ugyanaz történik, az eladó értékesít egy autót, de a részletek, a lefutás más és más, még akkor is, ha mindhárom lefutása eleje és vége teljesen ugyanaz.

Kérdezhetnénk, hogy hol van itt a sztori? Mindhárom leírás száraz felsorolása a lefutás lépéseinek. Sehol egy kis sztori. A sztori ott van, hogy a fenti lefutások ajánlott leírása nem ilyen száraz, hanem forgatókönyvszerű (angolul: scenario), amiben az egyes szereplőknek a cselekvéseit rögzítjük. Lássuk az első leírását tényleg forgatókönyv-szerűen: „Éva és Peti elmennek autót vásárolni. Évának megtetszik egy piros sportautó. Zoli, az eladó, rögzíti Éva adatait az *érdeklődő adatai rögzítése* képernyőn. Zoli még megmutat néhány autót Évának, de Évának annyira tetszik a piros sportautó, hogy végül meg is veszi. Ehhez Zoli elkéri Éva adatait, amit rögzít a *vásárló adatainak rögzítése* menüpont alatt, majd kinyomtatja a számlát. Éva fizet, és megkapja a piros sportautó kulcsait.”

Ez már tényleg egy sztori, és mint ilyen könnyen felfogható emberi aggyal. Az emberi agy ugyanis úgy fejlődött a törzsfejlődés során, hogy történetek feldolgozására, átadására nagyon alkalmas, hiszen ez szolgálta a túlélésünket a vadászó, halászó, gyűjtögető törzsekben.

A felhasználó sztorikat általában UML aktivitási diagramokkal szoktuk rögzíteni, Lássuk a fenti példa ábráját:



PlantUML szkript:

@startuml

|Vásárló|

start

:Érdeklődés;

|Eladó|

:Érdeklődő adatainak rögzítése;

|Vásárló|

:Autó választás;

|Eladó|

:Vásárlás adatainak rögzítése;

:Számlázás;

:Átadás;

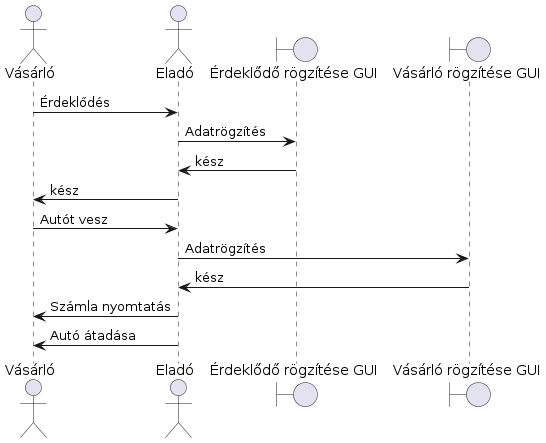
stop

@enduml

Amivel idáig nem találkoztunk, azok a partíciók, vagy más néven úszósávok (angolul: swimlane). Az úszósávokkal választjuk el, hogy melyik szereplő (angolul: actor), milyen tevékenységeket végez.

Az aktivitási diagramok nagyon hasonlítanak a folyamat ábrákra, és tényleg, pont erre jók, a folyamatok leírására. Ugyanúgy egy start pontjuk van, és legalább egy stop. A tevékenységeket itt inkább aktivitásnak hívják. Aktivitásokon túl tartalmazhat elágazásokat, üzenet küldést és fogadást is.

Aktivitási diagramon kívül használhatunk szekvencia diagramokat is, ha már világos, hogy milyen osztályaink lesznek a megvalósításban. Persze ez előtti is használhatjuk, ha kevésbé konkrét szekvencia diagramot akarunk készíteni. Nézzünk erre is egy példát, amely fenti példával egyenértékű:



PlantUML szkript:

@startuml

actor Vásárló as V

actor Eladó as E

boundary "Érdeklődő rögzítése GUI" as GUI1

boundary "Vásárló rögzítése GUI" as GUI2

V -> E: Érdeklődés

E -> GUI1 : Adatrögzítés

GUI1 -> E : kész

E -> V : kész

V -> E: Autót vesz

E -> GUI2 : Adatrögzítés

GUI2 -> E : kész

E -> V : Számla nyomtatás

E -> V : Autó átadása

@enduml

Annyival látunk többet, hogy itt már megjelennek az úgynevezett határosztályok (angolul: boundary object), ami általában egy felhasználói képernyő, vagy röviden GUI (angolul: Graphical User Interface). A határosztályok tartják a kapcsolatot a szereplőkkel.

### Egy felhasználó több szerepkörben

Nagyon gyakori jelenség, hogy egy konkrét felhasználó, mondjuk Kovács Pista, a rendszerrel több szerepkörben is kapcsolatban van. Egy egyetemi rendszer esetén egyszerre lehet hallgató, oktató, munkavállaló, és mondjuk sportklub tag. Vagy az autóvásárlás példánál maradva, egyszerre lehet, eladó, vevő, munkavállaló, és mondjuk a weblap felelőse.

A használati eset alapú tervezés ezeket a szerepköröket természetes módon elválasztja, és általában nem engedi, hogy egyszerre több szerepkörben egyszerre lépjen be a felhasználó a rendszerbe. A felhasználónak általában már a belépés (angolul: login) felületen választania kell, hogy melyik szerepkörében akarja használni a rendszert, hiszen alapvetően más-más funkciók tartoznak minden szerepkörhöz.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a használati eset (angolul: use case) alapú tervezés alapjait. A használati esetek központi szerepet játszanak a rendszer funkcionális leírásában, melyet leggyakrabban UML diagramok segítségével modellezünk. Ezek a diagramok egyszerűek és átláthatóak, így könnyen értelmezhetők mind a tervezők, mind a felhasználók számára, ezzel segítik a két fél kommunikációját. A használati eset diagramokban megtalálhatók a szereplők (angolul: actor), a használati esetek, a rendszerhatár, valamint a kapcsolatok a szereplők és használati esetek között. A tervezés során a használati eseteket kifejtjük felhasználói sztorik (angolul: user story), illetve egyéb UML diagramok formájában. Ezt nevezzük alábontásnak. Az alábontás ellentéte a visszakövethetőség. Az alábontás során keletkező dokumentumokba és UML ábrákon fel kell tüntetni, hogy melyik használati esethez tartoznak. Így, ha lesz egy változás, akkor a tervező könnyen megtalálja, mely dokumentumokba, mely ábrákon kell a változásokat átvezetnie.

Ellenőrző kérdések:

1. Kérdés: Mi áll a használati eset alapú tervezés középpontjában?

Válasz: A használati esetek.

1. Kérdés: Milyen célt szolgál a használati eset?

Válasz: A rendszer funkcionális leírására alkalmas.

1. Kérdés: Milyen diagramokat használnak leggyakrabban a használati esetek ábrázolására?

Válasz: UML használati eset diagramokat.

1. Kérdés: Milyen részek alkotják az UML használati eset diagramot?

Válasz: Szereplők, használati esetek, rendszerhatár és a kapcsolatok a szereplők és a használati esetek között.

1. Kérdés: Mi a használati esetek fő célja?

Válasz: A rendszer funkcionalitásának pontos és tömör rögzítése, valamint a tervező és a megrendelő közötti kommunikáció segítése. E miatt nem szabad a használati eset diagramokat túlbonyolítani.

1. Kérdés: Mire szolgál a használati esetek kifejtése?

Válasz: A mögöttük lévő rejtett komplexitás feltárására.

1. Kérdés: Miért fontos a visszakövethetőség a tervezés során?

Válasz: A változások kezeléséhez szükséges.

## Lecke 3.: Specifikáció írás

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a specifikáció írás jól bevált módszereivel.

Szükséges eszközök, források: Joel on Software blog „Fájdalommentes funkcionális specifikáció” című bejegyzése: <https://www.joelonsoftware.com/2000/10/02/>.

<https://markdownlivepreview.com/>

Feldolgozási idő: 90 perc.

Témakörök: Specifikáció írás, félreértések elkerülését szolgáló módszerek, a markdown nyelv.

A lecke tartalma: Ebben a leckében áttekintjük, hogyan lehet olvasmányos és hatékony specifikációkat írni a szoftverfejlesztés során, hogy elkerüljük a félreértéseket és könnyen összehasonlítható dokumentumokat készítsünk. A tervezők számára nagy kihívást jelent, hogy a megrendelők nem kedvelik a specifikációkat, noha ezek elengedhetetlenek a sikeres projektmenedzsmenthez és a félreértések elkerüléséhez. Az alfejezetek során bemutatjuk azokat a technikákat, amelyek segítenek az olvasmányos specifikációk írásában és a félreértések minimalizálásában. A specifikáció olvasmányossá tétele érdekében fontos, hogy humorral, tagolással és történetmeséléssel dolgozzunk. A szöveg legyen könnyed és érthető, hogy megakadályozzuk az agy figyelmének elkalandozását. Emellett fontos a veszélyekkel teli szöveg megteremtése a figyelem fenntartása érdekében. Végül bemutatásra kerül a Markdown nyelv, amely ideális eszköz specifikációk írásához, mivel könnyen összehasonlítható és csoportmunka esetén is nagy előnyt jelent.

### Bevezetés

A tervezők nagy bánata, hogy a megrendelő nem szívesen olvas specifikációt, habár ez lenne a kulcsa a sikeres szoftverfejlesztésnek, a félreértések elkerülésének. És ha már itt tartunk, félreértés, sajnos ez nagyon gyakori, mintha a megrendelő és a szoftverfejlesztő cég két külön nyelvet beszélne. Ebben a rövid alfejezetben azokat a technikákat vesszük sorra, ami segít olvasmányos specifikációk írásában, a félreértések elkerülésében, és könnyen összehasonlítható dokumentumok írásában.

### Olvasmányos specifikáció készítése

Tegyük fel, hogy épp egy fontos specifikációt írunk. 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 vérszomjas tigristől. Így, ha arra akarjuk rávenni, hogy unalmas dokumentumokat értelmezzen, akkor hamar elkalandozik agyunk figyelme, olvasás közben nem a szövegre figyelünk, mert nincs benne semmi veszély. Ugyanakkor tapasztalatból tudjuk, hogy vannak olyan könyvek, amiket szívesen olvasunk, pl. Rejtő Jenő könyveit. Tehát a feladatunk csak annyi, hogy olvasmányosan írjunk. Ezt kreatív írás tanfolyamokon tanítják. Egy ilyen tanfolyam elvégzése minden rendszerszervezőnek ajánlott.

Szerencsére néhány egyszerű tanácsot betartva könnyen írhatunk könnyen olvasható dokumentumokat. Először is, legyünk viccesek, például használjunk abszurd személyneveket, lehetőleg állandó jelzőkkel: „Amikor Gyorskezű Györke, a titkárok 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! Használjunk rövid sorokat, amit szemünk át tud fogni. Használjunk sok felsorolást, táblázatot, ábrát. Képeket érdemes elhelyezni, akkor is, ha nem közvetlenül kapcsolódik a témához, de a „mesébe” beleszőhető.

### Három olvasmányos példa

Az emberi agy könnyebben ért meg történeteket, mint bármi mást. Ezért érdemes törekedni, hogy egy történetet meséljünk el, amibe beleszőjük 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.”

„Gyorskezű Györke, a titkárok 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. Biztos, ami biztos. Tudja, hogy a dokumentumtárban minden, az eredeti levélhez tartozó levél lekérdezhető.”

„Gyorskezű Györke, a titkárok 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 titkárok gyöngye. Bátyjának, a lusták leglustábbikának válasza csak ennyi: „Megyek! Vegyél patront.” Gyorskezű Györke, a titkárok gyöngye szerencsére félreértette, milyen patront, 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 egy komoly megrendelő bugyutának 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 vész el a lényeg, a rendszer leírása.

Utolsó tanács, hogy a szövegünk legyen veszélyekkel teli. Ez segít ébren tartani figyelmünket. Agyunk szerint a környezetünk veszélyekkel terhes, minden pillanatban betoppanhat egy vérszomjas tigris, és ha mégse, akkor viszont lehet pihenni, akár aludni is. 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: <https://www.joelonsoftware.com/2000/10/02/>

### Félreértések elkerülését szolgáló módszerek

Minél kevesebb a félreértés a tervben, annál jobb lesz a terv minősége. A félreértések mindenképp kiderülnek, ha máskor nem, akkor az átvételi teszt, vagy a használat során. Ugyanakkor egy ilyen későn feltárt félreértés helyreigazítása nagyon drága. Minél hamarabb fény derül egy félreértése, annál olcsóbb kijavítani. Nézzünk egy klasszikus példát a félreértésre:

Megrendelő: Szeretnék egy járművet, amivel reggelente eljuthatok a munkahelyemre.

Kivitelező: Mik a jármű fontos tulajdonságai?

Megrendelő: Gyorsan eljussak a munkahelyemre. Kényelmes legyen.

Erre a kivitelező leszállít egy biciklit egy kényelmes üléssel.

Megrendelő: De ez nem véd meg az esőtől! Ezt így nem veszem át!

Erre a kivitelező a bicikli kormányára egy esernyőt erősít.

Megrendelő: De ezen nincs hely a táskámnak! Ezt így nem veszem át!

Erre a kivitelező 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 lelki 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 mást értett az alatt. Ez nagyon veszélyes! Nagyon sok egyszerű fogalom (ember, autó, kényelmes) mást és mást jelenthet a megrendelőnek és a tervezőnek.

A követelményspecifikáció nehézsége, hogy a megrendelő a számára evidens követelményeket nem mondja el. Ennek a problémának több ismert megoldása is van:

* Használati esetek gyűjtése: A 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 kérdezzünk 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 elégszünk 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 rejtett követelményre jöhetünk 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 félreértettük-e a megrendelő követelményeit. E nélkül általában csak az implementáció után derül fény a félreértésekre. 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 félreértés elkerülhető a segítségével.

### A Markdown nyelv rövid bemutatása

A markdown nyelv a GitHub elterjedésével vált népszerűvé. Majdnem mindegyik GitHub oldalon találunk egy README.md állományt, ami Markdown nyelven készült.

A Markdown nyelv azért annyira népszerű, mert a vizuális elemeket, mint a felsorolás, táblázat készítés, stb…, visszavezeti egyszerű szöveges elemekre, így ugyanannak a dokumentumnak két verziója könnyen összehasonlítható diff-fel, ami egy Word, vagy PDF dokumentumra nem mondható el. Ez csoportmunka esetén nagy előny. Ezért specifikációk írására ezt a nyelvet ajánljuk. Kiegészítve PlantUML szkriptekkel nagyon könnyen írhatunk olyan tervet, ami alapján egy AGI vagy egy ANI jó minőségű forráskódot tud írni.

A Markdown nyelv itt található leírását ChatGPT v.3.5. készítette a következő prompt-ra válaszként:

„Kérlek, készítsd el egyetemi jegyzet stílusban a Markdown nyelv bemutatását!”

ChatGPT válasza:

Természetesen, íme az egyetemi jegyzet stílusú Markdown bemutatása:

# Markdown

---

## Bevezetés

Markdown egy könnyen tanulható, szöveg formázási nyelv, melynek célja egyszerű és olvasható szövegek írása. John Gruber és Aaron Swartz fejlesztette ki 2004-ben, hogy lehetővé tegye a felhasználók számára, hogy egyszerűen formázzák a szövegüket az interneten. A Markdown széles körben elterjedt a webfejlesztés, a dokumentáció írás és az online kommunikáció területén.

## Alapok

A Markdown nyelv alapvetően szövegformázási elemekből áll, melyek speciális szintaxisokkal vannak jelölve.

- \*\*Címek\*\*: A címeket a "#" karakterrel jelöljük, például "# Cím", "## Alcím", stb.

- \*\*Félkövér és Dőlt betűk\*\*: Félkövér betűk készítéséhez két csillagot ("\*\*") vagy aláhúzást (\_) használhatunk. Dőlt betűk esetén egy csillagot ("\*") vagy aláhúzást (\_) használhatunk.

- \*\*Listák\*\*: Számozott listákhoz egyszerűen számozott sorokat használhatunk (pl. "1. elem"), míg felsorolásokhoz csak egy vagy több csillagot ("\* elem") vagy kötőjelet ("- elem") kell használnunk.

- \*\*Linkelés\*\*: Linkeléshez a hagyományos HTML <a> elem használható: [szöveg](url).

- \*\*Képek\*\*: Képek beszúrásához használható a kép URL-je: ![alternatív szöveg](kép\_url).

## Speciális elemek

A Markdown további speciális elemeket is tartalmaz a kényelmes és hatékony szövegformázás érdekében.

- \*\*Kódblokkok\*\*: Kódblokkokat három backtick (```) kezdése és vége közé lehet írni.

- \*\*Táblázatok\*\*: Táblázatokat létrehozhatunk a cellák elválasztásával ("|") és a fejléc és a tartalom elválasztásával ("---|---|---").

- \*\*Idézetek\*\*: Idézeteket ">" karakterrel jelölünk.

- \*\*Vonalak\*\*: Vonalakat húzhatunk a "----" vagy "\_\_\_\_" karakterek használatával.

## Felhasználás

A Markdown rendkívül sokoldalú és könnyen használható. Gyakran használják:

- \*\*Blogokban és weboldalakon\*\*: A Markdown gyakran használt formázási nyelv a blogokban és weboldalakon, mivel egyszerű és gyors módja a szövegek formázásának.

- \*\*GitHub\*\*: A GitHub és más verziókezelő rendszerek Markdownot használnak a readme fájlok formázására és a kód dokumentálására.

- \*\*Dokumentációk\*\*: A fejlesztők gyakran Markdownot használnak a projekt dokumentáció írásához, mivel könnyen érthető és könnyen karbantartható.

## Összegzés

A Markdown egy egyszerű, átlátható és hatékony módszer a szövegek formázására és strukturálására. A gyors és könnyű használat miatt széles körben elterjedt a webfejlesztés és a dokumentáció írás területén.

A ChatGPT remek összefoglalója után itt van még egy link, ahol sok példát és egy előnézetet adó megoldást is találunk: <https://markdownlivepreview.com/>.

### Miért Markdown? Mert a kontextus váltás káros!

Kicsit mélázzunk még el a Markdown nyelven. Mire kellhet? Miért ne használjunk valamilyen dokumentumszerkesztő programot, ami eleve olyan fájl kimenetet biztosít, amit akár az ügyfél is könnyen el tud olvasni? Természetesen nem azt mondjuk, hogy ezt kötelező használni, hanem ha megismerkedünk vele és megszokjuk, akkor napi munkánk során könnyebben tudunk a mindennapos tevékenységek mellett dokumentálni is.

A legtöbb programozó, vagy tesztelő komfortos munkakörnyezete valamilyen integrált fejlesztői környezet, vagy egy egyszerű szövegszerkesztő, vagy egy parancssori terminál, amennyiben erről váltani kell egy bonyolultabb szövegszerkesztőre, akkor azzal időt veszíthetnek. A számítástechnika, de akár az élet több területén is tapasztaljuk, hogy a kontextusváltás, vagy környezetváltás (angolul: context switching) időbe és energiába telik. Nem csak amiatt, mert egy másik alkalmazást kell betölteni, lehet az adott adatformátumhoz még telepíteni is kell azt, de a gondolkodásmódot is át kell alakítani a dokumentum íráshoz. Az is gyakori gond, hogy egy monitoros munkakörnyezetben nehezen fér el minden alkalmazás ablaka előttünk, váltogatni kell köztük.

Lényeges az is, hogy a komfortzóna elhagyását az ember halogatja (esetünkben ez lehet például a fejlesztői környezet), ami egy tervezés, vagy dokumentáció készítés esetében azt jelenti, hogy hamarabb fogunk a forráskód megírásába, mint annak megtervezésébe. Tehát célszerű olyan eszközt használni ezen munkákhoz is, ami továbbra is otthonos érzést biztosít. Egy programot nem szövegszerkesztőben írunk, viszont ezen eszközök egyszerű szövegfájlok elkészítését támogatják. Ugyanakkor ez a kimenet nincs szépen megformázva, tehát az eddigi követelmény, hogy olvasmányos legyen a dokumentációnk, nem valósul meg, vagy ha kinyomtatná valaki, akkor nem azt kapná, mint ami a képernyőn megjelent. Írhatunk magával ragadó anyagot, ha az olvasót elrettenti annak formázása, márpedig az lesz az első amit megpillant, és persze nincs második első benyomás.

Tehát mindenképp kell valamilyen jelölés amivel tagolni, kiemelni tudjuk a tartalmat. Ezen jelöléseket magába a szövegbe helyezhetjük el úgynevezett metaadatokkal. A számítástechnikában használt legtöbb jelölőnyelv (angolul: markup language) gépi feldolgozásra tartalmaz metaadatokat, gondoljunk csak a HTML vagy XML nyelvekre. De a közismert szöveges dokumentum formátumok (pl: doc, docx, odt) is jelölő nyelvet használnak a tartalom formázására, amik viszont nem humán feldolgozásra szolgálnak. Mindezekkel szemben a Markdown nagyon fontos sajátossága, hogy a jelölései az olvasó számára is értelmezhetők, jobban kiemelik a tartalmat már akkor is, ha nincsenek a jelölések kivitelezve, például csak jelezzük, hogy valamit \*\*félkövér betűkkel\*\* szeretnénk kiemelni. (Itt például a \*\*félkövér betűkkel\*\* látható, hogy fontosabb tartalmat takar, amit egyébként egy Markdown értelmezésre képes alkalmazás mint **félkövér betűkkel** írt szöveget jelenít meg.)

A Markdown-ra mint segítségre gondoljunk, ami leveszi a formázás megjelenítésének terhét a vállunkról. Nem kell fejezeteken gondolkodnunk, és azok alfejezetekre bontásán, vagy hogy miként fog majd mindez megjelenni. A dokumentum tartalmára koncentrálhatunk és a végleges megjelenítés feladatát átadhatjuk másnak, vagy egy másik alkalmazásnak. Így már egy megbeszélésen is gyorsan lehet jegyzetelni egy egyszerű szövegszerkesztővel, és menet közben “formázhatjuk” is Markdown jelölések segítségével mindezeket; nem kell egérrel kattintgatni, vagy speciális billentyűkombinációkat fejben tartanunk mindezekhez. Elmentve pedig egy kis méretű, kvázi formázott szöveget kapunk.

A Markdown egyszerűsége és elterjedtsége miatt nagyon sok megoldás létezik Markdown dokumentum más fájlformátumba való átalakítására, ezzel pedig létrejön annak lehetősége, hogy a tervezési, vagy fejlesztési dokumentáció egységes módon készüljön, de bármikor átalakítható legyen mások igényének megfelelő formátumra.

További információért érdemes ellátogatni John Gruber oldalára:  
 <https://daringfireball.net/projects/markdown/>

Végezetül pedig emeljünk emléket Aaron Swartz munkásságának is, hiszen az ő általa 2002-ben készített atx jelölőnyelv volt nagy hatással a Markdown szintaxisának kialakításában. <https://en.wikipedia.org/wiki/Aaron_Swartz>

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük, hogyan lehet olvasmányos és hatékony specifikációkat írni a szoftverfejlesztés során, hogy elkerüljük a félreértéseket és könnyen összehasonlítható dokumentumokat készítsünk. A tervezők számára nagy kihívást jelent, hogy a megrendelők nem kedvelik a specifikációkat, noha ezek elengedhetetlenek a sikeres projektmenedzsmenthez és a félreértések elkerüléséhez. Bemutattuk azokat a technikákat, amelyek segítenek az olvasmányos specifikációk írásában és a félreértések minimalizálásában. A specifikáció olvasmányossá tétele érdekében fontos, hogy humorral, tagolással és történetmeséléssel dolgozzunk. A szöveg legyen könnyed és érthető, hogy megakadályozzuk az agy figyelmének elkalandozását. Emellett fontos a veszélyekkel teli szöveg megteremtése a figyelem fenntartása érdekében. Végül bemutatásra került a Markdown nyelv, amely ideális eszköz specifikációk írásához, mivel könnyen összehasonlítható és csoportmunka esetén is nagy előnyt jelent.

Ellenőrző kérdések:

1. Kérdés: Miért nem szívesen olvas specifikációt a megrendelő, és mi lehet ennek a következménye?

Válasz: A megrendelők általában nem szívesen olvasnak specifikációkat, mert unalmasnak vagy nehezen érthetőnek találják azokat. Ennek következménye lehet, hogy félreértések alakulnak ki, és a végén olyan szoftvert kapnak, amit nem vártak.

1. Kérdés: Milyen trükkökhöz kell folyamodni annak érdekében, hogy a megrendelő olvassa és megértse a specifikációt?

Válasz: A specifikáció olvasmányos, vicces, tagolt, történetszerű és veszélyekkel teli kell legyen. Ezek segítenek fenntartani az agy figyelmét és könnyebben érthetővé teszik a dokumentumot.

1. Kérdés: Milyen módszereket érdemes használni a félreértések elkerülésére?

Válasz: Jó bevált módszer a használati esetek gyűjtése, szerepkör felvétele, prototípuskészítés, és az hasonló megoldások bemutatása, melyek segítenek kideríteni és megérteni a rejtett követelményeket.

1. Kérdés: Miért jó választás a Markdown nyelv használata specifikációk írásához?

Válasz: A Markdown nyelv egyszerű és könnyen összehasonlítható dokumentumokat tesz lehetővé, ami ideális csoportmunkához és verziókezeléshez.

## Lecke 4.: Követelményspecifikáció

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a szabad riport, az irányított riport készítés módszereivel, ezek alapján a követelményspecifikáció készítés módszereivel.

Szükséges eszközök, források: Joel on Software blog „Fájdalommentes funkcionális specifikáció” című bejegyzése: <https://www.joelonsoftware.com/2000/10/02/>.

Feldolgozási idő: 80 perc.

Témakörök: szabad riport, irányított riport, követelményspecifikáció.

A lecke tartalma: Ebben a leckében áttekintjük a követelményspecifikáció elkészítésének folyamatát és fontos elemeit. A követelményspecifikáció az adott projektben elvárt követelményeket tartalmazza, amelyek meghatározzák a szoftverfejlesztés irányát és célját. Az alapvető részei közé tartozik a jelenlegi helyzet leírása, a vízió, a rendszerre vonatkozó pályázatok, törvények, rendeletek, valamint a használati esetek és az igényelt üzleti folyamatok modellje. A szabad és irányított riportok fontosak a követelmények körülhatárolásában és pontosításában. A szabad riportok segítenek megérteni az új rendszer működését és együttműködését a meglévő rendszerekkel. Az irányított riportok pedig a követelmények pontosítására és számszerűsítésére szolgálnak. A riportok elemzése során meghatározzuk a rendszer fő funkcióit, a használati eseteket és a nem célokat. Ez segít a megrendelővel való egyeztetésben és az elkészített dokumentumok pontosításában.

### Bevezetés

Az előző fejezetben megismerhettük a követelményspecifikáció (angolul: requirement specification) helyét a szoftverfejlesztés életciklusában. Itt az elkészítéséhez szükséges technikákat ismertetjük.

A követelményspecifikáció alapjai a megrendelővel készített riportok. Főbb részei:

* Jelenlegi helyzet leírása.
* Vízió: Vágyálomrendszer leírása.
* Nem célok.
* A rendszerre vonatkozó pályázat, törvények, rendeletek, szabványok és ajánlások felsorolása.
* Használati esetek.
* Jelenlegi üzleti folyamatok modellje.
* Igényelt üzleti folyamatok modellje.
* Követelménylista.
* Megfeleltetés, hogyan fedik le a használati esetek a követelményeket.
* Irányított és szabad szöveges riportok szövege.
* Fogalomszótár.

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

### 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 rendszerében, illetve a backlogban.

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 esetet.

Ahhoz, hogy felfedezzük, hogy az elmondott rendszer lyukas, azaz nem minden esetet fed le, szükség van rendszerszemléletre. Minden programozónak van algoritmikus gondolkodása és szerencsére sokuknak van rendszerszemlélete. Belőlük lehet rendszerelemző, 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.

### Irányított riport

Egy-két szabad szöveges riport után következik az irányított riport. Erre általában egy egyoldalas kérdőívvel készülünk. Lehet hosszabb is a kérdőív, de tapasztalat szerint egy egyoldalas kérdőív esetén is egy-két óra kell az irányított riporthoz.

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?).

Kétféle irányított riportról beszélhetünk:

* tisztázó riport,
* alrendszerriport.

A tisztázó riport, ahogy a nevében is benne van, tisztázó kérdéseket tartalmaz. Ezek a kérdések általában egy fogalom vagy folyamat felderítésére szolgálnak.

Az alrendszerriport 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 alrendszerriportot 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 ellentétektől terhes a megrendelő. Ennek egyik fő oka, hogy a számítógépes információs rendszerek bevezetése változást jelent a megrendelő életében, ami lehet, hogy néhány munkatársnak nem érdeke. Általában a szoftverfejlesztő céghez 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 juttassunk el kérdőíveket.

### Funkcionális és nemfunkcionális követelmények

Minden olyan követelmény, amihez kapcsolódik majd valamilyen gomb, menü, képernyő, egyéb funkció, az funkcionális követelmény.

Minden, olyan követelmény, ami nem a közvetlen működésre vonatkozó követelmény, az úgynevezett nemfunkcionális követelmény. Ide tartozik például, hogy az alkalmazás legyen gyors, szép, könnyen használható, akár 1000 felhasználó is használhassa egyszerre.

Nagyon tipikus nemfunkcionális követelmény, hogy az alkalmazás színeiben tükrözze a cég színeit. Például, ha a cég domináns színe a világoskék, akkor az alkalmazásban is legyen a világoskék egy meghatározó szín.

Másik hasonlóan tipikus követelmény, hogy minden lista művelet kevesebb mint egy másodperc alatt végbemenjen, még akkor is, ha a legolcsóbb előfizetést használja a felhasználó.

Harmadik tipikus, nemfunkcionális követelmény, hogy az alkalmazás legyen reszponzív, azaz jól nézzen ki mobilon, tableten, laptopon, és asztali gépen is.

### Riportok elemzése

A riportok elemzésének célja a rendszer behatárolása, a funkcionális és a nemfunkcionális követelmények meghatározása, a fő funkciók, azaz a használati esetek feltárása, és az úgynevezett nem célok lefektetése. Ezek mindegyikét fontos a megrendelővel elfogadtatni, megbeszélni. Összegezve:

* A vízió megfogalmazása.
* Célok és nem célok meghatározása.
* A nem célok felsorolása a nem célok című fejezetben.
* A célokat lefedő fő funkciók feltárása, ezekből lesznek a használati esetek.
* A szereplők feltárása a már jól ismert szereplő / nem szereplő aktivitási diagram segítségével.
* Ha megvannak a szereplők, érdemes újra átgondolni a használati eseteket.
* Szerepkörök feltárása, ez segít a szereplők pontosabb elnevezésében, leírásában.
* Funkcionális és nemfunkcionális követelmények feltárása.
* Követelménylista elkészítése.
* Fogalomszótár elkészítése.
* Egyeztetés a megrendelővel.

### Szereplők és használati esetek

Habár az előzőkben nagy mélységben tárgyaltuk a használati eseteket, mint a használati eset központú tervezés fő alapegységét, mégis itt is meg kell újra említenünk, hiszen a követelményspecifikáció készítés az a lépés, amiben létrehozzuk a szereplőket, a használati eseteket, és a használati eset ábrákat. Az itt leírt fogalmak a jegyzet legelső 2015-ös verziójából valók, és így nem teljesen esnek egybe az előzőekben leírt fogalom rendszerrel. Mégis meghagytuk ezt a régebbi megértését ugyanazoknak a fogalmaknak, hogy lássa a tisztelt olvasó, hogy a megértésnek vannak fokozatai. Illetve, visszaolvasva a 2015-ös verziót, egész jól kiegészíti az újat.

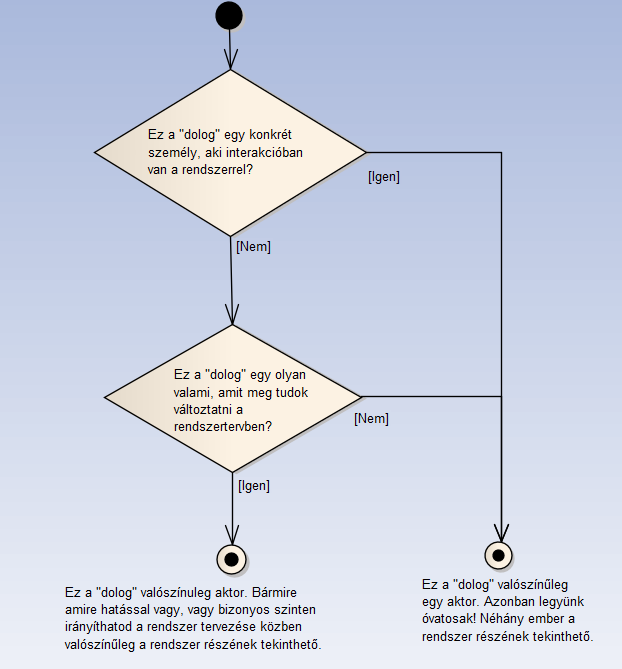
A használati eset (angolul: use case) a rendszer és a felhasználók közötti kommunikációt írja le. Olyan eseménysor, amely során a szereplő (angolul: actor) számára érték, például információ, keletkezik. A rendszer belső működési részlete (pl. adattovábbítás) nem használati eset. Lehet olyan használati eset is, amelyet időzítő indít.

Egy használatieset-ábra részei:

* rendszerhatár,
* szereplők,
* használati esetek.

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 eldönteni, hogy a szabad szöveges riport szövegében talált főnév szereplő, vagy idegen szóval, aktor vagy sem.



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.

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 funkcionalitást?
* 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

Szereplő leírása:

* Mit vár a rendszertől?
* Felelősségi köre, jogosultságai.
* Fizikai környezete, ebből adódó fizikai megszorítások.
* Hány szereplőpéldányt képvisel?
* Milyen gyakran használja a rendszert?
* Van-e számítástechnikai tapasztalata?

A használati esetek megtalálásához nézzük át a riportokat és tegyük fel a következő kérdéseket minden megtalált szereplő szemszögéből:

* Ez a szereplő mit vár a rendszertől? Mire használja?
* Ez a szereplő fog-e adatot bevinni a rendszerbe, módosítani vagy törölni? Hogyan?
* Ezt a szereplőt kell-e informálni a rendszerben bekövetkezett valamely eseményről? Hogyan?
* Ez a szereplő használ-e egyéb, nem elsődleges funkciót? Melyek ezek a funkciók?
* Ez a szereplő indíthatja, vagy leállíthatja a rendszert? Hogyan?
* Ez a szereplő végez-e karbantartást? Milyen funkciók támogatják ebben?
* Ezt a szereplő tájékoztatni kell valamely hirtelen bekövetkezett eseményről, hibáról? Hogyan?
* Ez a szereplő módosíthatja, vagy konfigurálhatja a rendszer működését? Hogyan?

A feltárt használati eseteket össze is vonhatjuk, főleg, ha kiderül, hogy alternatív lefutásai ugyanannak a használati esetnek. Illetve, 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” és „mit” csinál a rendszerrel a 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. Fontos, hogy az informatikai rendszerre értelmezhető állapotokat határozzunk meg. Opcionális.
* *Egyéb:* nemfunkcionális követelményeket írjuk ide.

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.

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 információ cserék sorozatát mutatjuk meg. A használati eset végén a szereplő számára értéket teremt a rendszer.

A rendszert nem funkciók, 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 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.

### 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 azzal, hogy proaktív módon próbálnak elémenni a változásoknak és inkább a tervező javasolja a változásokat, nem várja meg, hogy megrendelő jöjjön elő azokkal. Bár néha ez elkerülhetetlen.

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 tulajdonságai?
* Kérem, mondja el forgatókönyv-szerűen, 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, …
* Használati eset ID: A követelményhez kapcsolódó használati eset azonosítója, pl.: UC1, UC2, …
* 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 vágyálom, vagy reálisan csak későbbi verzióban elérhető követelmények. Az 1.0. verzió felel meg az MVP-nek (angolul: Minimum Viable Product). 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. Megjegyzéseket tartalmazhat a programozóknak, tesztelőknek vagy más szerepkörű olvasóknak.

Az ennek megfelelő követelménylista-sablon a következő:

| **Modul** | **ID** | **UC ID** | **Név** | **V.** | **Kifejtés** |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

### Fogalomszótár

A fogalomszótár szerepe, hogy hidat verjen a megrendelő által használt nyelv és az informatikusok által használt nyelv közé. Habár úgy tűnik, hogy a megrendelő és a szoftverfejlesztők is magyarul beszélnek, mégse értik meg egymást, mert mind a ketten különböző szakzsargont használnak.

Ennek a problémának a kivédésére találták ki az üzleti elemzőket, akiknek az a feladatuk, hogy megtanulják a megrendelő nyelvét, beletanuljanak a megrendelő terület (angolul: domain) specifikus tudásába, és így tolmácsként tudjanak közvetíteni a két fél között.

Ugyanerre jó a modern módszertanokban a termék tulajdonos (angolul: Product Owner – PO) szerepkör, aki annyira átlátja a megrendelő igényeit, hogy a megrendelő nevében képes priorizálni a feladatokat, elmagyarázni a megrendelő szemszögéből az egyes igényeket.

Akár van a projektben üzleti elemző, akár nincs, akár van PO, akár nincs, érdemes fogalomszótárt készíteni, amiben leírjuk azoknak a szavaknak, szókapcsolatoknak a jelentését, amit a szabad és irányított riportok szövegében felleltünk, mint a tervezőnek idegen, vagy nem érthető szót, és a megrendelő segítségével megértettünk.

Az igazán nagy veszélyt azok a szavak jelentik, amikhez mindkét szakzsargonban jelentés társul. Ezeket érdemes a szótár elejére mozgatni. Illetve érdemes a rövidítésekből is külön listát készíteni. Egyébként pedig ez egy sima ABC sorrendes szótár. Ha van a megrendelőnek esetleg Web2-es megoldása, ahol a saját új alkalmazottainak gyűjti a saját fogalmait, folyamatait, akkor azt kérjük el, az sokat segíthet.

Gondoljunk arra, hogy a tervezőt, azaz minket, bármikor elüthet a busz, kaphatunk egy nem visszautasítható állásajánlatot. Ilyenkor át kell adnunk a tervet, a megszerzett domain tudást az utódunknak. Ha jót akarunk a volt cégünknek, illetve az utódunknak, akkor egy szép bő fogalomszótár is ott lesz a terv elemei közt.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a követelményspecifikáció elkészítésének folyamatát és fontos elemeit. A követelményspecifikáció az adott projektben elvárt követelményeket tartalmazza, amelyek meghatározzák a szoftverfejlesztés irányát és célját. Az alapvető részei közé tartozik a jelenlegi helyzet leírása, a vízió, a rendszerre vonatkozó pályázatok, törvények, rendeletek, valamint a használati esetek és az igényelt üzleti folyamatok modellje. A szabad és irányított riportok fontosak a követelmények körülhatárolásában és pontosításában. A szabad riportok segítenek megérteni az új rendszer működését és együttműködését a meglévő rendszerekkel. Az irányított riportok pedig a követelmények pontosítására és számszerűsítésére szolgálnak. A riportok elemzése során meghatározzuk a rendszer fő funkcióit, a használati eseteket és a nem célokat. Ez segít a megrendelővel való egyeztetésben és az elkészített dokumentumok pontosításában.

Ellenőrző kérdések:

1. Kérdés: Mi a követelményspecifikáció alapja, és mi a legfontosabb része?

Válasz: A követelményspecifikáció alapja a megrendelővel készített riportok. A legfontosabb része a követelménylista.

1. Kérdés: Milyen kérdések közbeszúrása elfogadott a szabad szöveges riportoknál?

Válasz: Elfogadott, ha a megrendelő önmagával ellentmondásba keveredik, vagy ha egy fogalom nem világos, vagy ha az elmondottak nem fednek le minden estet.

1. Kérdés: Miért fontos a rendszerszemlélet a szabad riportok elemzése során?

Válasz: A rendszerszemlélet segít felfedezni, ha az elmondott rendszer lyukas, azaz nem minden esetet fed le.

1. Kérdés: Mi a különbség a szabad és az irányított riportok között?

Válasz: A szabad riportokban a megkérdezett szabadon beszél, míg az irányított riportokban általában egyoldalas kérdőívek segítik a riport elkészítését.

1. Kérdés: Mire irányul a riportok elemzése a követelményspecifikáció készítése során?

Válasz: A riportok elemzése során a rendszer behatárolása, a funkcionális és nemfunkcionális követelmények meghatározása, a fő funkciók feltárása és a nem célok lefektetése történik, illetve a használati esetek meghatározása történik.

1. Kérdés: Mi a használati eset szerepe a rendszerfejlesztés során?

Válasz: A használati esetek a rendszer és a felhasználók közötti kommunikációt írják le. Egy használati eset a rendszer egy fő funkcióját írja le.

1. Kérdés: Mi a fogalomszótár célja a szoftverfejlesztés során?

Válasz: A fogalomszótár hidat ver a megrendelő által használt nyelv és az informatikusok által használt nyelv közé, megkönnyítve a kommunikációt a két fél között.

## Lecke 5.: Funkcionális specifikáció

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a funkcionális specifikáció elkészítésének jól bevált módszereivel.

Szükséges eszközök, források: Joel on Software blog „Fájdalommentes funkcionális specifikáció” című bejegyzése: <https://www.joelonsoftware.com/2000/10/02/>.

Feldolgozási idő: 100 perc.

Témakörök: követelményelemzés, képernyőtervek, felhasználói sztori.

A lecke tartalma: Ebben a leckében áttekintjük a funkcionális specifikáció készítésének lépéseit. A funkcionális specifikáció a rendszer működését a felhasználók szemszögéből írja le. Fontos elemei közé tartoznak a felhasználói sztorik és a képernyőtervek, melyek segítenek a rendszer funkcióinak megértésében és kialakításában. A követelményelemzés során részletesen elemezzük a követelményeket, finomítjuk a használati eseteket és képernyőterveket készítünk. A használati esetek és a képernyőtervek alapján felhasználói sztorikat készítünk. Ezek segítenek a rendszer használatának átgondolásában, elősegítve a tervezési folyamatot és a felhasználói elégedettséget.

### Bevezetés

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, 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.
* Vízió: Vágyálomrendszer leírása.
* A rendszerre vonatkozó külső megszorítások: pályázat, törvények, rendeletek, szabványok és ajánlások felsorolása.
* Jelenlegi üzleti folyamatok modellje.
* Igényelt üzleti folyamatok modellje.
* Használati esetek.
* Követelménylista.
* Felhasználói sztorik, forgatókönyvek.
* Képernyőtervek.
* Megfeleltetés, hogy a felhasználói sztorik lefedik-e a használati eseteket, a követelményeket.
* 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 mélységben kell elkészíteni. Általában elmondható, hogy a modern módszertanok használatieset-központúak, azokat felhasználói sztorik segítségével bontják alá, fejtik ki.

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

### Követelményelemzés

A követelményspecifikációan leírt követelményeket ebben a feladatban elemezzük, hogy feltárjuk a követelmények megvalósításához szükséges funkciókat, gombokat, listákat, menüket, és ezeket képernyő tervekbe szervezzük. E közben gyakran finomítani kell a használati eseteket, illetve azokat lehet részletezni felhasználói sztorikkal, ahol már hivatkozni lehet az egyes gombokra, funkciókra. Fontos, hogy ezeket megmutassuk a megrendelőnek, azok a megrendelő elfogadja. Összegezve:

* Fő és mellék funkciók meghatározása, ennek megfelelően használati esetek finomítása.
* Listák, gombok, menük feltárása
* Felhasználói felületek meghatározása.
* Felhasználói sztorik, vagy más néven, forgatókönyvek megírása.
* Egyeztetés a megrendelővel.

### Képernyőtervek

A képernyőtervek mutatják meg, hogy mely funkciók kerülnek egymás mellé, melyik képernyőről mely képernyőre juthatunk. A grafikus felhasználói felület (angolul: graphical user interface, röviden: GUI) tervezésének ismert néhány szabálya, amelyeket itt csak felületesen ismertetünk. A következő elveket (kivéve az első két elvet) Ian Sommerville Szoftverrendszerek fejlesztése című könyvéből 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, funkciócsoportok.
* 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.

Felhasználói felület tervezésére könnyen találhatunk úgynevezett drótváz (angolul: wireframe) tervező eszközt. Ezek közül a legismertebb a [www.figma.com](http://www.figma.com).

A drótvázra feltehetünk listákat, menüket, gombokat, egyéb vizuális elemeket. A gombok akár kattintható gombok is lehetnek, de igazi logika nincs mögöttük, legfeljebb annyi, hogy átvisznek a következő képernyőre. Habár nincs mögötte logika, mégis a drótvázból könnyen generálható forráskód, méghozzá HTML, illetve CSV állományok.

A drótváz tervező eszközök nagyban hozzájárulnak, hogy egy korai prototípust tudjunk mutatni a megrendelőnek, aki a látottak alapján eldöntheti, hogy jók-e a képernyőtervek.

### Felhasználói sztori, forgatókönyv

A felhasználói sztorik, vagy más néven, a forgatókönyvek a rendszer egy-egy tipikus felhasználását, egy használati eset egy-egy lefutását mutatják be. A forgatókönyvnek általában van egy célja, például egy iktató rendszerben: Levél érkeztetése, címzett értesítése, dokumentumtárba 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 használná a rendszer, csak még képzeletben.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a funkcionális specifikáció készítésének lépéseit. A funkcionális specifikáció a rendszer működését a felhasználók szemszögéből írja le. Fontos elemei közé tartoznak a felhasználói sztorik és a képernyőtervek, melyek segítenek a rendszer funkcióinak megértésében és kialakításában. A követelményelemzés során részletesen elemezzük a követelményeket, finomítjuk a használati eseteket és képernyőterveket készítünk. A használati esetek és a képernyőtervek alapján felhasználói sztorikat készítünk. Ezek segítenek a rendszer használatának átgondolásában, elősegítve a tervezési folyamatot és a felhasználói elégedettséget.

Ellenőrző kérdések:

1. Kérdés: Mi a funkcionális specifikáció célja?

Válasz: A funkcionális specifikáció a rendszer működését a felhasználók szemszögéből írja le.

1. Kérdés: Melyek a funkcionális specifikáció fontosabb részei?

Válasz: A funkcionális specifikáció legfontosabb részei a használati esetek, a képernyőtervek, és a felhasználói sztorik.

1. Kérdés: Mi a feladata a követelményelemzésnek?

Válasz: A követelmények alapján finomítjuk a használati eseteket és készítjük el a képernyőterveket.

1. Kérdés: Milyen fontos szempontokat kell figyelembe venni a grafikus felhasználói felület tervezésekor?

Válasz: Teljesség, átláthatóság, konzisztencia és felhasználói jártasság.

1. Kérdés: Miben segítenek a felhasználói sztorik?

Válasz: Segítenek a rendszer használatának átgondolásában, elősegítve a tervezési folyamatot és a felhasználói elégedettséget.

## Ütemterv

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.

Az árajánlat legfontosabb része az ütemterv. Az ütemterv határozza meg, hogy mely funkciók kerülnek be a rendszer következő verziójába és melyek maradnak ki. 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, a 2-es nagyon hasznos funkció, a 3-as kényelmi funkció. A 0-s prioritás jelentése, hogy az adott funkció már kész van. Természetesen ezek csak ajánlások.

Érdemes az ütemtervben tartalékot 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: <https://www.joelonsoftware.com/2000/03/29/painless-software-schedules/>.

### Napidíj

A napidíj megállapításánál szokásos az a módszer, hogy kiszámoljuk a cégünk működési költségeit és a napidíjat úgy állítjuk be, hogy 50%-os kihasználtság esetén nullszaldós legyen a cég. A napidíjnak fedeznie kell a programozó bérét a bérre rakodó adókat, az egy programozóra jutó bérleti költségeket és rezsit, a menedzsment fizetését és egyéb járulékos költségeket.

Egy cég általában nem egy, hanem több napidíjjal dolgozik. Általában legalább megkülönböztetjük a szenior (azaz tapasztalt), a medior (azaz középtapasztalt) 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.

### 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 konstruktívköltség-modellnek fordítjuk.

A modell első változata a vízesés módszertanhoz 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 (angolul: cost drivers) definiál, amelyek 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\*(SLOC/1000)^B
* Fejlesztésre szánt hónapok száma = C\*(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, kevéssé specifikált feladat.
* Semi-detached project – Félig-meddig projekt: közepesen nagy fejlesztőcsapat, gyakorlott és gyakorlatlan programozókkal, jól és kevésbé jól specifikált követelményekkel.
* Embedded project – Beágyazott projekt: jól specifikált követelmények jól definiált megszorításokkal.

Az egyes projekttípusok esetén a konstansok értéke:

| **Projekt típusa** | ***A*** | ***B*** | ***C*** | ***D*** |
| --- | --- | --- | --- | --- |
| 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: <https://strs.grc.nasa.gov/repository/forms/cocomo-calculation/>
* COCOMO II: <http://softwarecost.org/tools/COCOMO/>

## Árajánlat

Az 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 erősségeit, referenciáit. Ezek az ajánlólevelei. A vezetői összefoglaló általában nagyon rövid, egy-két oldal. A felhasznált technikák bemutatásánál elsüthetünk olyan varázsszavakat, amik megnyitják a megrendelő pénztárcáját, pl.: Mesterséges intelligencia, deep learning, 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 meglévőekben felfedezett hibák javítását. Pluszszolgáltatásként ajánlhatunk 1 évesnél hosszabb ingyenes hibajavítást.

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 cégünk 24 órás felügyelettel, 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 hagyjunk időt. Ennek megfelelően a hibajavítás lehet 1 munkanapos, 2 munkanapos, 1 hetes vagy 2 hetes határidejű. A kritikus hibáknál, 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 térni. 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 3-5 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 mindig vannak költségei a vételáron felül, amivel a megterülésnél számolni kell. Ezek általában a következők:

* rendszergazdák bére,
* szoftver-, hardverhibákból adódó termeléskiesés.

A gazdasági adatokat gyakran megvalósíthatósági tanulmányba foglaljuk.

### 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.

Vegyünk egy tipikus projektütemezést:

* F1, adatbázisterv elkészítése, 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:



### Kritikus út

A kritikus út módszert (angolul: 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. Egy részfeladat közvetlenül függ egy másiktól, ha csak annak elvégzése után kezdhetünk hozzá. Ebből lesz a függőségi gráf, miután START és STOP csúcsot 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 STOP-ba 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 (angolul: 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 nagyobb 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 csökkenthető, 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 programozónk van, akkor N(N-1)/2 kommunikációs út létezik köztük. Könnyen beláthatjuk, hogy egy új programozó felvétele négyzetesen, matematikai jelöléssel O(N2), növeli a kommunikációs utak számát. A sok megbeszélés időt emészt fel, ami csak további késést okoz.

A függőségi gráfba elhelyezhetünk mérföldköveket (angolul: milestone) is. Minden mérföldkőnél átadás történik, ami általában fizetséggel is jár. Ha a megrendelő elfogadja az átadott részeket, akkor egy teljesítési igazolást / jegyzőkönyvet kapunk, ami alapján kiszámlázhatjuk 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 másképp képzelték az eddig elkészült részeket. Ha minden rendben van, akkor kérjünk teljesítési 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 tesszük 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 kezelni szeretnénk, akkor már érdemes projektmenedzsment szoftvert használnunk. Erre a feladatra például jól megfelel az MS Project.

## 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 más. 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 megvalósíthatóságának 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.

## Lecke 6.: Rendszerterv

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a rendszerterv fogalmával, változataival és elkészítésének néhány módszerével

Szükséges eszközök, források: -

Feldolgozási idő: 40 perc.

Témakörök: Határosztály-tervezés, menühierarchia-tervezés, storyboard.

A lecke tartalma: Ebben a leckében áttekintjük a rendszerterv fontosabb részeit és az elkészítésének módszereit. A rendszerterv alapja a funkcionális specifikáció. A rendszerterv írja le, hogy hogyan kell a tervezett funkciókat megvalósítani. A logikai rendszerterv csak az üzleti logika megszorításait veszi figyelembe, ezzel szemben a fizikai rendszerterv a fizikai, pénzügyi megszorításokat is. A határosztály-tervezés és a menühierarchia-tervezés segít abban, hogy meghatározzuk az adatbeviteli felületeket és a menüstruktúrát. A Storyboard-modellezés felsőbb szintű nézetet ad arról, hogy melyik ablakból milyen szolgáltatások érhetők el. Ezekkel a módszerekkel finomítjuk tovább a képernyőterveket, a felhasználói sztorikat. A logikai rendszerterv részletezi az üzleti folyamatok modelljét, a követelményeket a logikai megszorítások szempontjából vizsgálja, leír egy ideális megoldást, mintha végtelen erőforrásaink lennének. A fizikai rendszerterv az osztályterveket, az adatbázistervet, a tesztterveket és a telepítési terveket foglalja magában, annak figyelembevételével, hogy erőforrásaink korántsem végtelenek. Mindkét terv célja a rendszer megvalósításának részletes leírása és specifikálása.

### 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 lépéseket 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 legfontosabb része az úgynevezett megfeleltetés, ami megmutatja, hogy a követelményspecifikációban felsorolt minden megvalósítandó 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 időpont, vagy csak egy időintervallum. Ezeket olyan részletességgel adja meg, hogy a tervező részvétele 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.

\* Költözés 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

\* Költözés, 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 alrendszere). 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 terveztünk, inkább legyen tartalék időnk, mint feszített ütemezésünk, mert váratlan "apróságok" biztosan be fognak következni, csak most még nem tudjuk, hogy mik.

### A rendszerterv fejezetei

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 (angolul: 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:
  + Funkcionális 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 funkcionalitását leíró osztályok. Részei:
  + Rendszerszereplők
  + Rendszerhasználati esetek és lefutásaik
  + Határ osztályok
  + Menühierarchiák
  + Képernyőtervek
* 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 tűzfal, az milyen portokat engedélyez. Ha vannak megvásárolt komponenseink, azokat is itt kell megadnunk. Részei:
  + Vásárolt szoftverkomponensek és külső rendszerek
  + Hardver és hálózati topológia
  + Fizikai alrendszerek
  + 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 kapjunk. Részei:
  + Perzisztencia osztályok
  + Üzleti logika osztályai
  + Kliens oldal osztályai
* Tesztterv: Lefekteti a tesztelés elveit, folyamatát és kontrollját. Meghatározza a fő teszteseteket. 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 teszttervek. Általában csak akkor készül el, ha már egy verziót átadtunk és a következő verziót tervezzük.

### 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 amely:

* 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, vagy annak egy része is. Például egy keresési ablakban a keresőmező és a találati lista két különböző határosztálynak tekinthető.

Határosztályok közötti függőségeket, hogy melyik határosztályból hová lehet eljutni, úgynevezett kollaborációs diagramokkal mutatjuk meg. A diagramokon a határosztályok lesznek a csomópontok és az őket összekötő asszociációk reprezentálják a képernyők egymásutániságát.

### 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 alrendszerek 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.

### 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:*

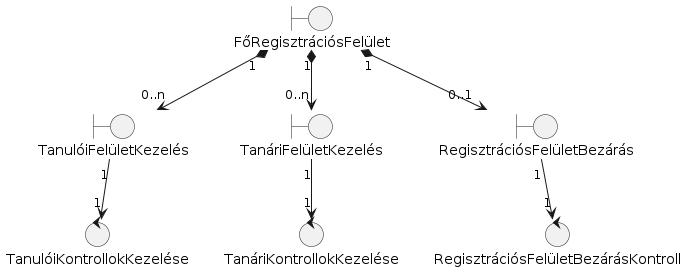
* Milyen szolgáltatásokra lenne a felhasználónak szüksége?
* Milyen szolgáltatásokat tud a rendszer adni?
* A fentiek közül milyen szolgáltatásokat kellene megvalósítani?

Storyboard példa:

A használati eset kezeli a bejövő üzeneteket, mielőtt használhatósági szempontokkal kiegészülne.

1. A használati eset akkor indul, amikor a levelező user kérést küld, hogy üzeneteket kezelhessen, és a rendszer megjeleníti azokat.
2. A levelező user ezután az alábbiakat hajthatja végre:
   1. Rendezheti az üzeneteket küldő vagy tárgy szerint.
   2. Elolvashatja az üzeneteket.
   3. Fájlként mentheti az üzenetet.
   4. Az üzenet csatolmányát mentheti fájlként.
3. A használati eset véget ér, amikor a levelező user kérést indít a bejövő üzenetkezelőből.

Példa UML ábra:



PlantUML szkriptje:

@startuml

boundary FőRegisztrációsFelület as B1

boundary TanulóiFelületKezelés as B2

boundary TanáriFelületKezelés as B3

boundary RegisztrációsFelületBezárás as B4

control TanulóiKontrollokKezelése as C1

control TanáriKontrollokKezelése as C2

control RegisztrációsFelületBezárásKontroll as C3

B1 "1" \*--> "0..n" B2

B1 "1" \*--> "0..n" B3

B1 "1" \*--> "0..1" B4

B2 "1" --> "1" C1

B3 "1" --> "1" C2

B4 "1" --> "1" C3

@enduml

### 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

### Fizikai rendszerterv

A fizikai rendszerterv a következő főbb részekből áll:

* osztálytervek
* adatbázisterv
* teszttervek
* telepítési terv
* rendszerspecifikációk (fejlesztési, futtatási környezet)
* szoftverarchitektúra
* az alkalmazás rétegei
* adatspecifikációk/objektumspecifikációk (környezetfüggő adattervek)
* programspecifikációk (modulvázak)

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a rendszerterv fontosabb részeit és az elkészítésének módszereit. A rendszerterv alapja a funkcionális specifikáció. A rendszerterv írja le, hogy hogyan kell a tervezett funkciókat megvalósítani. A logikai rendszerterv csak az üzleti logika megszorításait veszi figyelembe, ezzel szemben a fizikai rendszerterv a fizikai, pénzügyi megszorításokat is. A határosztály-tervezés és a menühierarchia-tervezés segít abban, hogy meghatározzuk az adatbeviteli felületeket és a menüstruktúrát. A Storyboard-modellezés felsőbb szintű nézetet ad arról, hogy melyik ablakból milyen szolgáltatások érhetők el. Ezekkel a módszerekkel finomítjuk tovább a képernyőterveket, a felhasználói sztorikat. A logikai rendszerterv részletezi az üzleti folyamatok modelljét, a követelményeket a logikai megszorítások szempontjából vizsgálja, leír egy ideális megoldást, mintha végtelen erőforrásaink lennének. A fizikai rendszerterv az osztályterveket, az adatbázistervet, a tesztterveket és a telepítési terveket foglalja magában, annak figyelembevételével, hogy erőforrásaink korántsem végtelenek. Mindkét terv célja a rendszer megvalósításának részletes leírása és specifikálása.

Ellenőrző kérdések:

1. Kérdés: Mi a rendszerterv célja?

Válasz: A rendszerterv egy írásban rögzített specifikáció, amely leírja, mit, miért, hogyan, mikor és miből akarunk a jövőben létrehozni.

1. Kérdés: Milyen fajta rendszertervet különböztetünk meg?

Válasz: Három fajta rendszertervet különböztetünk meg: konceptuális, nagyvonalú és részletes.

1. Kérdés: Hogyan különbözik egymástól a konceptuális és a nagyvonalú rendszerterv?

Válasz: A konceptuális rendszerterv röviden írja le, mit és miért akarunk létrehozni, míg a nagyvonalú rendszerterv ezeken túl kiegészül a hogyan és miből részekkel is.

1. Kérdés: Mit tartalmaz a részletes rendszerterv?

Válasz: A részletes rendszerterv a mit, miért, hogyan, miből és mikor kérdésre is válaszol, azaz tartalmazza a lépések idejét is.

1. Kérdés: Mi a legfontosabb része a nagyvonalú rendszertervnek?

Válasz: A nagyvonalú rendszerterv legfontosabb része az úgynevezett megfeleltetés, ami megmutatja, hogy minden megvalósítandó követelményhez van-e azt kielégítő lépés.

1. Kérdés: Mi a célja a határosztály-tervezésnek?

Válasz: A határosztály-tervezés célja az adatbeviteli felületek meghatározása és a kapcsolat létrehozása az adatmodell entitásosztályai és az adatbeviteli felületek között.

1. Kérdés: Hogyan ábrázolhatjuk a határosztályok közötti függőségeket?

Válasz: A határosztályok közötti függőségeket kollaborációs diagramokkal mutathatjuk meg.

1. Kérdés: Milyen lépéseket foglal magában a menühierarchia-tervezés?

Válasz: A menühierarchia-tervezés három lépésből áll: az alkalmazás fő menüstruktúrájának elemzése, a logikai alrendszerek menüstruktúrájának meghatározása, majd a menüelemek összekapcsolása a határosztályokkal egy osztálydiagramon.

1. Kérdés: Mi a Storyboard-modellezés célja?

Válasz: A Storyboard-modellezés célja, hogy felsőbb szintű nézetet adjon arról, hogy melyik ablakból milyen szolgáltatások érhetők el.

1. Kérdés: Melyek a logikai rendszerterv főbb részei?

Válasz: A logikai rendszerterv főbb részei az üzleti folyamatok modellje, a követelmények, a feldolgozási folyamatok, a funkcionális felépítés, a felhasználói felületek, menük, az adatszótár, logikai adatmodell, és az adatfolyam-diagramok.

1. Kérdés: Melyek a fizikai rendszerterv főbb részei?

Válasz: A fizikai rendszerterv főbb részei az osztálytervek, az adatbázisterv, a teszttervek, a telepítési terv, a rendszerspecifikációk, a szoftverarchitektúra, az alkalmazás rétegei, az adatspecifikációk/objektumspecifikációk és a programspecifikációk.

# Principles: Az objektumorientált tervezés alapelvei

## Absztrakt alapelvek

Mielőtt rátérünk az objektumorientált tervezés alapelveire, érdemes megvizsgálni azokat az absztrakt alapelveket, amiből minden más általunk tárgyalt alapelv levezethető. Absztrakt alapelvek:

* A program kódja állandóan változik.
* Amit szét lehet választani, azt érdemes szétválasztani.
* A szoftver is csak egy termék, olyan, mint egy doboz müzli.
* A szoftverfejlesztésnek nincs Szent Grálja.

### 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-fő elvünk, amiből közvetve, vagy közvetlenül minden más elv következik.

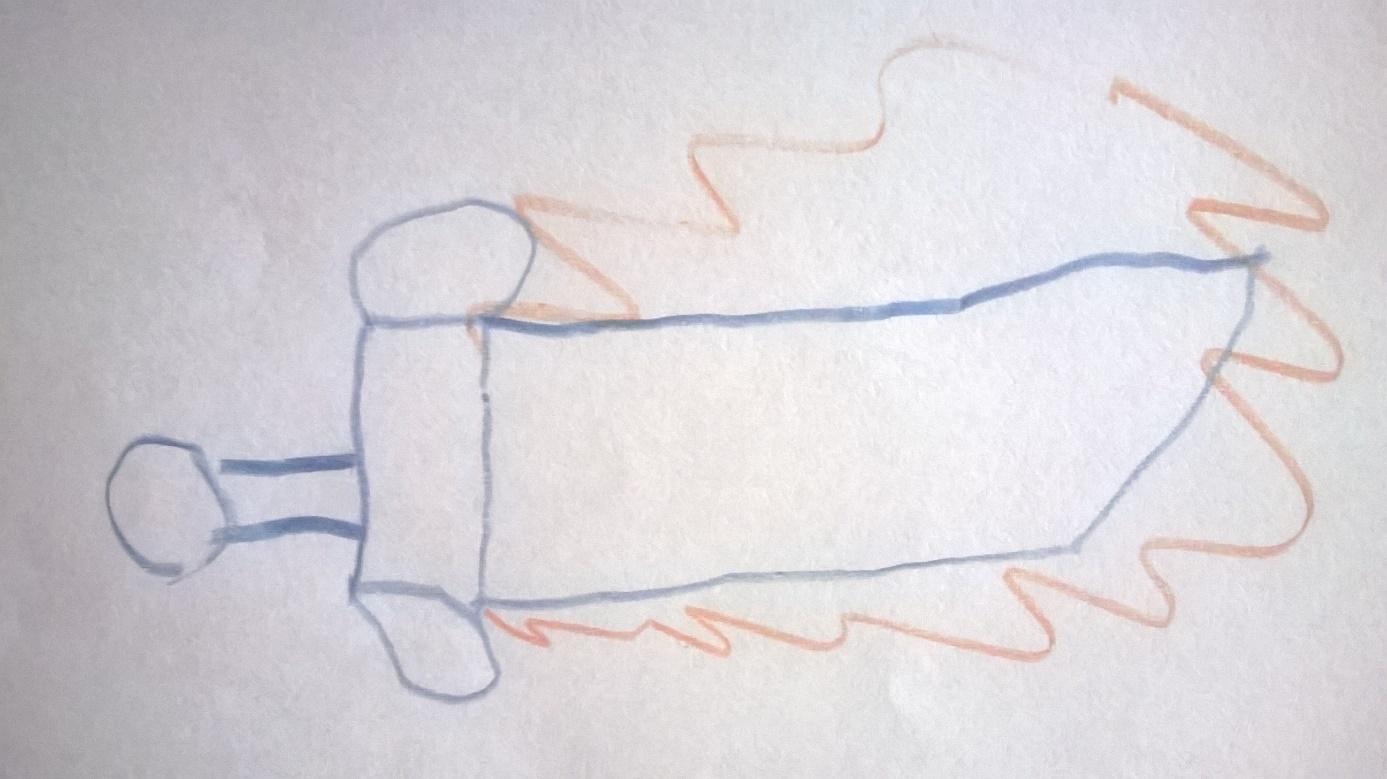
### Ügyek szétválasztása – Separation of Concerns

Az előző elv írja le, hogy mi a legnagyobb probléma a szoftverfejlesztés során, amivel minden programozónak és tervezőnek szembe kell nézni: 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 kezelni.

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 forráskódot, a terveket.



Az ügyek szétválasztása elvet egy lángoló kardként képzelhetjük el

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

A következő 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.

### A szoftverfejlesztésnek nincs Szent Grálja

A következő 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 szoftverkrízisről szóló részeket). Ugyanakkor a modern módszertanok, jól bevált módszerek, tervezési eszközök és újabban a mesterséges intelligencia kombinációja nagyságrendekkel jobb és olcsóbb szoftverfejlesztést tesz lehetővé.

### A terv legyen a lehető legegyszerűbb, és csak annyira bonyolult, amennyire muszáj

A tervezés egyik legnépszerűbb alapelve a KISS:

* Maradjon ostobán egyszerű! Angolul: Keep It Simple, Stupid!

Ez azt jelenti, hogy a terv akkor jó, ha már nem lehet tovább egyszerűsíteni, a funkciók letisztultak, még egy ostoba is tudja használni. Ennek az elvnek egy másik változata a YAGNI elv:

* Úgyse lesz rá szükség! Angolul: You Aren't Gonna Need It!

Egy funkciót, egy osztályt, egy plusz absztrakciós szintet csak akkor adunk hozzá a tervhez, az elkészült termékhez, ha biztosan szükség van rá. Ez az extrém programozás egyik alapelve.

Forráskód szintjén a DRY alapelvet szoktuk használni:

* Ne ismételd önmagad! Angolul: Don’t Repeat Yourself!

Azaz a forráskódban ne legyenek ismétlődő részletek, hanem azokat emeljük ki egy metódusba, egy másik osztályba, vagy akár egy plusz absztrakciós szintbe. E miatt a DRY alapelv ellene hat a KISS alapelvnek: néha muszáj plusz absztrakciós szintet bevezetni. Ha nem tartjuk be a DRY elvet, akkor azt szoktuk mondani, hogy a forráskód nem elég száraz. A KISS és a DRY elveket együtt érhetjük tetten az egy ismert mérnöki alapelvben:

* Olyan egyszerűen, amennyire csak lehet, és csak annyira bonyolultan, amennyire muszáj. Angolul: As simple as possible, as complicated as necessary.

Azaz, **a terv legyen a lehető legegyszerűbb, és csak annyira bonyolult amennyire muszáj**. Tehát, a KISS mellett muszáj a DRY elvet is követni.

### További elvek

A jegyzet további elveket is tartalmaz, ezek:

* az objektumorientált programozás alapelvei,
* az objektumorientált tervezés alapelvei.

Az objektumorientált programozás alapelvei:

* Egységbezárás (angolul: encapsulation)
* Öröklődés (angolul: inheritance)
* Többalakúság (angolul: polymorphism)

Ebben a könyvben a debreceni és az egri programozó képzés hagyományait követve a fenti 3 objektumorientált programozási alapelvet használjuk. Ezeket részletesen kifejtjük a későbbi fejezetekben.

Ugyanakkor meg kell jegyezni, hogy az angol nyelvű irodalom általában 4 objektumorientált programozás alapelvet használ. A fenti hármat kiegészítik az absztrakció elvével. Illetve sok forrás az objektumorientált 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 objektumorientált programozási alapelvnek 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 alapelvgyűjtemények 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 Substitution 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. 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 el 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 potenciálját hivatott tesztelni:

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 segédeszközöket 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, amelyek csak egy dolgot csinálnak, de azt jól csinálják”. 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.”

## Az objektumorientált programozás alapelvei

A szoftverkrízisre 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 absztrakciója. Hogy ez a darabka kicsi vagy nagy, az az osztály felbontását (más szavakkal: granularitását, szemcsézettségét, durvaságát, angol szóval: granularity) adja 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 viszont nem.

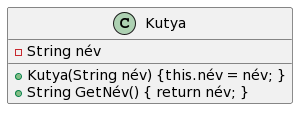
A rekord az eljárásorientált nyelvek (vagy más néven az imperatív nyelvek) kedvenc típusa. Az eljárások rekordokon dolgoznak. Az OOP is az imperatív programozási nyelvek családjába tartozik, csak itt már a rekordokat és a rekordokon dolgozó eljárásokat egybeolvasztjuk, méghozzá osztályokba. Azt mondjuk, hogy az adatokat és a rajtuk végrehajtott műveleteket egységbe 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

A Kutya osztály UML ábrája:

[](https://www.plantuml.com/plantuml/img/SoWkIImgAStDuKhEIImkLl0kBQbChERI3IufoinBLyWxlBACIniig84afAbGNPAHMQm7OYlOWeMi5MgvjA5At5DB_8126a1b2aMf9QL5UN0bjLmEgNafGFi0)

PlantUML szkriptje:

| @startuml class Kutya{ -String név +Kutya(String név) {this.név = név; } +String GetNév() { return név; } } @enduml |
| --- |

Itt jegyezzük meg, hogy a jegyzetben egy kicsit szabadon használjuk az UML osztálydiagramját. Az UML nyelvi definíciója szerint a visszatérési típust nem a függvény neve elé kellene írni, hanem mögé. A kapcsos zárójel pedig nem a függvény törzsének helye, hanem megszorításokat lehet közé írni. Mégis, mi ezt a C alapú nyelvekhez közel lévő, fenti leírást részesítjük előnyben. Ezt azért tehetjük meg, mert az évek során kiderült, hogy az UML csak két dologra jó, az ötletek gyors felrajzolására (skiccelésre), és a kiforrott megoldások dokumentálására. Tehát fontos, hogy egy UML ábra olvasmányos legyen, kényelmes egy Java vagy C# nyelvhez szokott szem számára.

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 konstruktorában adhatjuk meg, amikor példányosítjuk.

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

* felülete és
* megvalósítása (idegen szóval: implementációja, angolul: implementation).

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. Általánosan: Ami publikus, az az osztály felületéhez tartozik. 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 megvalósítását a metódusainak (nem csak a publikus, hanem az összes metódusának) a törzse adja meg. Például a GetNév metódus implementációja az, hogy visszaadja a név mező értékét. Habár ez a szokásos kódja a GetNév-nek, más megvalósítást 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 osztályénál. Ilyen nyelv pl. a Smalltalk.

Az objektum belső állapotát mezőinek pillanatnyi értéke határozza meg. Mivel az osztály metódusai megváltoztathatják 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.

Az objektum viselkedését a metódusainak implementációja adja. Az objektum viselkedése függ a belső állapottól. Ha van egy if utasítás a forráskódban, akkor már kétféle viselkedés lehet. Hogy melyik fog futni, az a mezők értékétől (belső állapot), és a paraméterek értékétől (külső állapot) függ.

Azt szoktuk mondani, hogy az osztály megvalósítása az maga a forráskód, vagyis a forráskód statikus vetülete. Az objektum viselkedése pedig a forráskód dinamikus vetülete, azaz, a futó forráskód. A viselkedés arra a kérdésre ad választ, hogy a forráskód mely ága hajtódik végre. Ez utóbbi nyilván a mezők értékétől, azaz az objektum belső állapotától, és a futó metódus paramétereitől, azaz a külső állapottól függ.

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

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

### Egységbezárás – Encapsulation

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

### Öröklődés – Inheritance

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

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

### Többalakúság – Polymorphism

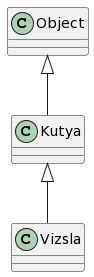
A jegyzetben ismertetett tervezési alapelvek és tervezési minták majd mindegyike a többalakúságon (angolul: polymorphism) alapszik. Tehát ez egy nagyon fontos alapelv. Maga a többalakúság az öröklődés következménye. Mivel a gyermekosztály örökli az ős felületét, ezért a gyermekosztály példányai megkapják 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.

Nézzük meg az Object - Kutya - Vizsla öröklődési hierarchia UML diagramját:



PlantUML scriptje:

| @startuml  Object <|-- Kutya  Kutya <|-- Vizsla  @enduml |
| --- |

Egy osztály példányai az öröklődési hierarchián felfelé haladva rendelkeznek az összes típussal. Ennek megfelelően minden objektum Object típusú is, hiszen ha nem adjuk meg egy osztály ősét, akkor az az Object osztályból származik. Akárhogy is, minden osztályhierarchia tetején, Java és C# nyelv esetén is, az Object osztály van.

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.

## Az OOP hasznos megoldásai

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

* Automatikus szemétgyűjtés (angolul: garbage collection, angol rövidítése: GC),
* Mező, mint lokális-globális változó,
* Többalakúság használata osztály behelyettesítésre,
* Csatoltság csökkentése objektum-összetétellel.

### Automatikus szemétgyűjtés

Az automatikus szemétgyűjtés 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, vagy mást szabadít 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.

Ahhoz, hogy megértsük az automatikus szemétgyűjtést, nézzük át mit is jelent a memória foglalás, illetve mi is az a memória szoftver szemszögből nézve.

A processzor az adatokat és metódusokat a véletlen hozzáférésű tárban (angolul: Random Access Memory - RAM) helyezi el. Véletlen a hozzáférése mert bármelyik adatát el tudjuk érni, mégpedig annak memória címével. Ez a cím a legtöbb magasszintű programnyelvben nem érhető el, mivel nem közvetlenül alacsony szinten kezeli a címeket a kódunkat futtató virtuális gép, hanem a számára biztosított memória területet felosztja a metódusok és adatok tárolására alkalmas területekre. Az adatokat a halom (angolul: heap) és a verem (angolul: stack) fogja tárolni. A heap-ben az osztályok, a példányaik és a példány változók tárolódnak, míg a stack a metódusok lokális változóit tartalmazza. Ebből is látható, hogy a stack addig létezik, míg a metódus, amihez tartozik fut, ezzel szemben a heap az független a metódusok életciklusától, így az objektumok léte nem közvetlen függ az azokat használó metódusoktól.

De hogyan is valósul meg mindezek használata a programunk futtatása során? Nézzünk erre egy példát! Tételezzük fel, hogy van egy olyan Java program, amiben van egy Circle osztály, az pedig rendelkezik egy radius változóval és ki tudja számolni a saját területét. Továbbá rendelkezik három metódussal. Az m1() csak saját lokális változókat használ, de meghívja az m2() metódust. Az m2() rendelkezik bemenő paraméterrel és egy saját lokális változóval is, továbbá meghívja az m3() metódust. Végezetül az m3() létrehoz egy Circle példányt és kiírja annak területét. Java programunk belépési pontja a main(), ami csak az m1() metódust fogja meghívni.

public class Main {

public static void main(String[] args) {

m1();

}

static void m1(){

int x = 0;

m2(x);

}

static void m2 (int b){

int a = 0;

m3();

}

static void m3(){

Circle circle = new Circle(10);

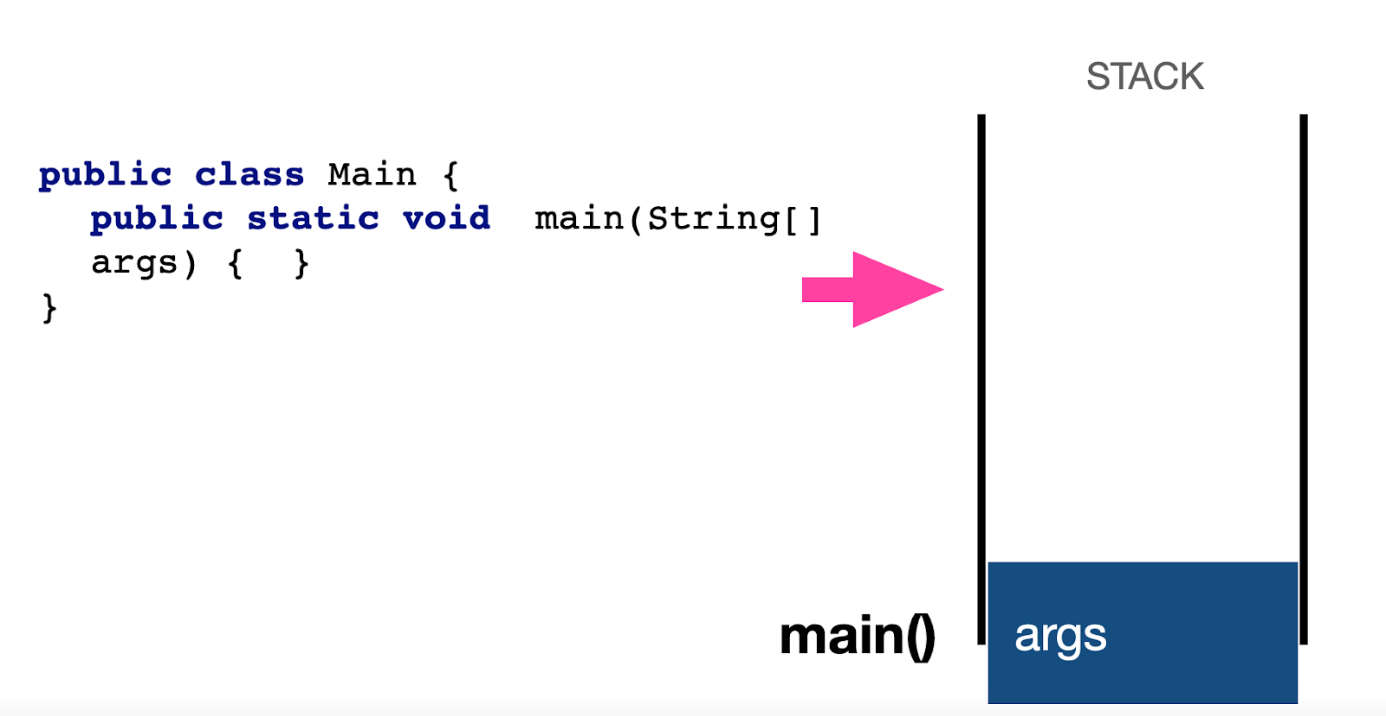
System.out.println(circle.area());

}

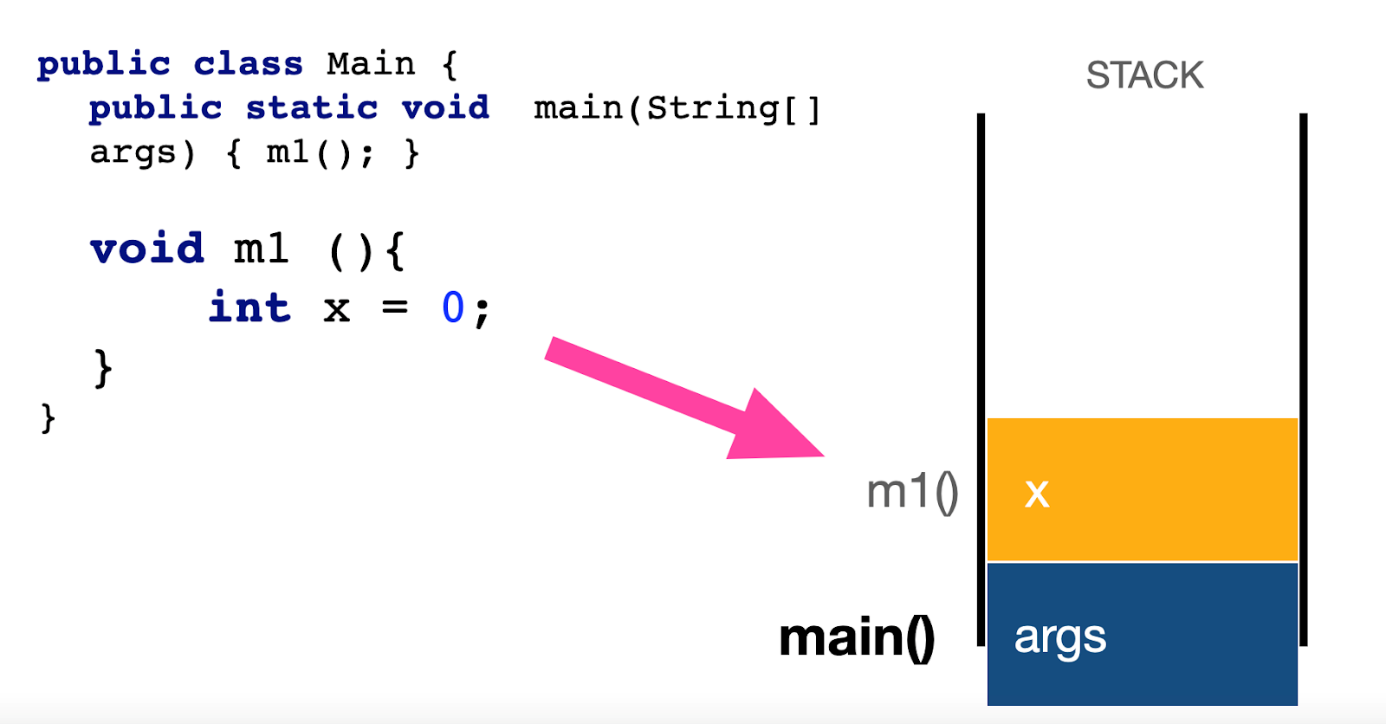
}

A memóriafoglalás folyamatának megértését kezdjük az elején, a main() metódusnál!

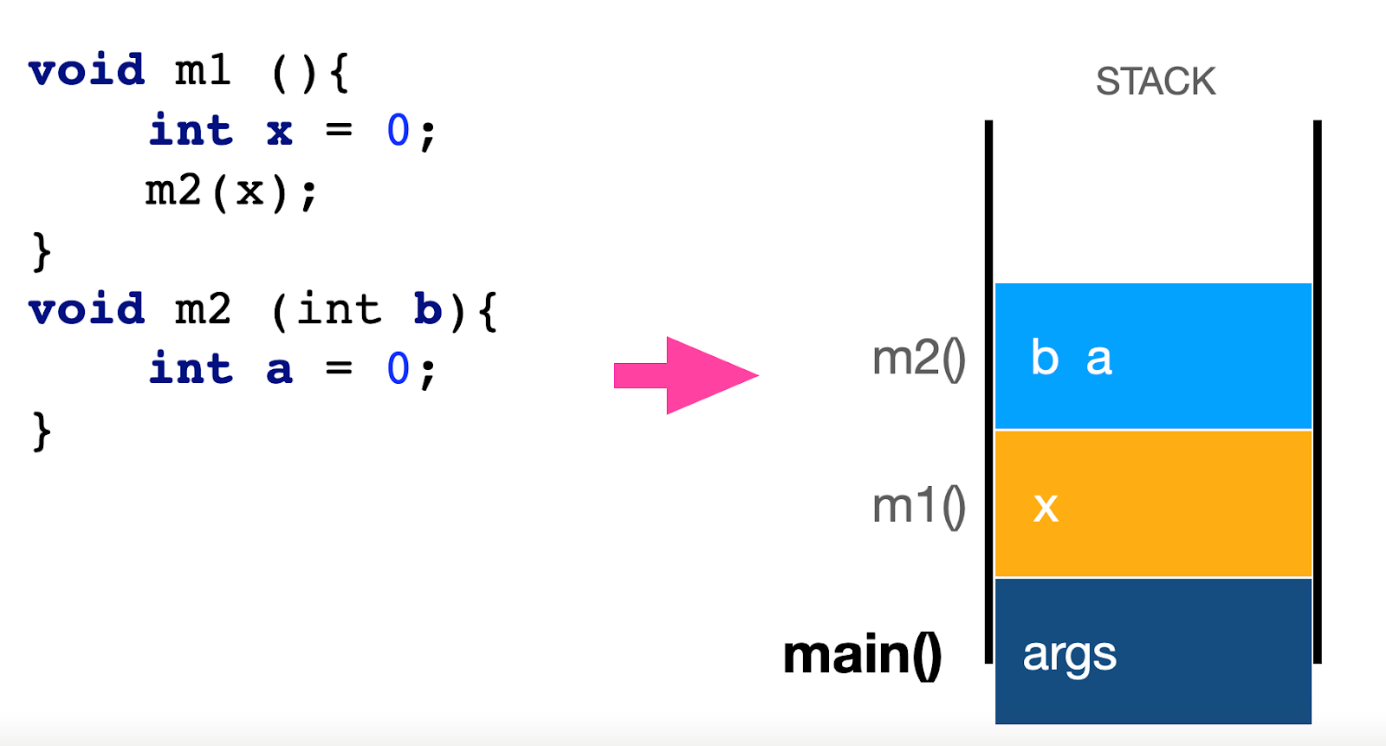
Mikor a program végrehajtása elér ide, akkor a main() metódus számára létrejön egy lefoglalt terület a stack-en és ebbe bekerül az args változó és a hozzá tartozó adat.



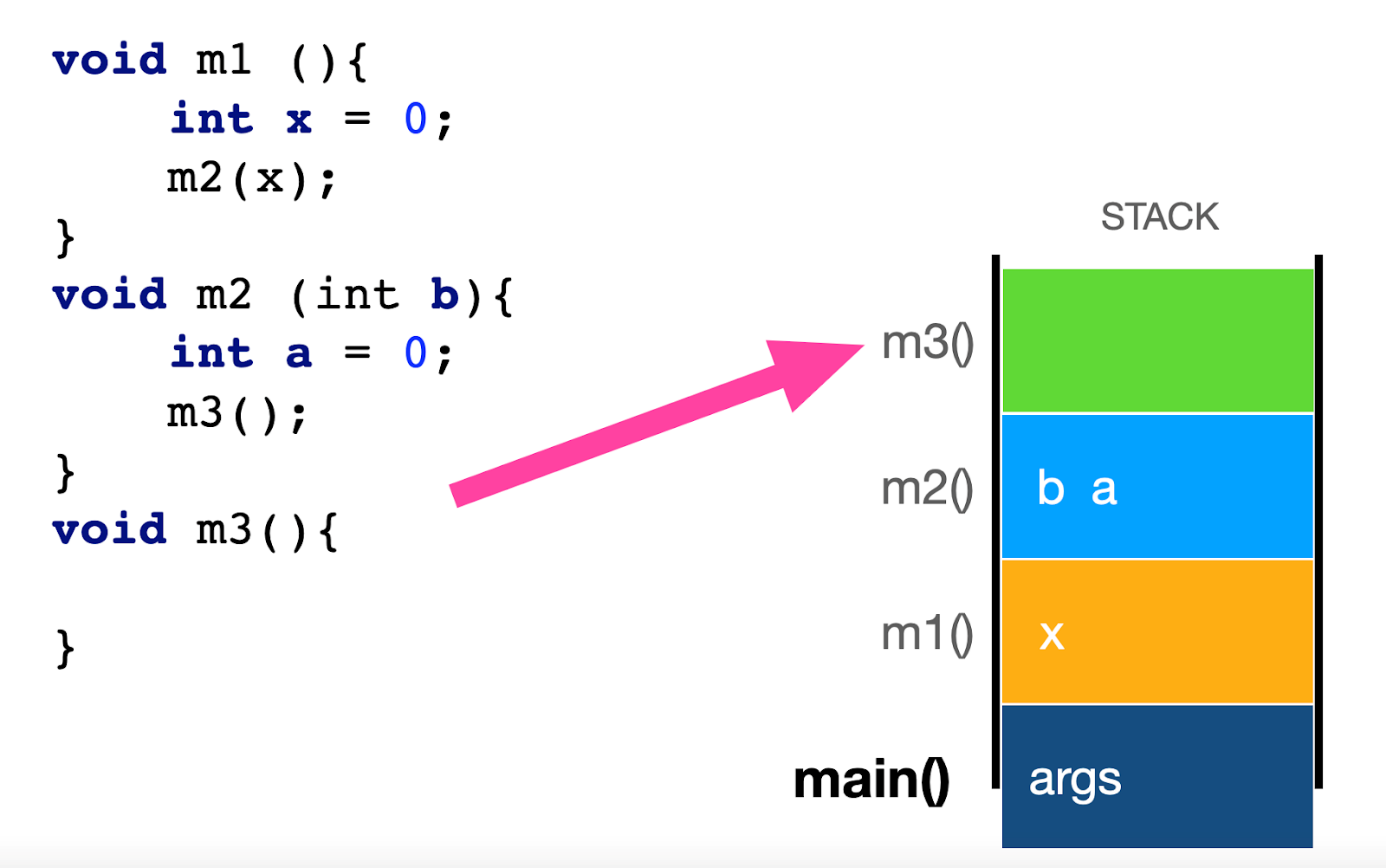
Folytatódik a main() feldolgozása és mikor elér oda, hogy meghívja az m1() metódust, akkor annak is lefoglalásra kerül egy stack terület az x változójával együtt.



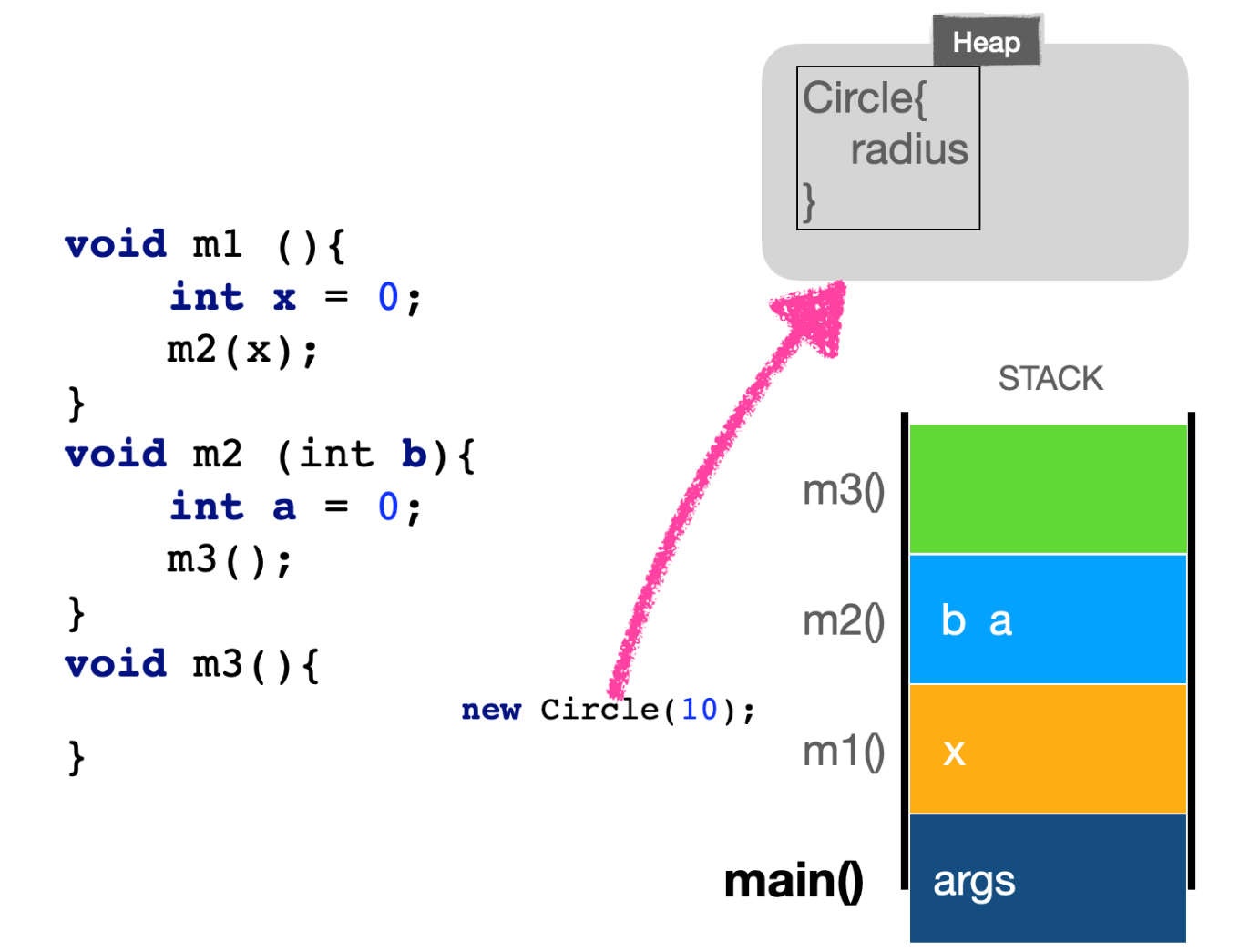
Mivel az m1() meghívja az m2() metódust, így annak is lefoglalásra kerül egy terület a stack-en és ebbe két változó (b  és a) tárolódik el.



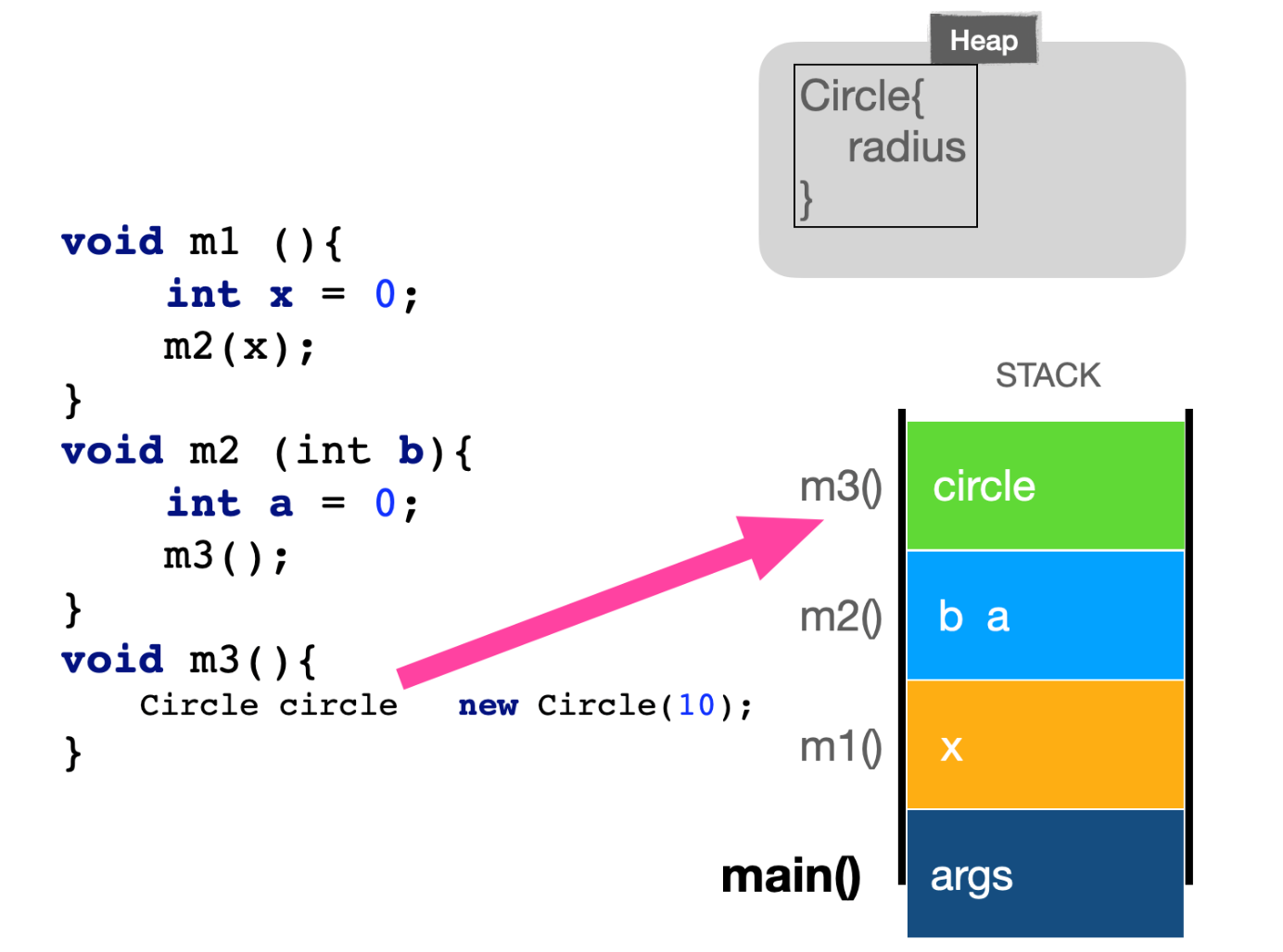
Amint az m2() meghívja az m3() metódust, természetesen ahhoz is létrejön a stack-en egy tárterület.



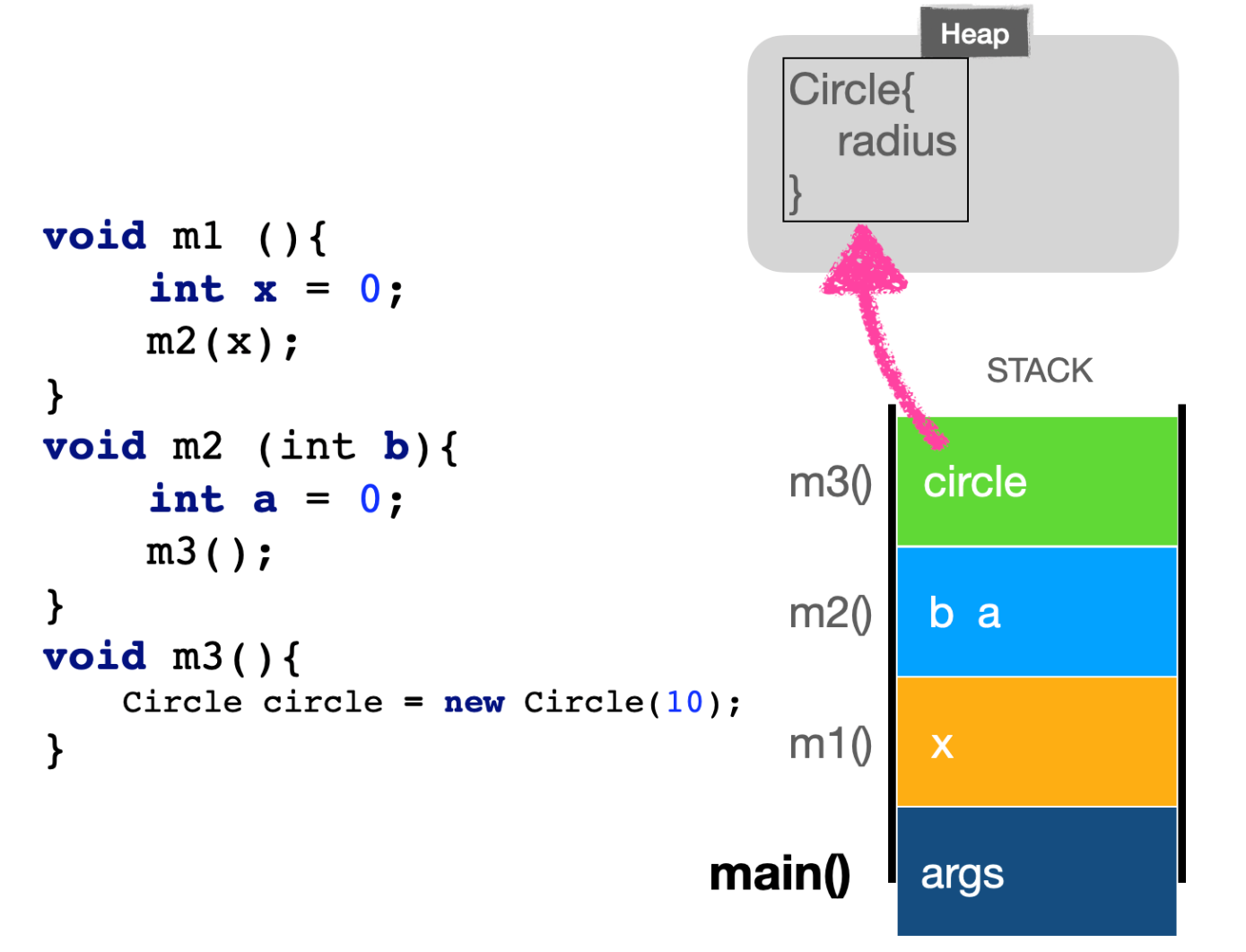
Az m3() egyetlen circle lokális változóval rendelkezik. Viszont ennek példányosítási folyamata el fog térni az eddigiektől, ugyanis lefoglalásra kerül a new kulcsszó hatására a Circle osztály definíción alapuló példány a heap-en. Erre hivatkozás, vagyis referencia még nem létezik, tehát hivatkozni rá nem tudunk az m3() metódusból.



Ahhoz, hogy el tudjuk érni a heap-en létrejött példányt, a stack-en kell egy referencia rá, ez lesz a circle változó, ami egy Circle típusú objektumra fog mutatni.

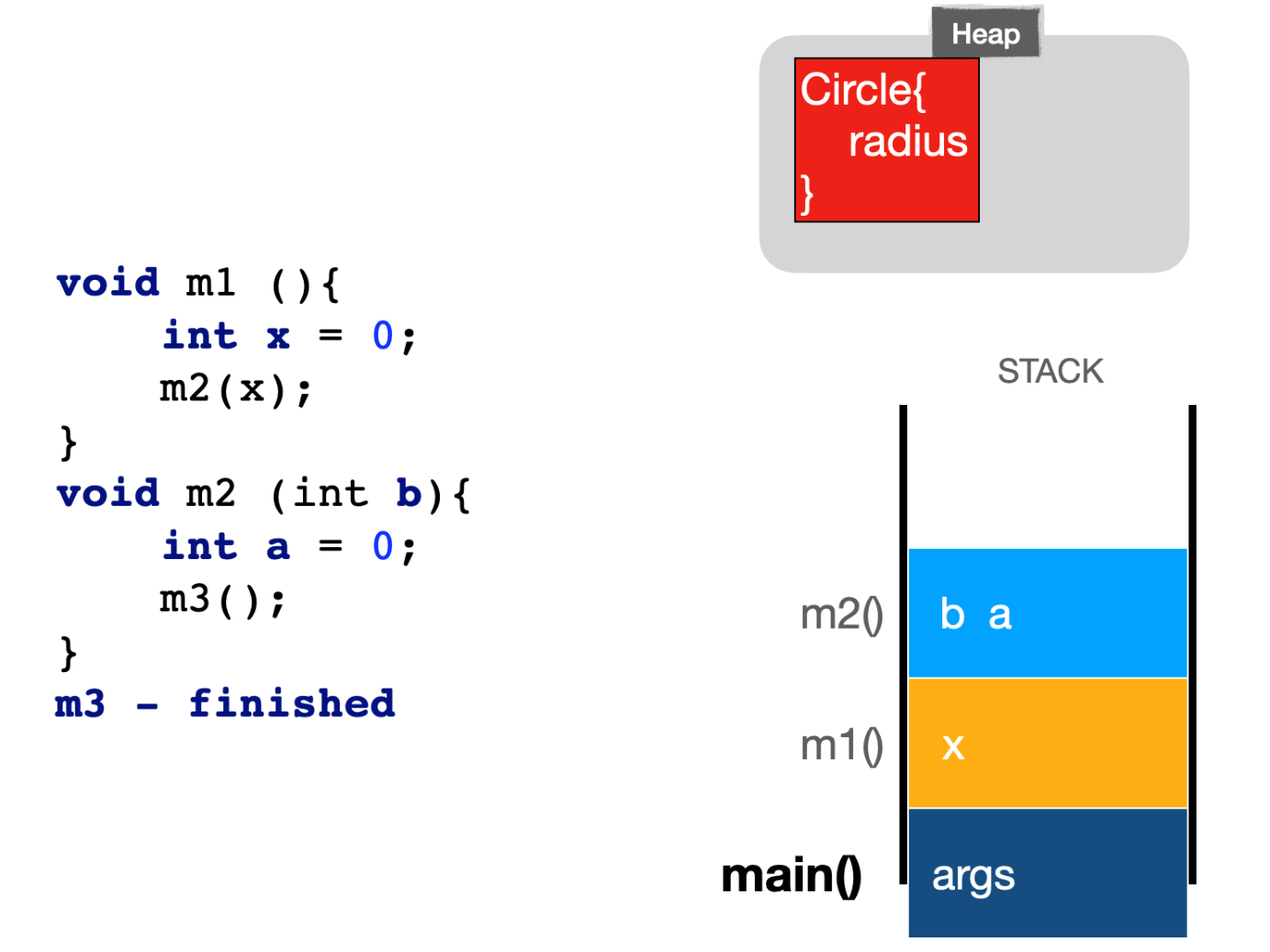


Most már rendelkezünk referenciával és objektummal is, de a kettő még nincs összerendelve. Ezt tesszük meg mikor az egyenlőség jellel összekötjük őket.



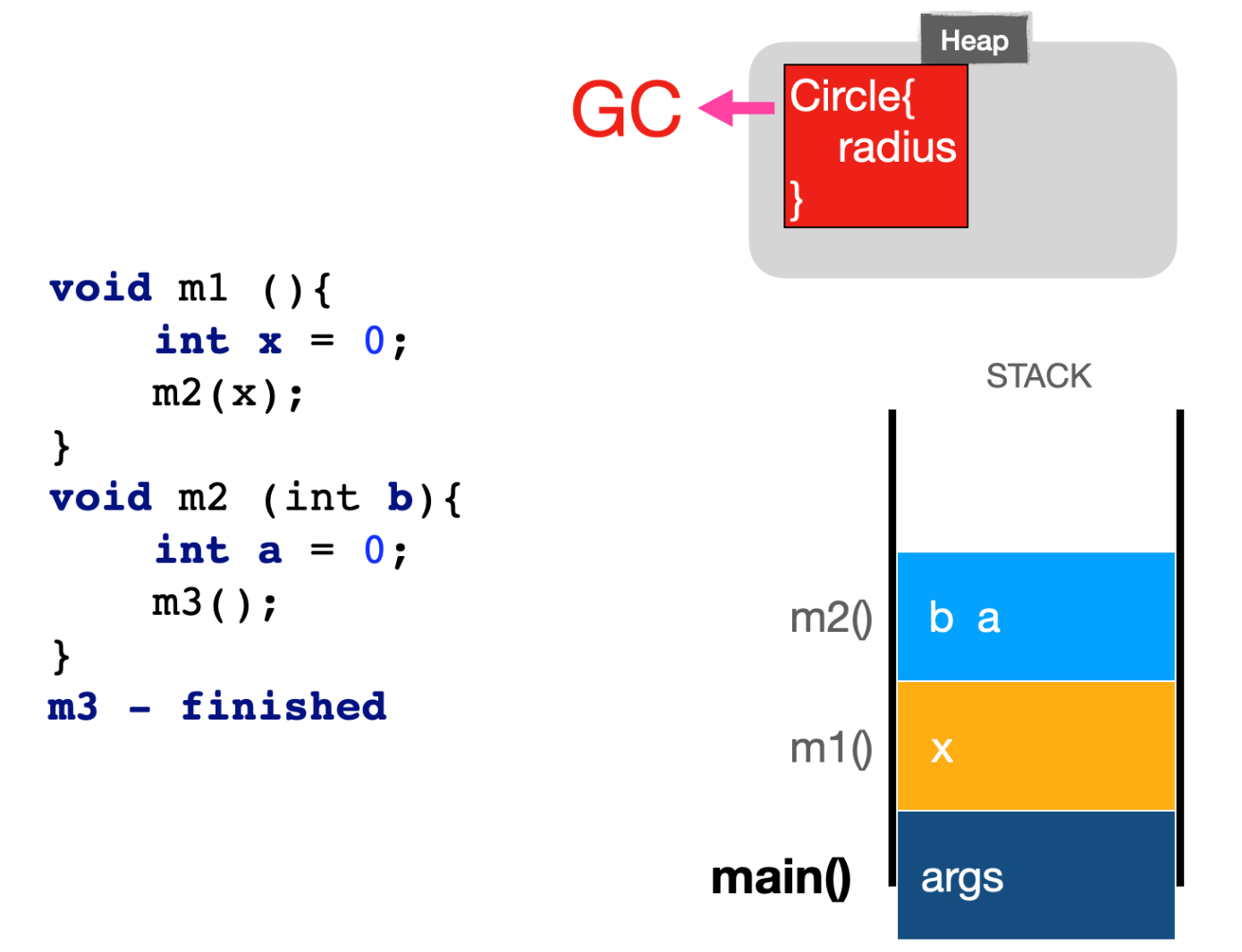
Összefoglalva, a Circle circle = new Circle(10); sor mind a heap, mind a stack területen lefoglal az m3() metódus számára helyet, és a circle lesz az a referencia amivel az m3()-ból hivatkozni tudunk a heap-en tárolt adatokra.

Miért jó ez? A válasz a stack és heap közti különbségen alapul.  Ugyanis amikor az m3() lefut, akkor megszűnik és vele együtt a stack-en tárolt területe is felszabadul.

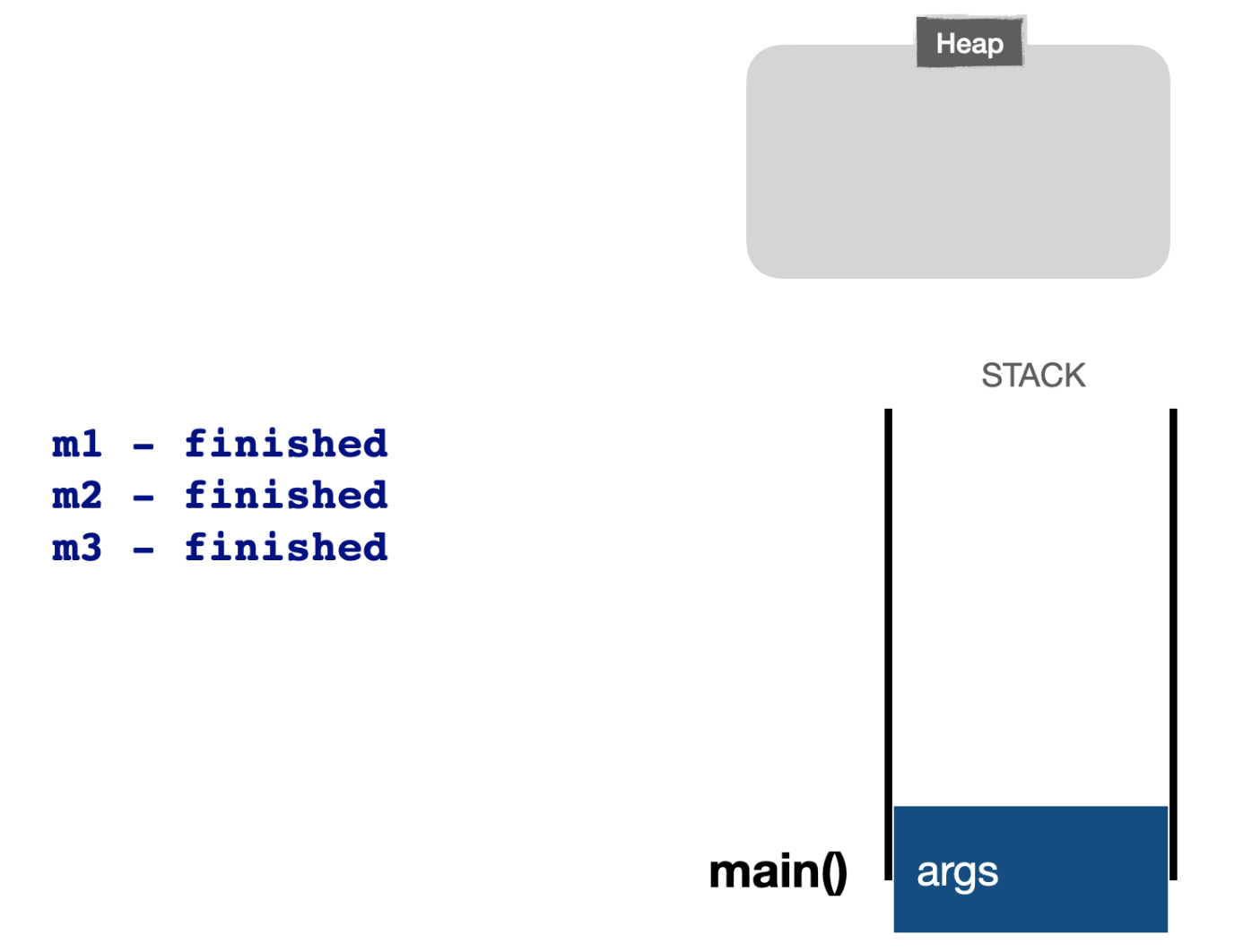


De a heap-en tárolt példányra más referencia még hivatkozhat, annak nem kell megszűnnie. Lényegében csak a circle lokális változó törlődik automatikusan.

Természetesen amennyiben a Circle példányt nem használja más, akkor idővel a Garbage Collector ezt felfedezi és fel fogja szabadítani azt.



Végezetül pedig ahogy az m2() és m1() is lefut, azok stack-en foglalt területei is felszabadulnak, és ha a main()-ben sincs más, ami fenntarthatná, annak tárhelye is törlődik, végül a program leáll.



További érdekességek ebben a témában: Java Stack and Heap - Java Memory Management, link: <https://www.youtube.com/watch?v=450maTzSIvA>.

### 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 egységbezárás, illetve azon belül az információrejtés miatt a publikus mezőket természetellenesnek érezzük, így senki sem használ már globális változókat OOP nyelveken.

### 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 gyermekosztályú példány használható mindenütt, ahol ősosztály típusú paramétert várunk. Ez a többalakúság lényege.

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

Hol van itt a többalakúság, hiszen eddig szinte csak öröklődésről beszéltünk? Helyes megfigyelés, hiszen többalakúság nincs öröklődés nélkül. A gyermekosztály 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 (ősosztályú példányt várunk, de gyermeket kapunk),
* Példányosítás (a referencia ősosztály típusú, de egy gyermek példányra mutat),
* Felelősséginjektálás (kívülről kapunk egy objektumot, aminek csak a felületét ismerjük).

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

### Csatoltság csökkentése objektum-összetétellel

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 leszármazottja B-nek, vagy implementálja B-t.

A csatoltság szintjei (legerősebbtől a leggyengébbig):

* erősen csatolt (angolul: tightly coupled)
* gyengén / lazán csatolt (angolul: loosely coupled)
* réteg (angolul: layer)

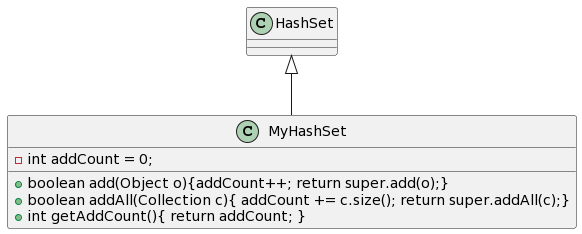
Az erős csatoltság erős függőséget is jelent. A következő fajta függőségeket szoktuk megkülönböztetni:

* 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; } } |
| --- |

A MyHashSet UML ábrája:

[](https://www.plantuml.com/plantuml/img/PP0z3i8m38NtdC9ZewWLUwZKgWiBOk04QMf1KKXGVWOekJi9IbKXDjjwtxEVE-U5zU6kY5J2EJW8TpsZNsBfZyzrGkfPUn3JD9YGYXPsdB3H68L2vt5r6cyeFHYwh2B6E5ZqmMfmuO6soJ93UVm1UwMgmIYLuDbea7JPjh0MPEFc5rRqtog3ykkMBxkWxmjM9OiYNfquH199oGBxTrrlqKY7Uael-000)

PlantUML szkriptje:

| @startuml  class HashSet{}  class MyHashSet{  -int addCount = 0;  +boolean add(Object o){addCount++; return super.add(o);}  +boolean addAll(Collection c){ addCount += c.size(); return super.addAll(c);}  +int getAddCount(){ return addCount; }  }  HashSet <|-- MyHashSet  @enduml |
| --- |

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 eggyel 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());  } } |
| --- |

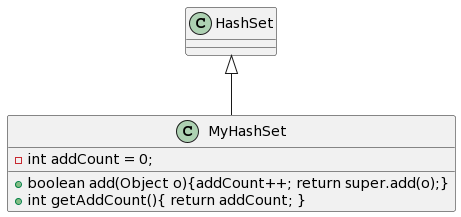
Ebben a példában létrehoztunk egy 3 elemű tömböt, azt addAll metódussal hozzáadtuk a MyHashSet egyik példányához. Ezután kiíratjuk, hány elemet adtunk hozzá a halmazhoz. Azt várnánk, hogy a program azt írja ki, hogy 3, de ehelyett az írja, hogy 6.

Mi történt? Nem tudtuk, hogy az ősben, 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 hívtuk, az hozzáadott 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 mindig 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; } } |
| --- |

A MyHashSet UML ábrája:

[](https://www.plantuml.com/plantuml/img/HOwz3W8X48JxVOejZn2CFPgSiR4n5Zu1TspyoGaNM0g3lBkIGEns9j_CpXZP14wl1UR5n8Wd4-zNuboQFh-x0-hX68srHvz-nnwt6kJa_KB6LNkuJ4-Q6Rt87P9IOo1EmM5CAuLDnRpG1MJjkX4V6Zc8tCaUrbYWGFkEkuzI_p4mahDrz1S0)

PlantUML szkriptje:

| @startuml  class HashSet{}  class MyHashSet{  -int addCount = 0;  +boolean add(Object o){addCount++; return super.add(o);}  +int getAddCount(){ return addCount; }  }  HashSet <|-- MyHashSet  @enduml |
| --- |

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 függőség!

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

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

| | class A {  public void M1() {  Console.Write("hello");  } } class B : A {    public void M2() { M1(); } } class Program {  static void Main(string[] a)  {  B b = new B();  b.M2();  Console.ReadLine();  } } | | --- | | | class A {  public void M1() {  Console.Write("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();  } } | | --- | |
| --- | --- | --- | --- |
| Itt a B osztály az A osztály gyermeke, így örökli az A osztályból az M1 metódust, amit M2 metódusban hív meg. A főprogramban meghívjuk az M2 metódust, amely meghívja az ősben 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. |
| | @startuml  class A{  +void M1()  }  class B{  +void M2()  }  A <|-- B  @enduml | | --- | | | @startuml  class A{  +void M1()  }  class B{  -A a  +void M2()  }  A <-down-o B: -a  @enduml | | --- | |

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

Amikor objektum-összetételnél egy metódust úgy valósítunk meg, hogy annak lényegi része csak az, hogy az összetételt megvalósító referencián keresztül meghívunk egy metódust, akkor azt mondjuk, hogy delegáljuk a felelősséget a beágyazott objektumnak. A fenti példában ilyen az M2 metódus, ami csak meghívja az M1 metódust. A felelősség delegálás fogalmát nem szabad összekeverni a C# nyelv delegate kulcsszavával, hiszen azzal függvénytípust hozunk létre, nem felelősség-átadásra való.

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

A későbbiekben látni fogjuk, hogy habár az öröklődést mindig ki lehet váltani objektum-összetétellel, nem mindig ez a célravezető, hiszen öröklődés nélkül nincs többalakúság. Többalakúság nélkül pedig nem lehet rugalmas kódot írni.

## Tervezési alapelvek

Az objektumorientált tervezés alapelvei (angolul: 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 érthető más programozók számára is.

A tervezési alapelvek segítenek, hogy ne essünk például abba a hibába, hogy egy osztályba kódolunk mindent, 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. Gondoljunk 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 rászánni a fejlesztésre és biztosítani, hogy a jövőben könnyű legyen a változások kezelése. Ezt biztosítja számunkra az alapelvek betartása.

## Lecke 7.: A GOF könyvei alapelvei

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a GOF könyv két tervezési alapelvével, a GOF1 és GOF2 alapelvekkel, így képesek leszünk ezeket a gyakorlatban alkalmazni.

Szükséges eszközök, források: A GOF könyv. A könyv magyar címe: „Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz.” A könyv angol címe: „Design Patterns: Elements of Reusable Object-Oriented Software”.

Feldolgozási idő: 70 perc.

Témakörök: GOF1, GOF2, objektum-összetétel, HAS-A kapcsolat, becsomagolás.

A lecke tartalma: Ebben a leckében áttekintjük a GOF könyv két alapelvét és gyakorlati alkalmazásukat. Először is, megvizsgáljuk a „Programozz felületre implementáció helyett” alapelvet (GOF1), amely arra ösztönzi a tervezőket, hogy az interfészre programozzanak, ne pedig az implementációra. A következő alapelv, a „Használj objektum-összetételt öröklés helyett, ha csak lehet” (GOF2) arra ösztönzi a tervezőket, hogy inkább objektum-összetételt alkalmazzanak az öröklés helyett. Az öröklés előnyeit és korlátait összehasonlítva láthatjuk, hogy az objektum-összetétel rugalmasabb és jobban támogatja a tervezési változásokat. A továbbiakban megvizsgáljuk az IS-A és HAS-A kapcsolatot. Az IS-A kapcsolat az öröklődést jelenti, míg a HAS-A kapcsolat az objektum-összetételt. Ezután részletesen foglalkozunk az aggregációval és a kompozícióval, két különböző objektum-összetétel típussal. Aggregáció esetén a rész létezhet az egész nélkül is. Kompozíció esetén a rész nem létezhet az egész nélkül. Végül, megvizsgáljuk az átlátszó és átlátszatlan becsomagolást, amik szintén az objektum-összetétel típusai.

### A GOF könyv 1. alapelve – GOF1

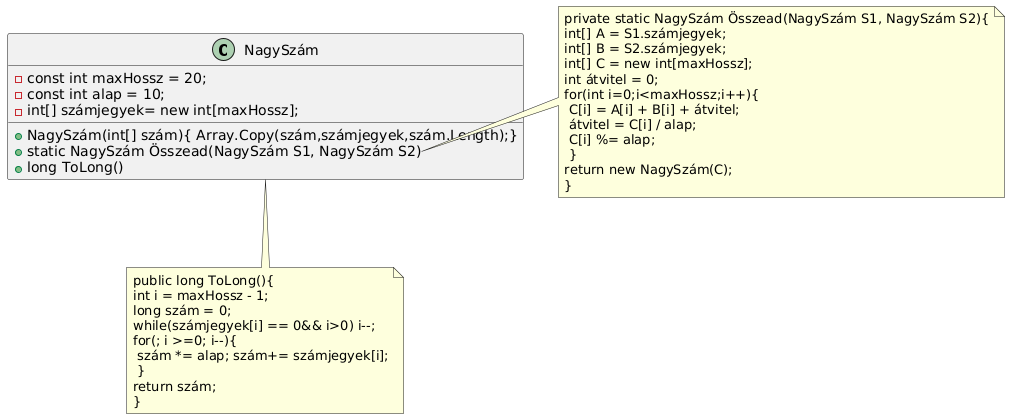
A GOF1 alapelv a Gang of Four (GOF) könyvben jelent meg 1995-ben. A könyv magyar címe: „Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz.” A könyv angol címe: „Design Patterns: Elements of Reusable Object-Oriented Software”. Az alapelv eredeti angol megfogalmazása: „Program to an interface, not an implementation”, azaz „Programozz 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 ős implementálva. Egy másik példa:

| class NagySzám {  *//maximum ennyi számjegyű nagy szám*  private const int maxHossz = 20;  *//használt számrendszer 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 NagySzám UML ábrája:



PlantUML szkriptje:

| @startuml  class NagySzám{  -const int maxHossz = 20;  -const int alap = 10;  -int[] számjegyek= new int[maxHossz];  +NagySzám(int[] szám){ ArrayAray.Copy(szám,számjegyek,szám.Length);}  +static NagySzám Összead(NagySzám S1, NagySzám S2)  +long ToLong()  }  note right of NagySzám::Összead  private 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);  }  endnote  note bottom of NagySzám  public long ToLong(){  int i = maxHossz - 1;  long szám = 0;  while(számjegyek[i] == 0&& i>0) i--;  for(; i >=0; i--){  szám \*= alap; szám+= számjegyekszámjegeyek[i];  }  return szám;  }  endnote  @enduml |
| --- |

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 konstruktor 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 számrendszert használunk, az elromlik. Például az alábbi is:

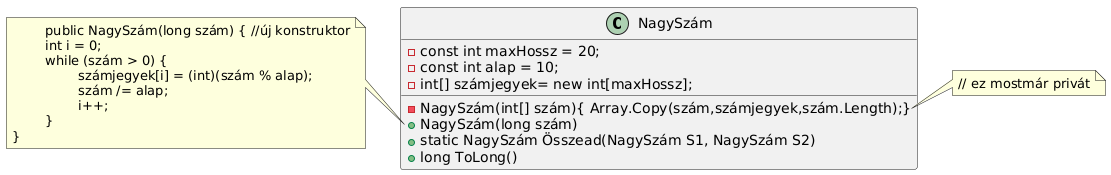
| class Program {  static void Main(string[] args) {  int[] a = { 3, 5 }; *//53*  int[] b = { 1, 2, 3 }; *//321*  NagySzám A = new NagySzám(a);  NagySzám B = new NagySzám(b);  NagySzám C = NagySzám.Összead(A, B);  Console.WriteLine(C.ToLong());  Console.ReadLine();  } } |
| --- |

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 konstruktor*  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.ToLong()); *//374*  Console.ReadLine();  } } |
| --- |
|  |

A NagySzám UML ábrája:



PlantUML szkriptje:

| @startuml  class NagySzám{  -const int maxHossz = 20;  -const int alap = 10;  -int[] számjegyek= new int[maxHossz];  -NagySzám(int[] szám){ ArrayAray.Copy(szám,számjegyek,szám.Length);}  +NagySzám(long szám)  +static NagySzám Összead(NagySzám S1, NagySzám S2)  +long ToLong()  }  note right of NagySzám::NagySzám  // ez mostmár privát  endnote  note left of NagySzám::"NagySzám(long szám)"  public NagySzám(long szám) { //új konstruktor  int i = 0;  while (szám > 0) {  számjegyek[i] = (int)(szám % alap);  szám /= alap;  i++;  }  }  endnote  @enduml |
| --- |

Most már akármilyen számrendszert használ a NagySzám, mindig 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.

### A GOF könyv 2. alapelve – GOF2

A GOF2 alapelv, hasonlóan a GOF1-hez, a GOF könyvben jelent meg 1995-ben. Az alapelv eredeti angol megfogalmazása: „Favor object composition over class inheritance”, azaz „Használj objektum-összetételt 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 mindig azt használjuk?

Már láttuk, hogy objektum-összetétellel mindig ki lehet váltani az öröklődést. Az öröklés azért jó, mert megörököljük az ős összes szolgáltatását (metódusait), amit használni tudunk. Objektum-összetételnél ezen osztály egy példányára szerzek 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.

### 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.

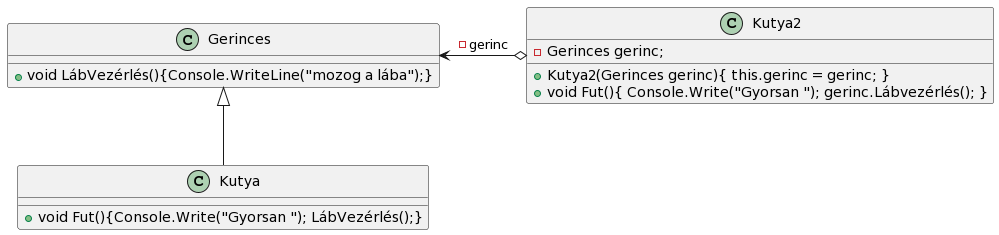
IS-A kapcsolatnak nevezzük azt is, amikor egy osztály megvalósít egy interfészt, hiszen ekkor az osztály példányai megkapják az interfész típusát is.

Az objektum-összetételt HAS-A kapcsolatnak hívjuk. Ha a Kutya osztályban van egy gerinc nevű mező, ami Gerinces osztály típusú, akkor azt mondjuk, hogy „a kutyának 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 Kutya : Gerinces {  public void Fut() {  Console.Write("Gyorsan ");  LábVezérlés();  } } class Kutya2 {  Gerinces gerinc;  public Kutya2(Gerinces gerinc) { this.gerinc = gerinc; }  public void Fut() {  Console.Write("Gyorsan ");  gerinc.LábVezérlés();  } } class Program {  static void Main(string[] args) {  Kutya bodri = new Kutya();  bodri.Fut();  Kutya2 rex = new Kutya2(new Gerinces());  rex.Fut();  Console.ReadLine();  } } |
| --- |

Az osztályok UML ábrája:



PlantUML szkriptje:

| @startuml  class Gerinces{  +void LábVezérlés(){Console.WriteLine("mozog a lába");}  }  class Kutya{  +void Fut(){Console.Write("Gyorsan "); LábVezérlés();}  }  class Kutya2{  -Gerinces gerinc;  +Kutya2(Gerinces gerinc){ this.gerinc = gerinc; }  +void Fut(){ Console.Write("Gyorsan "); gerinc.Lábvezérlés(); }  }  Gerinces <|-- Kutya  Gerinces <-o Kutya2: -gerinc  @enduml |
| --- |

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

### 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 információnk 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 információnk.

### 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.

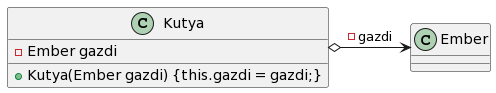
Az első szempont azt helyezi a középpontba, hogy az objektum-összetétel mindig birtoklást fejez ki. Pl. a kutyának van farka. A második szempont pedig azt hangsúlyozza ki, hogy az objektum-összetételnél a birtokolt objektumot a birtokló becsomagolja. Pl. a tűzfal becsomagol egy szervert, amely csak rajta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

A birtoklás módja szerint az objektum-összetétel lehet aggregáció és 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 Kutya  {  Ember gazdi; *// HAS-A kapcsolat, amit a gazdi nevű mező valósít meg*  public Kutya(Ember gazdi) { this.gazdi = gazdi; }  }  class Program  {  public static void Main()  {  Ember gabi = new Ember();  Kutya buksi = new Kutya(gabi); *// gabi nem csak buksi gazdija*  Kutya cézár = new Kutya(gabi); *// nem kizárólagos tulajdonlás*  }  } |
| --- |

Az osztályok UML ábrája:



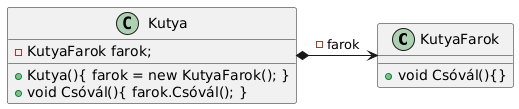
PlantUML szkriptje:

| @startuml  class Ember{}  class Kutya{  -Ember gazdi  +Kutya(Ember gazdi) {this.gazdi = gazdi;}  }  Ember <-left-o Kutya:-gazdi  @enduml |
| --- |

* 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 KutyaFarok { public void Csóvál() { } }  class Kutya  {  KutyaFarok farok; *// HAS-A kapcsolat, amit a farok nevű mező valósít meg*  public Kutya() { farok = new KutyaFarok(); } *// kizárólagos tulajdonlás*  public void Csóvál() { farok.Csóvál(); }  } |
| --- |

Az osztályok UML ábrája:



PlantUML szkriptje:

| @startuml  class KutyaFarok {  +void Csóvál(){}  }  class Kutya{  -KutyaFarok farok;  +Kutya(){ farok = new KutyaFarok(); }  +void Csóvál(){ farok.Csóvál(); }  }  KutyaFarok <-left-\* Kutya: -farok  @enduml |
| --- |

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 összetételt kell választani, azt egy egyszerű kérdéssel lehet eldönteni: Ha a gitáros meghal, vele temetik a gitárját is? Ha igen, akkor kompozícióról beszélünk, ha nem, aggregációról. Azaz, ha senki másnak nincs rá referenciája, és ezért a szemétgyűjtő felszabadítja, akkor kompozíció. Aggregációra szép példa többek közt a stratégia tervezési minta. Kompozícióra szép példa az állapot tervezési minta.

### 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 szóhasználattal azt mondjuk: A saját szolgáltatásainkhoz szükségünk van a birtokolt objektum szolgáltatásaira. Ha azért birtoklunk egy objektumot, hogy a saját szolgáltatásaink felelősségét részben vagy egészben átadjuk neki (idegen szóval delegáljuk), akkor birtoklás mellett becsomagolásról is beszélünk.

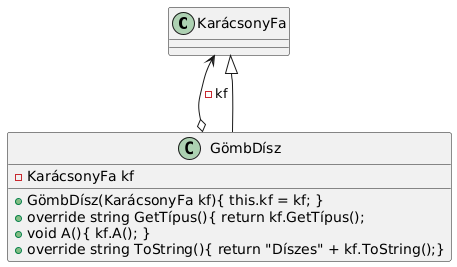
A becsomagolás módja szerint a becsomagolás lehet átlátszó vagy átlátszatlan:

* 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 karácsonyfa karácsonyfa marad, akárhány díszt is teszünk rá. Ezt úgy érjük el, hogy a dísszel becsomagoljuk a karácsonyfát, úgy hogy az objektum majd minden, díszítéstől nem függő, metódusa egyszerűen meghívja a becsomagolt karácsonyfa ugyanolyan nevű metódusát. Azért van értelme, hogy a díszítés után a karácsonyfa karácsonyfa maradjon, mert így további díszeket lehet rátenni, lásd a díszítő tervezési mintát.

Példa kódrészlet, ami követi a fenti magyarázatot:

| class GömbDísz : KarácsonyFa *// IS-A kapcsolat a KarácsonyFa osztállyal*  {  KarácsonyFa kf; *// HAS-A kapcsolat a KarácsonyFa osztállyal*  public GömbDísz(KarácsonyFa 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*  }  } |
| --- |

A GömbDísz UML ábrája:



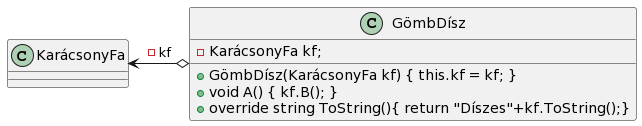
PlantUML szkriptje:

| @startuml  class KarácsonyFa{}  class GömbDísz{  -KarácsonyFa kf  +GömbDísz(KarácsonyFaKraácsonyFa kf){ this.kf = kf; }  +override string GetTípus(){ return kf.GetTípus();  +void A(){ kf.A(); }  +override string ToString(){ return "Díszes" + kf.ToString();}  }  KarácsonyFa <--o GömbDísz: -kf  KarácsonyFa <|-- GömbDísz  @enduml |
| --- |

* 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 hogy hívjuk, hanem akár hívhatjuk 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 karácsonyfa nem marad karácsonyfa, 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 KarácsonyFa osztállyal*  {  KarácsonyFa kf; *// csak HAS-A kapcsolat van*  public GömbDísz(KarácsonyFa kf) { this.kf = kf; }  public 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*  }  } |
| --- |

GömbDísz UML ábrája:



PlantUML szkriptje:

| @startuml  class KarácsonyFa{}  class GömbDísz{  -KarácsonyFa kf;  +GömbDísz(KarácsonyFa kf) { this.kf = kf; }  +void A() { kf.B(); }  +override string ToString(){ return "Díszes"+kf.ToString();}  }  KarácsonyFa <-o GömbDísz: -kf  @enduml |
| --- |

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 állunk: „Az ős gyermeke vagyok, hogy ős típusként használható legyek. Illetve becsomagolom az ősöm 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.

### Rész – egész viszony

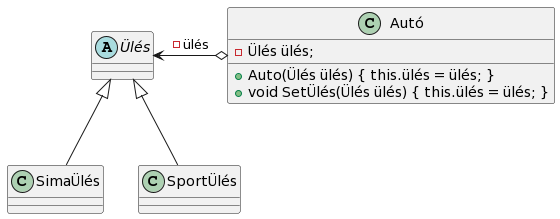
A kompozíció megfogalmazható úgy is, hogy a rész nem létezhet az egész nélkül. A kutya farka csak a kutyával együtt létezhet, ha nincs kutya, akkor nincs farka sem. Ezzel szemben aggregáció esetén a rész létezhet az egész nélkül is. Az autóból kiszedhető az ülés, az ülésen lehet ülni anélkül is, hogy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

| abstract class Ülés { */\*...\*/* } class SimaÜlés : Ülés { */\*...\*/* } class SportÜlés : Ülés { */\*...\*/* } class Autó {  Ülés ülés;  public Auto(Ülés ülés) { this.ülés = ülés; }  public void SetÜlés(Ülés ülés) { this.ülés = 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*  }  } |
| --- |

Az osztályok UML ábrája:



PlantUML szkriptje:

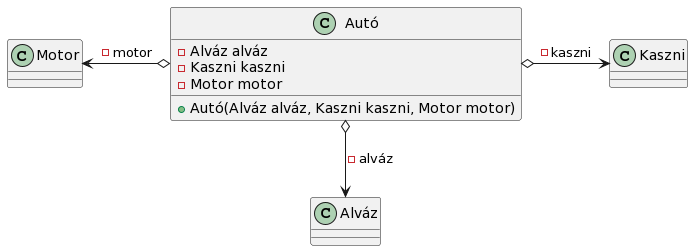
| @startuml  abstract class Ülés{}  class SimaÜlés{}  class SportÜlés{}  class Autó{  -Ülés ülés;  +Auto(Ülés ülés) { this.ülés = ülés; }  +void SetÜlés(Ülés ülés) { this.ülés = ülés; }  }  Ülés <|-- SimaÜlés  Ülés <|-- SportÜlés  Ülés <-o Autó: -ülés  @enduml |
| --- |

### 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;  } } |
| --- |

Az osztályok UML ábrája:



PlantUML szkriptje:

| @startuml  class Alváz{}  class Kaszni{}  class Motor{}  class Autó{  -Alváz alváz  -Kaszni kaszni  -Motor motor  +Autó(Alváz alváz, Kaszni kaszni, Motor motor)  }  Alváz <-up-o Autó: -alváz  Kaszni <-left-o Autó: -kaszni  Motor <-right-o Autó: -motor  @enduml |
| --- |

A fenti példában hármas objektum-összetételt látunk. Az autó 3 fő alkatrészből 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.

### 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 használjunk 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: 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.

### GOF2 a gyakorlatban

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

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a GOF könyv két alapelvét és gyakorlati alkalmazásukat. Először is, megvizsgáltuk a „Programozz felületre implementáció helyett” alapelvet (GOF1), amely arra ösztönzi a tervezőket, hogy az interfészre programozzanak, ne pedig az implementációra. A következő alapelv, a „Használj objektum-összetételt öröklés helyett, ha csak lehet” (GOF2) arra ösztönzi a tervezőket, hogy inkább objektum-összetételt alkalmazzanak az öröklés helyett. Az öröklés előnyeit és korlátait összehasonlítva láthattuk, hogy az objektum-összetétel rugalmasabb és jobban támogatja a tervezési változásokat. A továbbiakban megvizsgáltuk az IS-A és HAS-A kapcsolatot. Az IS-A kapcsolat az öröklődést jelenti, míg a HAS-A kapcsolat az objektum-összetételt. Ezután részletesen foglalkoztunk az aggregációval és a kompozícióval, két különböző objektum-összetétel típussal. Aggregáció esetén a rész létezhet az egész nélkül is. Kompozíció esetén a rész nem létezhet az egész nélkül. Végül, megvizsgáltuk az átlátszó és átlátszatlan becsomagolást, amik szintén az objektum-összetétel típusai.

Ellenőrző kérdések:

1. Kérdés: Minek a rövidítése a GOF?

Válasz: Gang of Four, magyarul: négyek bandája.

1. Kérdés: Melyik könyvre utalunk a GOF könyv szókapcsolattal?

Válasz: A könyv magyar címe: „Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz.” A könyv angol címe: „Design Patterns: Elements of Reusable Object-Oriented Software”. A könyvnek 4 szerzője van, akiket együtt a négyek bandájának, angolul: Gang of Four, szoktunk hívni. Ennek a rövidítése a GOF.

1. Kérdés: Az öröklődés az IS-A kapcsolat, vagy a HAS-A kapcsolat?

Válasz: Az öröklődés az IS-A kapcsolat.

1. Kérdés: Az objektum-összetétel az IS-A kapcsolat, vagy a HAS-A kapcsolat?

Válasz: Az objektum-összetétel a HAS-A kapcsolat.

1. Kérdés: A kompozíció az IS-A, vagy a HAS-A kapcsolat egy fajtája?

Válasz: A HAS-A kapcsolat egy fajtája.

1. Kérdés: Az aggregáció az IS-A, vagy a HAS-A kapcsolat egy fajtája?

Válasz: A HAS-A kapcsolat egy fajtája.

1. Kérdés: A becsomagolás az IS-A, vagy a HAS-A kapcsolat egy fajtája?

Válasz: A HAS-A kapcsolat egy fajtája.

## Lecke 8.: SOLID tervezési alapelvek

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a SOLID tervezési alapelvekkel.

Szükséges eszközök, források: Robert C. Martin Tiszta kód című könyve. Robert C. Martin PPP könyve.

Feldolgozási idő: 90 perc.

Témakörök: A SOLID alapelvek, úgy mint: SRP, OCP, LSP, ISP, DIP.

A lecke tartalma: Ebben a leckében áttekintjük a szakirodalomban leginkább elfogadott 5 objektum orientált tervezési alapelvet, amit összefoglaló néven SOLID alapelveknek szoktunk nevezni. Robert C. Martin, akit sokan csak Bob bácsinak (angolul: Uncle Bob) hívnak, szerint a legfontosabb alapelv a SOLID elvek közül az első, az SRP, azaz az Egy felelősség elve. Ez az elv kimondja, hogy minden modulnak, osztálynak csak egy felelősségi köre legyen, de azt fedje le teljesen. A jegyzet szerzői szerint a legfontosabb alapelv, a SOLID elvek közül a második, az OCP, azaz a Nyitva-zárt alapelv. Ez az elv kimondja, hogy az osztályok, illetve az osztály hierarchiák legyenek nyitottak a bővítésre, de zártak a módosításra. Ez az elv azért nagyon fontos, mert szintaxis szintjén is megfogalmazható: Ne használjunk override kulcsszót, csak absztrakt, illetve horog (angolul: hook) metódusok felülírására, illetve ne használjunk if-else if szerkezeteket. Az SRP-t nem lehet ilyen alacsony szinten megfogalmazni. A többi SOLID alapelvet is bemutatjuk egy-egy példán keresztül.

### Bevezetés

A SOLID tervezési alapelvek fogalom a legfontosabb, a szakirodalomban leginkább elismert, 5 objektumorientált tervezési alapelvet jelöli. Ezek a következők:

* SRP (Single Responsibility Principle) – Egy felelősség - egy osztály alapelve
* OCP (Open-Closed Principle) – Nyitva-zárt alapelv
* LSP (Liskov Substitution Principle) – Liskov-féle behelyettesítési alapelv
* ISP (Interface Segregation Principle) – Interfészszegregációs-alapelv
* DIP (Dependency Inversion Principle) – Függőség megfordításának alapelve

Az első betűket összeolvasva kapjuk: SOLID. Lássuk ezeket bevezető szinten.

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

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 kapunk, amihez már senki se mer hozzányúlni. 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, lásd egy későbbi fejezetben. Ezeket a felelősségeket, mint például a naplózás, kiemeljük egy úgynevezett aspektusba, amit bármely osztályhoz hozzákapcsolhatunk.

Robert C. Martin Tiszta kód című könyve szerint a legfontosabb SOLID alapelv az SRP.

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

### OCP (Open-Closed Principle) – Nyitva-zárt alapelv

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 metódusunk és azt megváltoztatjuk, akkor több negatív dolog is történhet:

* a változás miatt az eddig működő ágak hibásak lesznek,
* a változás miatt a vele implementációs függőségben lévő kódrészeket is változtatni kell,
* a változás általában azt jelenti, hogy olyan esetet kezelünk 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 gyermekosztályok opcionálisan bővíthessék megvalósításukat. 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 őst, az absztrakt Alakzatot. A konkrét alakzatok csak felülírják az ős 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 ős másik előnye az, hogy ha a kódban a gyermekosztály példányait csak az közös ős 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 horog (angolul: hook) metódusokra is ad példát.

### LSP (Liskov Substitution Principle) – Liskov-féle behelyettesítési alapelv

A Liskov-féle behelyettesítési elv (angolul: Liskov Substitution Principle), rövid nevén LSP, kimondja, hogy a program viselkedése nem változhat meg attól, hogy az ős osztály egy példánya helyett a jövőben valamelyik gyermekosztályának példányát használjuk. Azaz a program által visszaadott érték nem függ attól, hogy egy Kutya vagy egy Vizsla vagy egy Komondor példány lábainak számát adjuk 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 gyermekosztályban 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 gyermekosztályát a Négyzetet példányosítja. Ha az LSP igaz lenne, akkor mindegy lenne, melyik osztály példányán keresztül hívjuk a Terület metódust, de ez nem igaz, mert a SetA és a SetB teljesen másképp 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.

### Szerződésalapú programozás – Design by Contract

Az előzőekben használtuk az altípus fogalmát az LSP megfogalmazásához, de nem definiáltuk, mi is az altípus. Ezt pótolja ez a rész. 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 kimenetre 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 bemenetre 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 állapot közötti átmenetet ír le, a metódus futása előtti és utáni állapotét. Az elő- és utófeltétel párok megadása helyett adhatunk egy úgynevezett 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 állapotátmeneti megszorításából következik a T állapotátmeneti megszorítása.

Az utolsó feltételre azért van szükség, mert a gyermekosztályban lehetnek új metódusok is, és ezeknek is be kell tartaniuk az ős állapotátmeneti 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 == 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 ÉS b == x ==> a == x ÉS b == \régi(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 megszorítását (angolul: 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.

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

Az interfészszegregációs-alapelv (angolul: Interface Segregation Principle – ISP) azt mondja ki, hogy egy sok szolgáltatást nyújtó osztály fölé el kell helyezni interfészeket, 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 függjön, 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 interfészek 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) kizárja, 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.

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

A függőség megfordításának elve (angolul: Dependency Inversion Principle – DIP) azt mondja ki, hogy a magas szintű komponensek ne függjenek alacsony szintű implementációs részleteket kidolgozó osztályoktól, se fordítva, hanem a magas szintű és az alacsony szintű komponensek is absztrakciótól függjenek. 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 moduloktól. Mindkettő függjön az absztrakciótól.” Ezt ennél frappánsabban így szoktuk mondani: „Absztrakciótól függj, ne függj konkrét osztályoktól”.

Az alacsony szintű komponensek újrafelhasználása jól megoldott az úgynevezett osztálykönyvtárak (angolul: 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 fájl vége jelig. Ezt a logikát sok helyen fel lehet használni, hiszen a forrás bármi lehet és a cél is, ami karaktereket tud beolvasni, illetve kiírni. Ha most ezt a kódot újra akarjuk hasznosítani, akkor két lehetőségünk van. Az első, hogy if – else – if szerkezet segítségével megállapítjuk, hogy most melyik forrásra, illetve célra van szükségünk. Ez nagyon csúnya, nehezen átlátható, módosítható kódot eredményez. A másik lehetőség, hogy a forrás és a cél referenciáját kívülről adjuk meg a hívó felelősség injektálásával (angolul: dependency injection).

A felelősség injektálásának több típusa is létezik:

* Felelősség injektálása konstruktorral: Ebben az esetben az osztály a konstruktorán keresztül kapja meg azokat a referenciákat, amiken keresztül a neki hasznos szolgáltatásokat meg tudja hívni. Ezt más néven objektum-összetételnek is nevezzük és a leggyakrabban épp így programozzuk le. (Javascript keretrendszereknél gyakran alkalmazott módszer, Angular JS)
* Felelősség injektálása szetter metódusokkal: Ebben az esetben az osztály szetter metódusokon keresztül kapja meg azokat a referenciákat, 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 paraméterosztály paramétereként is jöhet az interfész.
* Felelősség injektálása elnevezési konvenció, konfigurációs állomány, vagy annotáció alapján. Ez általában keretrendszerekre 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 függőség megfordítás elvének megfelelő változata látható a következő példában. A fenti lehetőségek közül a felelősség beinjektálás konstruktorral 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. Mindenesetre a haszna vitathatatlan, ha rugalmas kód fejlesztésére törekszünk.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a szakirodalomban leginkább elfogadott 5 objektum orientált tervezési alapelvet, amit összefoglaló néven SOLID alapelveknek szoktunk nevezni. Robert C. Martin, akit sokan csak Bob bácsinak (angolul: Uncle Bob) hívnak, szerint a legfontosabb alapelv a SOLID elvek közül az első, az SRP, azaz az Egy felelősség elve. Ez az elv kimondja, hogy minden modulnak, osztálynak csak egy felelősségi köre legyen, de azt fedje le teljesen. A jegyzet szerzői szerint a legfontosabb alapelv, a SOLID elvek közül a második, az OCP, azaz a Nyitva-zárt alapelv. Ez az elv kimondja, hogy az osztályok, illetve az osztály hierarchiák legyenek nyitottak a bővítésre, de zártak a módosításra. Ez az elv azért nagyon fontos, mert szintaxis szintjén is megfogalmazható: Ne használjunk override kulcsszót, csak absztrakt, illetve horog (angolul: hook) metódusok felülírására, illetve ne használjunk if-else if szerkezeteket. Az SRP-t nem lehet ilyen alacsony szinten megfogalmazni. A többi SOLID alapelvet is bemutattuk egy-egy példán keresztül.

Ellenőrző kérdések:

1. Kérdés: Mik azok a SOLID alapelvek?

Válasz: A szakirodalomban leginkább elfogadott 5 legfontosabb objektum orientált tervezési alapelv: SRP, OCP, LSP, ISP, DIP.

1. Kérdés: Minek a rövidítése az SRP és mint mond ki?

Válasz: Ennek a rövidítése: Single Responsibility Principle, magyarul: Egy felelősség elve. Kimondja, hogy minden osztálynak, illetve modulnak csak egy felelősségi köre legyen, de azt fedje le teljesen.

1. Kérdés: Minek a rövidítése az OCP és mit mond ki?

Válasz: Ennek a rövidítése: Open-Closed Principle, magyarul: Nyitva-zárt alapelv. Kimondja, hogy az osztályok, illetve az osztály hierarchiák legyenek nyitottak a bővítésre, de zártak a módosításra.

1. Kérdés: Robert C. Martin Tiszta kód című könyve szerint melyik a legfontosabb SOLID alapelv?

Válasz: Az SRP.

1. Kérdés: Mi a jelentősége az OCP alapelvnek?

Válasz: Az OCP elvet kimondhatjuk a szintaxis szintjén is, míg ez az SRP esetén nem ilyen egyértelmű.

1. Kérdés: Melyik SOLID alapelv megértéséhez van szükség a szerződés alapú programozásra?

Válasz: Az LSP megértéséhez kell bevezetni a szerződés alapú programozást.

## További tervezési alapelvek

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

### 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 áhított 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 tevékeny várakozás (angolul: busy waiting).
* Következő alkalommal a titkár már jó előre közli minden színésszel, 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 tevékeny várakozás 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 tevékeny várakozás megoldásnak van létjogosultsága, de csak nagyon kevés helyzetben. A legismertebb az őrkutya (angolul: watchdog), amikor egy távoli objektumot kérdezgetünk (pingelgetünk) 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 az őrkutya észreveszi, hogy a távoli objektum lehalt, akkor annak feladatát másra osztják.

A Hollywood elv azt mondja ki, hogy ne azt kérdezgessük, hogy megtörtént-e egy adott esemény, hanem ez az esemény értesítse a várakozókat. Ezt a megoldást használja például a Java eseménykezelése. Ha lenyomunk egy gombot, akkor keletkezik egy esemény, de ezen túl semmi se történik. Ha azt akarjuk, hogy történjen is valami, akkor fel kell iratkoztatni az eseményre egy figyelőt (Java nyelvhasználattal listener-t). Ha kiváltódik az esemény, akkor az összes feliratkozott figyelő értesítést kap. Pontosan ezt valósítja meg a Megfigyelő 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.

### 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 referenciánk, azaz az A.getB().C() alakú hívások tilosak. Ez az elv azért hasznos, mert ha betartjuk, akkor a változások mindig csak lokális hatásúak lesznek.

# Patterns: 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:

* „The GOF book”: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, 1995.
* „A GOF könyv”: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Programtervezési minták, Újrahasznosítható elemek objektumközpontú programokhoz, Kiskapu Kiadó, 2004.
* Robert C. Martin: Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2002.
* Eric Freeman, Elisabeth Robson, Kathy Sierra, Bert Bates: Head First Design Patterns, O'Reilly Media, 2004.
* Alan Shalloway, James Trott: Design Patterns Explained, A New Perspective on Object Oriented Design, 2004.
* Cay S. Horstmann: Object-Oriented Design and Patterns, Wiley, 2006.
* Robert C. Martin: Clean Architecture, A Craftsman's Guide to Software Structure and Design, 2017.
* Vaskaran Sarcar: Design Patterns in C#: A Hands-on Guide with Real-world Examples, 2020.
* J.J. Geewax: API Design Patterns, 2021.
* Addy Osmani: Learning JavaScript Design Patterns: A JavaScript and React Developer's Guide, 2023.

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

A minták alkalmazásával könnyen bővíthető, módosítható osztályszerkezetet kapunk, tehát rugalmas kódot. Az ár, amit ezért fizetünk, a bonyolultabb, nehezebben 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 absztrakciója. 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.

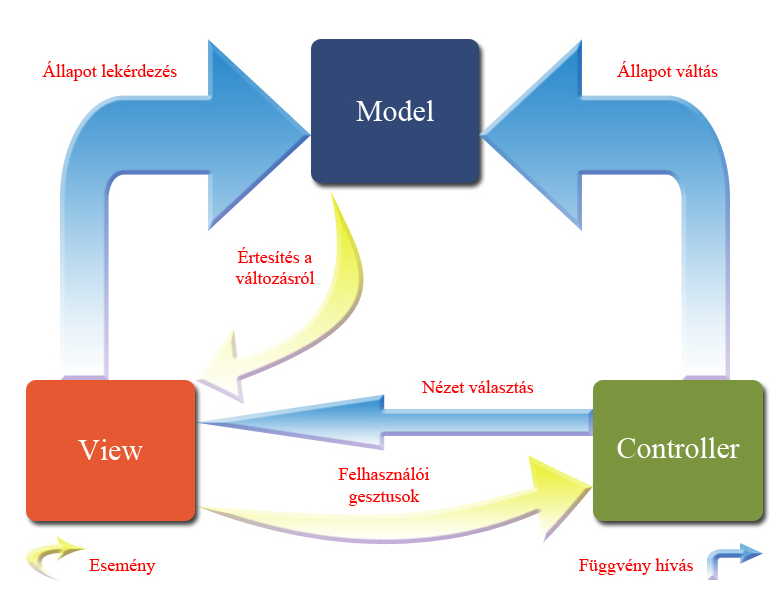
## Architekturális minták

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

### MVC – Model-View-Controller

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

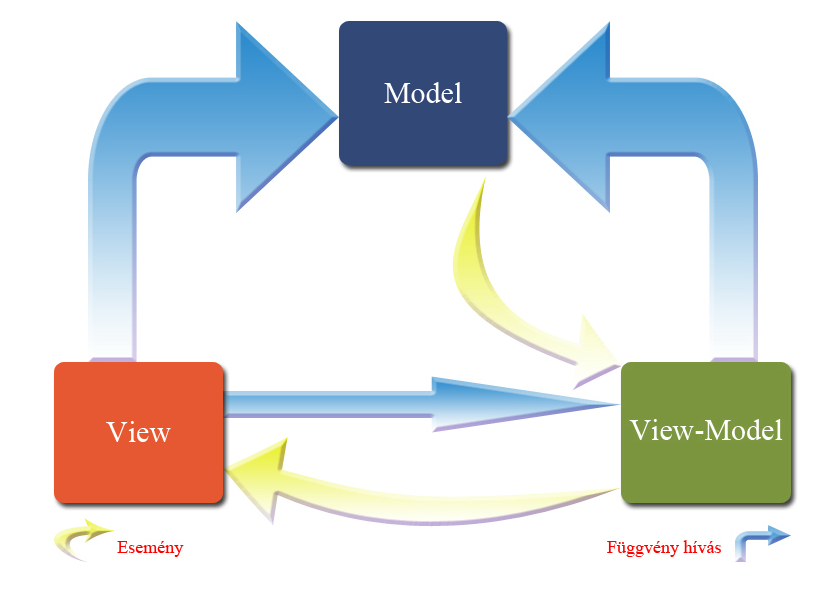
* Model (magyarul modell): Az adatokat kezelő réteg. Ez felel az adatok tárolásáért, visszaolvasásáért. Itt foglalnak helyet azok a függvények is, amik műveleteket végeznek az adatokon. Része az adatbázis is.
* View (magyarul nézet): A felhasználói felület megjelenítéséért, a felhasználó különféle nyűgjeinek 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.



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 :), illetve, 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. Bármelyik nézeten lenyomnak egy gombot, az az esemény eljut a kontrollerhez. A kontroller meghívja a modell megfelelő metódusát. Ha emiatt a modell belső állapota megváltozik, akkor a modell a megfigyelő tervezési mintának megfelelően értesíti a nézeteket, hogy változás történt, nekik is meg kell változni.



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. A nézet nem lép interakcióba az üzleti logikával.
* 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.

### Többrétegű architektúra

Réteg alatt a program olyan jól elszeparált részét értjük, amely akár külön számítógépen futhat. A rétegek jól definiált interfészeken keresztül kommunikálnak, mindig 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.

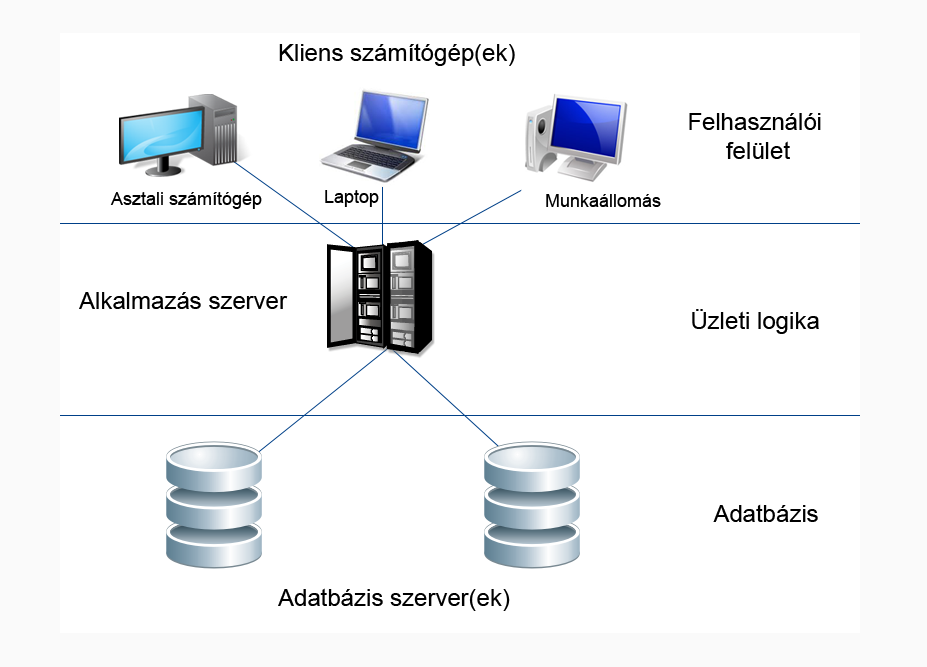
A 3 rétegű architektúra általában három számítógépet használ:

* 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 rendelkezésére. | 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.



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 (angolul: load balancing) megold helyettünk az alkalmazásszerver.

## Lecke 9.: Létrehozási tervezési minták

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk néhány létrehozási tervezési mintával egy-egy példán keresztül.

Szükséges eszközök, források: A GOF könyv. A Head First Design Patterns könyv.

Feldolgozási idő: 90 perc.

Témakörök: Létrehozási tervezési minták, Egyke, Prototípus, Gyártómetódus, Absztrakt gyár.

A lecke tartalma: Ebben a leckében áttekintjük a 4 legegyszerűbb létrehozási tervezési mintát, úgy mint: Egyke, Prototípus, Gyártómetódus, Absztrakt gyár. A létrehozási tervezési minták fő feladata, hogy a példányosítás folyamatát egységbe zárják, így, ha módosul a létrehozás folyamata, akkor azt csak egy helyen kell megváltoztatni.

### Bevezetés

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 mindenhová 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.

### 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 konstruktorával készíthetünk. Ha van publikus konstruktor az osztályban, akkor akárhány példány készíthető belőle. Tehát publikus konstruktora nem lehet az egykének. De ha nincs konstruktor, 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 mindenkinek ugyanazt a példányt adja vissza. Ezt a metódust szokás globális hozzáférési pontnak is hívni. Természetesen ezt a példányt is létre kell hozni, de a privát konstruktort a szerezPéldány hívhatja, hiszen ő is az egyke osztály része. Forráskód:

using System;

namespace Singleton

{

public class Singleton

{

// statikus mező az egyetlen példány számára

private static Singleton uniqueInstance=null;

// privát konstruktor, hogy ne lehessen 'new' kulcsszóval példányosítani

private Singleton() { }

// biztosítja számunkra a példányosítást és egyben visszaadja a példányt

// mindenkinek 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 konstruktor 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();

}

}

Szálbiztos megoldás

using System;

namespace SingletonThreadSafe

{

public sealed class Singleton

{

// A statikus konstruktor 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 konstruktor külső 'new' példányosítás ellen

private Singleton() { }

// statikus konstruktor

// 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 GetInstance { return instance; }

}

class Program

{

static void Main(string[] args)

{

Singleton s1 = Singleton.GetInstance();

Singleton s2 = Singleton.GetInstance();

if (s1 == s2) { Console.WriteLine("OK"); }

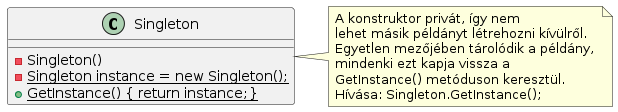
Console.ReadKey();

}

}

}

UML ábrája:



Példa UML ábra az Egyke tervezési minta

PlantUML szkriptje:

@startuml

class Singleton {

- Singleton()

- {static} Singleton instance = new Singleton();

+ {static} GetInstance() { return instance;}

}

note right of Singleton

A konstruktor privát, így nem

lehet másik példányt létrehozni kívülről.

Egyetlen mezőjében tárolódik a példány,

mindenki ezt kapja vissza a

GetInstance() metóduson keresztül.

Hívása: Singleton.GetInstance();

end note

@enduml

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 nem?

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 úgynevezett 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 konstruktorhívással 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 ahhoz 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 konstruktort, 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 konstruktor, attól van konstruktora, tehát példányosítani is lehet, igaz ez a konstruktor csak belülről hívható.
3. Igen, belülről minden metódus hívható, akármilyen is a láthatósági szintje.
4. Igen, ez jó megoldás. Arra kell figyelni, hogy a konstruktor 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 mindenkinek 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 private vagy protected konstruktora, nincs public konstruktora.
* Van egy Egyke típusú osztály szintű metódusa, aminek szokásos neve „instance”.
* Van egy publikus Egyke típusú metódusa, amely mindig 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 kellenek 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 egy főprogramot, amely kétszer is lekéri az Egykéből a híváslistát. Adjunk hozzá az egyikhez egy hívást, és listázzuk ki a másikat. Vegye észre, hogy ha az egyiket szerkesztjük, a másik is változik, hiszen a két lista ugyanoda mutat, mivel csak egy Híváslista példány van, és 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 helyettesítenek.

### 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 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 referenciáját 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 sekély 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.

Példa: A prototípus mintát egy példán keresztül mutatjuk be: Hurrá, a magyar gépkocsigyártás újra feléledt, legalábbis a példánk kedvéért. Megjöttek a befektetők, és utasítást adtak a tehergépkocsi gyártás beindítására! 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 eladás bevételeiből, és végre megvettük a fent említett márkákat, könnyű dolgunk lesz az implementáció során, hiszen egy új osztályban beállítjuk a sportkocsi végsebességét, a tankméretet kisebbre vesszük és indulhat a sorozatgyártás a gyáron keresztül, és a határ a csillagos ég, vagy a Forma 1! A gyártósor egy prototípust vár. Mindegy, hogy milyen Gépkocsit kap, mindent tud gyártani, mert csak klónozza a prototípust. A klónozáson túl csak festeni tud. Szóval a gyár buta, de hatékony.

A lenti forráskódban figyeljük meg, hogy sekély klónozást használunk. Ezt kétféleképpen 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. 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

Gépkocsi[] tk = gyartosor.sorozatgyartas(prototipus2, "Szürke", 20);

foreach (Teherautó t in tk) { Console.WriteLine(t); }

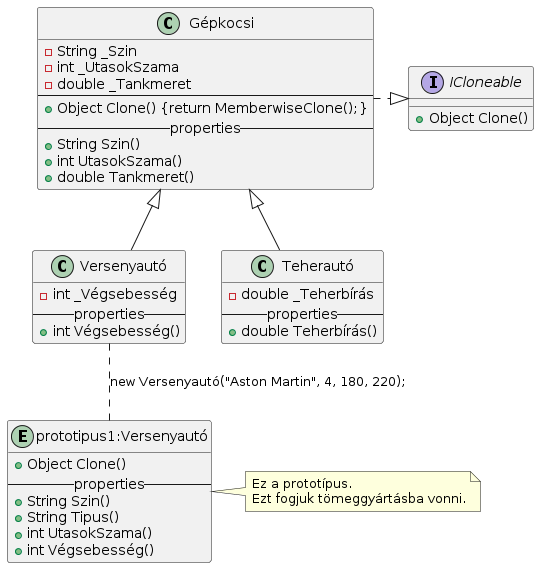
Console.ReadLine();

}

}

}

UML ábra:



Példa UML ábra a prototípus tervezési mintára

PlantUML szkript:

@startuml

interface ICloneable {

+ Object Clone()

}

class Gépkocsi {

- String \_Szin

- int \_UtasokSzama

- double \_Tankmeret

--

+Object Clone() {return MemberwiseClone();}

--properties--

+String Szin()

+int UtasokSzama()

+double Tankmeret()

}

class Versenyautó {

- int \_Végsebesség

--properties--

+ int Végsebesség()

}

class Teherautó {

- double \_Teherbírás

--properties--

+double Teherbírás()

}

Gépkocsi .right.|> ICloneable

Gépkocsi <|-down- Teherautó

Gépkocsi <|-down- Versenyautó

entity prototipus1:Versenyautó {

+Object Clone()

--properties--

+String Szin()

+String Tipus()

+int UtasokSzama()

+int Végsebesség()

}

Versenyautó .. "prototipus1:Versenyautó" : new Versenyautó("Aston Martin", 4, 180, 220);

note right of "prototipus1:Versenyautó"

Ez a prototípus.

Ezt fogjuk tömeggyártásba vonni.

end note

@enduml

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 végiggondolni, 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 funkcionalitását 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 szériaszámot kell adnia, akkor ezt elég a Gyárt metódusba lekódolni, míg a másik esetben minden new után bele kellene illeszteni ezt a kódba.

### 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 a készít, angolul a create szóval kezdjük. A gyártómetódus a nevében megadott 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() konstruktor hívás, mert itt az elkészítés algoritmusát egységbe 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 gyermekosztály dönti el.

Tehát az ősben lévő gyártómetódus leírja a gyártás algoritmusát, a gyermekosztály 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 eldöntse, hogy mit kell gyártani. A gyermekosztályok itt hívják meg a termék konstruktorát.
* A gyártás opcionális lépései: Ezek az ősben horog (angolul: 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 ős és a gyermekosztályai IoC (angolul: Inversion of Control) viszonyban állnak. Azaz nem a gyermek hívja az ős metódusait, hanem az ős a gyermekét. Ezt úgy érjük el, hogy a gyártómetódus absztrakt-, illetve virtuális metódusokat hív. Amikor a gyermekosztály példányán keresztül hívjuk majd a gyártómetódust, akkor a késői kötés miatt ezen metódusok helyett az őket felülíró gyermekbéli metódusok fognak lefutni. 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-mínősítésben 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();

}

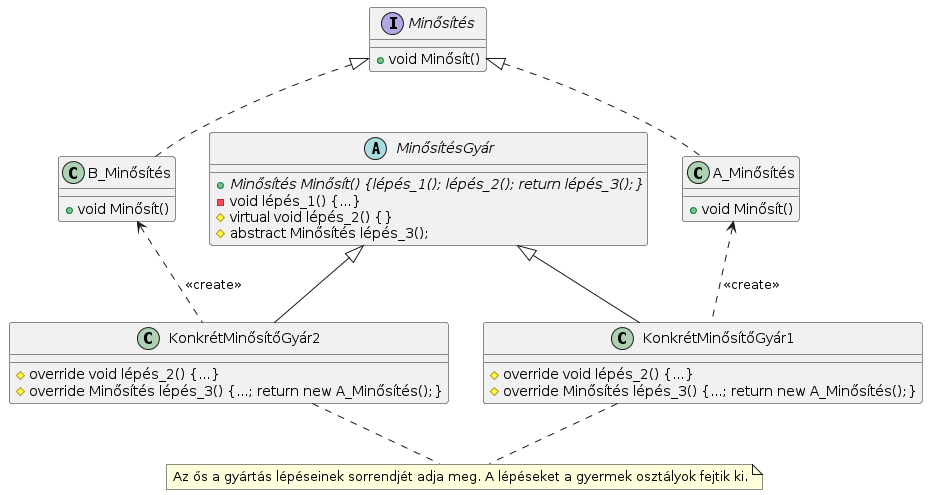
Console.ReadLine();

}

}

}

UML ábra:



Példa UML ábra a gyártómetódus tervezési mintára

PlantUML szkript:

@startuml

interface Minősítés {

+ void Minősít()

}

class A\_Minősítés {

+ void Minősít()

}

class B\_Minősítés {

+ void Minősít()

}

abstract class MinősítésGyár {

+ {abstract} Minősítés Minősít() {lépés\_1(); lépés\_2(); return lépés\_3();}

- void lépés\_1() {...}

# virtual void lépés\_2() {}

# abstract Minősítés lépés\_3();

}

class KonkrétMinősítőGyár1 {

# override void lépés\_2() {...}

# override Minősítés lépés\_3() {...; return new A\_Minősítés();}

}

class KonkrétMinősítőGyár2 {

# override void lépés\_2() {...}

# override Minősítés lépés\_3() {...; return new A\_Minősítés();}

}

note "Az ős a gyártás lépéseinek sorrendjét adja meg. A lépéseket a gyermek osztályok fejtik ki." as N1

Minősítés <|.. A\_Minősítés

Minősítés <|.. B\_Minősítés

MinősítésGyár <|-- KonkrétMinősítőGyár1

MinősítésGyár <|-- KonkrétMinősítőGyár2

KonkrétMinősítőGyár1 .up.> A\_Minősítés : <<create>>

KonkrétMinősítőGyár2 .up.> B\_Minősítés : <<create>>

KonkrétMinősítőGyár1 .. N1

KonkrétMinősítőGyár2 .. N1

@enduml

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

### Absztrakt gyár – Abstract Factory

Az Absztrakt gyár (angolul: Abstract 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 absztrakt 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. 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 Motor CreateMotor() { return new OpelMotor(); }

}

class MerciGyár : AutóGyár

{

public override Alváz CreateAlváz() { return new MerciAlváz(); }

public override Motor CreateMotor() { return new MerciMotor(); }

}

class Program

{

static void Main(string[] args)

{

AutóGyár gyár = new OpelGyár();

Alváz alváz1 = gyár.CreateAlváz();

Motor motor1 = gyár.CreateMotor();

Console.WriteLine("Létrehozott egy {0} alvázat és egy {1} motort.", alváz1.GetType().Name, motor1.GetType().Name);

Console.ReadLine();

}

}

}

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 mindkét 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.

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

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a 4 legegyszerűbb létrehozási tervezési mintát, úgy mint: Egyke, Prototípus, Gyártómetódus, Absztrakt gyár. A létrehozási tervezési minták fő feladata, hogy a példányosítás folyamatát egységbe zárják, így, ha módosul a létrehozás folyamata, akkor azt csak egy helyen kell megváltoztatni.

Ellenőrző kérdések:

1. Kérdés: Mi a létrehozási tervezési minták feladata?

Válasz: A példányosítás folyamatának egységbe zárása, így, ha változik a folyamata, akkor azt csak egy helyen kell megváltoztatni.

1. Kérdés: Mi a Singleton (magyarul: egyke) tervezési minta célja?

Válasz: Létrehozni egy olyan osztályt, aminek csak egy példánya lehet.

1. Kérdés: Miben különbözik a mohó és a lusta Singleton (magyarul: egyke) tervezési minta?

Válasz: A mohó Singleton akkor is létrehozza a példányt, ha azt soha senki nem kéri le. A lusta Singleton csak akkor hozza létre, ha valaki lekéri a példányt, de az még nincs kész.

1. Kérdés: Milyen típusai vannak a klónozásnak, ezek miben különböznek?

Válasz: Két szélsőséges típusa van: a sekély klónozás és a mély klónozás. Sekély klónozás esetén minden mezőt egyszerű értékadással másolok. Mély klónozás esetén a referenciákat klónozom. A két szélsőség közt bármilyen átmenet elképzelhető.

1. Kérdés: Minden Office alkalmazásnak van Fájl->Új parancsa, amik nagyon hasonlóan működnek, mégis mindegyiknek teljesen más az eredménye. Melyik tervezési minta van ennek a hátterében?

Válasz: A gyártómetódus, hiszen a gyártómetódusban a gyártási folyamat lépései fixek, de az egyes gyermekosztályok fejtik ki ezeket a lépéseket.

1. Kérdés: Ha több összefüggő terméket kell gyártani, akkor melyik tervezési mintát érdemes használni?

Válasz: Az absztrakt gyárat, hiszen ez pont arra való, hogy több, összefüggő terméket gyártsunk a segítségével.

## Lecke 10.: Szerkezeti tervezési minták

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk néhány szerkezeti tervezési mintával egy-egy példán keresztül.

Szükséges eszközök, források: A GOF könyv. A Head First Design Patterns könyv.

Feldolgozási idő: 25 perc.

Témakörök: Szerkezeti tervezési minták, Illesztő, Díszítő, Helyettes.

A lecke tartalma: Ebben a leckében áttekintjük a 3 legegyszerűbb szerkezeti tervezési mintát, úgy mint: Illesztő, Díszítő, Helyettes. A szerkezeti tervezési minták fő feladata, hogy megmutassák, hogyan kell használni az objektum-összetételt, hogy az elvárásainknak megfelelő futási idejű szerkezeteket tudjuk létrehozni.

### Bevezetés

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 fajtáit, 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 kizárólagos tulajdona az őt tartalmazó objektumnak,
* kompozíció: amikor kizárólagos tulajdona,
* átlátszó csomagolás: amikor a tulajdonos ugyanolyan típusú, mint az összetételben szereplő objektum, és így a felületük ugyanolyan.
* átlátszatlan csomagolás: amikor a tulajdonos nem ugyanolyan típusú, mint az összetételben szereplő objektum, és így a felületük más.

### 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 felülethez, 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 kizárólagosan 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.

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. Forráskód:

using System;

abstract class Ember

{

public abstract string GetNév();

public abstract int GetIQ();

}

class Robot

{

string ID;

int memory; //memoria 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();

}

}

### Díszítő – Decorator

A díszítő tervezési minta az átlátszó becsomagolás jó példája. Gyakran a karácsonyfa díszítés feladatán keresztül mutatjuk be: Attól, hogy a karácsonyfára felteszek egy gömböt, az még karácsonyfa marad, azaz a díszítés átlátszó. Ezt úgy érjük el, hogy az objektum-összetételben szereplő mindkét 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 lennének.

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

A díszítés során az ős minden metódusát implementálni kell, úgy, hogy a becsomagolt példány metódusát meghívjuk, illetve ahol ez szükséges, ott hozzáadjuk a plusz funkcionalitást. Kétféle díszítésről beszélhetünk, azaz a díszítő mintának két fajtája van:

* Amikor a meglévő metódusok felelősségkörét 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())…);

Az ilyen fajta teleszkópszerűen nyúló konstruktor hívás szerkezeteket teleszkópikus konstruktornak hívjuk.

Mivel a csomagolás átlátszó, ezért akárhányszor becsomagolhatjuk a példányunkat, akár egy díszítővel kétszer 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 ős 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.

Példa: A díszítőmintát a következő példával mutatjuk be. Képzeljük el, hogy egy versenypályán üzemeltetünk egy autókölcsönzőt. Az autókölcsönzőben természetesen több típusú autót, többnyire versenyautót adunk kölcsönzésre. A lényeg, hogy előfordulhat az, hogy újabb autókkal bővítjük az állományt. Felkészülve erre, először egy alapautó-osztályt hoznak létre, amelyben a bérelhető autók információi szerepelnek, mint gyártó neve, a modell neve, a bérlés időtartama körökben számolva és a bérlés díja. A kölcsönzőben időnként akciókkal kedveskednek az ügyfeleknek, valamint változó, hogy egy bizonyos autó kölcsönözhető-e vagy sem. Ezen extrák hozzáadását a díszítőminta implementálásával tették lehetővé. Az alapautó-osztályból származik az alapdekorátor-osztály, amely elvégzi a becsomagolást. A konkrét díszítőosztályoknak már csak a funkciók kibővítésével kell foglalkozniuk. Amint egy autót feldíszítünk, mint kölcsönözhető, az már csak a bérlőjét 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. Forráskód:

using System;

namespace DecoratorDesignPattern

{

public abstract class VehicleBase // alap osztály, adott funkcionalitásokkal

{

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 kölcsönzött {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");

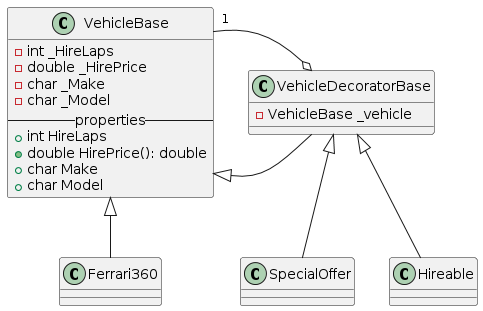
Console.ReadLine();

}

}

}

UML ábra:



Példa UML ábra a díszítő tervezési mintára

PlantUML szkript:

@startuml

class VehicleBase{

- int \_HireLaps

- double \_HirePrice

- char \_Make

- char \_Model

--properties--

+ int HireLaps

+ double HirePrice(): double

+ char Make

+ char Model

}

class VehicleDecoratorBase{

- VehicleBase \_vehicle

}

class Ferrari360{}

class SpecialOffer{}

class Hireable{}

VehicleBase <|-- Ferrari360

VehicleBase <|-right- VehicleDecoratorBase

VehicleDecoratorBase o-left- "1" VehicleBase

VehicleDecoratorBase <|-- SpecialOffer

VehicleDecoratorBase <|-- Hireable

@enduml

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 csillogó-villogó bringát fel-alá gurulni a szalonban. Okosan, vagy inkább objektumorientáltan-t kéne mondanom, 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: BringaCsengovel21sebAluvazSarvedoveldecsakElolAkcios. 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 neveztünk 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 konstruktoruk 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, meghívjuk 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 karácsonyfa programot, amely egy alap fenyőfát díszíthetünk különböző díszekkel. A legfontosabb megfigyelés az az, hogy egy karácsonyfa karácsonyfa marad a díszítés után is. Ez azt jelenti, hogy átlátszó becsomagolást kell használnunk.

A feladat megoldásához kell egy absztrakt ős, amely rögzíti a karácsonyfa 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 őst létrehozni.

AbsztraktKarácsonyfa osztály:

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

LucFenyő osztály:

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

AbsztraktDísz osztály:

* Az AbsztraktKarácsonyfa osztályból származik, ezt lehet díszíteni.
* karácsonyfa: AbsztraktKarácsonyfa típusú mező. Azt a karácsonyfát tartalmazza, amire a dísz kerül. Ez egy objektumösszetétel, azon belül is átlátszó becsomagolás, mert az AbsztraktDísz és a karácsonyfa is AbsztraktKarácsonyfa típusú.
* AbsztraktDísz(karácsonyfa): Konstruktor, beállítja a karácsonyfa nevű mezőt. Mivel egy objektum összetétel kívülről kap értéket, ezért ez egy felelősség beinjektálás.
* DíszRajzolás(): Absztrakt metódus, a gyermekosztályok fogják kifejteni. Kirajzolja, vagy csak kiírja, a díszt.
* KiRajzol(): Kódja: DíszRajzolás(); karácsonyfa.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 karácsonyfa, amit tovább lehet díszíteni.
* GömbDísz(karácsonyfa): Konstruktor, egyszerűen meghívja az ős konstruktorát a karácsonyfa paraméterrel.
* DíszRajzolás(): 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.

### Helyettes – Proxy

A helyettes (angolul: proxy) tervezési minta egy nagyon egyszerű kompozícióra ad példát, ami ráadásul átlátszó becsomagolás. Egy valamilyen 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.
* Gyorsítótár (angolul: cache): Ha van olyan számítás (ide sorolva a letöltéseket is), ami drága, akkor a számítás eredményét érdemes letárolni egy gyorsítótárban, ami szintén egyfajta proxy.

Mivel nagyon sokféle helyettes létezik, ezért ezeket összefoglalóan helyettes állatkertnek (angolul: Proxy Zoo) szoktuk nevezni.

Forráskód – Példa 1.

using System;

namespace helyettes

{

class MainApp

{

static void Main()

{

// Készítünk egy helyettest 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(); }

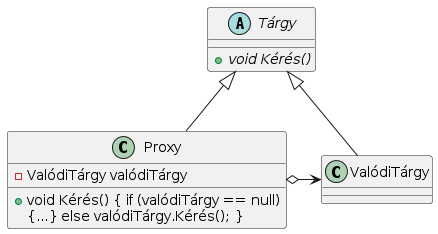
valódiTárgy.Kérés();

}

}

}

UML ábra:



UML ábra a helyettes tervezési mintára

PlantUML szkript:

@startuml

abstract class Tárgy {

+ {abstract} void Kérés()

}

class ValódiTárgy {}

class Proxy {

- ValódiTárgy valódiTárgy

+ void Kérés() { if (valódiTárgy == null)

{...} else valódiTárgy.Kérés(); }

}

ValódiTárgy -up-|> Tárgy

Proxy -up-|> Tárgy

Proxy o-right-> ValódiTárgy

@enduml

Forráskód – Példa 2.

using System;

using System.Collections.Generic;

abstract class Faktoriális

{

public abstract long Fakt(int n); //n faktoriálist 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.TryGetValue(n, out long value)) return value;

value = f.Fakt(n);

t[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();

}

}

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 futási idejét!

**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 cégünknél, hogy szeretné az interneten keresztül felügyelni a gépek állapotát, mint például tudni, hogy mennyi golyó van még benne. A megvalósítási megbeszélésen egyik kollégánk megemlítette, hogy Ő volt egy csapatépítő tréningen, ahol az esti tábortűznél egy nagyszakállú, bölcs és kellőképpen öreg programozó mesélt nekik a proxyról és hogy az pont valami ilyesmire 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) eljuttassa 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 rejt még meglepetést az arra sétálóknak. Meghívja a MennyiGolyo() függvényt, majd a kapott eredményt visszajuttatja 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.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a 3 legegyszerűbb szerkezeti tervezési mintát, úgy mint: Illesztő, Díszítő, Helyettes. A szerkezeti tervezési minták fő feladata, hogy megmutassák, hogyan kell használni az objektum-összetételt, hogy az elvárásainknak megfelelő futási ideje szerkezeteket tudjuk létrehozni.

Ellenőrző kérdések:

1. Kérdés: Melyik tervezési mintát kell használni, ha egy értékes objektumhoz szabályozott hozzáférést szeretnék biztosítani?

Válasz: Helyettes (angolul: Proxy).

1. Kérdés: A három tervezési minta közül melyik használ átlátszó becsomagolást: Illesztő, Díszítő, Helyettes?

Válasz: A Díszítő (angolul: Decorator) és a Helyettes (angolul: Proxy).

1. Kérdés: Mikor lesz egy becsomagolás átlátszó?

Válasz: Az átlátszó becsomagoláshoz kell HAS-A kapcsolat, azaz becsomagolás, és IS-A kapcsolat is, azaz a becsomagolónak és a becsomagoltnak kell, hogy legyen közös felülete.

1. Kérdés: Miért marad a karácsonyfa díszítés után is karácsonyfa?

Válasz: Mert a Díszító (angolul: Decorator) tervezési minta átlátszó becsomagolást használ.

1. Kérdés: Az Illesztő tervezési minta által használt becsomagolás miért átlátszatlan?

Válasz: Mert a becsomagolás teljesen elrejti a becsomagolt objektum felületét, habár minden viselkedést a becsomagolt viselkedésére kell visszavezetni.

1. Kérdés: Milyen tervezési mintát kell használni gyorsítótár (angolul: cache) megvalósításához?

Válasz: Helyettes (angolul: Proxy) tervezési mintával lehet ezt a feladatot jól megoldani.

1. Kérdés: Mit jelent a helyettes állatkert (angolul: Proxy Zoo) kifejezés?

Válasz: Azt jelenti, hogy helyettesből nagyon sok fajta van: gyorsítótár, okos referencia, távoli metódushívás, stb.

## Lecke 11.: Viselkedési tervezési minták

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk néhány viselkedési tervezési mintával egy-egy példán keresztül.

Szükséges eszközök, források: A GOF könyv. A Head First Design Patterns könyv.

Feldolgozási idő: 50 perc.

Témakörök: Viselkedési tervezési minták, Állapot, Megfigyelő, Sablonmetódus, Stratégia, Látogató.

A lecke tartalma: Ebben a leckében áttekintünk 5 fontos viselkedési tervezési mintát, úgy mint: Állapot, Megfigyelő, Sablonmetódus, Stratégia, Látogató. A viselkedési tervezési minták olyan megoldást biztosítanak, amelyek könnyen bővíthetők új viselkedéssel. Ehhez általában csak egy új alosztállyal kell bővíteni a meglévő osztály hierarchiát.

### Bevezetés

A viselkedési tervezési minták arra adnak módszert, hogy hogyan írjunk 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:

* Amit szét lehet választani, azt érdemes szétválasztani.

Különösen igaz ez az úgynevezett 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 kontroll 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 kontroll megfordítása (angolul: Inversion of Control, vagy röviden IoC) az a programozási módszer, amikor nem az üzemi kód hívja az előre megírt általános kódot, pl. egy könyvtár (angolul: library) függvényt, hanem az általános kód hívja az üzemi 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.

### 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 függenek.

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.

Példa: Az állapot tervezési mintát a következő példán keresztül mutatjuk be: Feladatunk, hogy elkészítsünk egy rendkívül egyszerű audio lejátszót. A lejátszónknak a következőképpen kell működnie: ha a lejátszó készenléti állapotban van, akkor a lejátszás gomb hatástalan, 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 audiolejátszó 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 audioforrás lenyomására, az egyes állapotoktól 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. 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 PressAudioSource() { state.PressAudioSource(this); }

}

public abstract class AudioPlayerState // állapot reprezentálása

{

// a két gomblenyomása

public abstract void PressPlay(AudioPlayer player);

public abstract void PressAudioSource(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 PressAudioSource(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 PressAudioSource(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 PressAudioSource(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 PressAudioSource(AudioPlayer player)

{

player.SetState = new StandbyState();

}

}

class Program

{

static void Main(string[] args)

{

AudioPlayer player = new AudioPlayer(new StandbyState());

player.PressPlay();

player.PressAudioSource();

player.PressPlay();

player.PressPlay();

player.PressAudioSource();

player.PressPlay();

player.PressAudioSource();

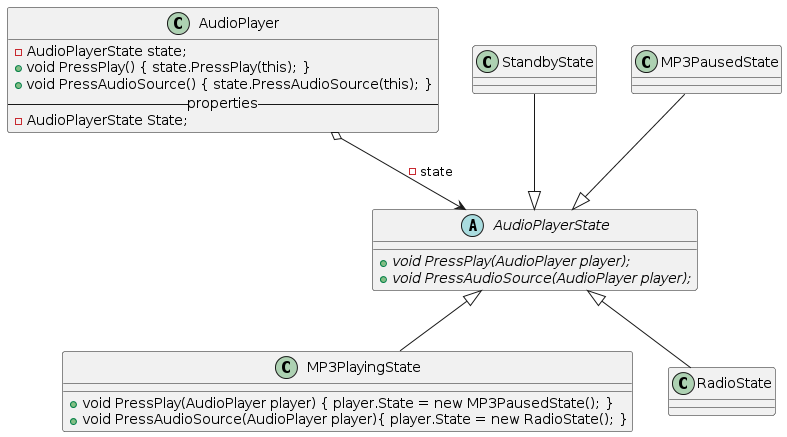
Console.ReadLine();

}

}

}

UML ábra:



Példa UML ábra az állapot tervezési mintára

PlantUML szkript:

@startuml

class AudioPlayer {

- AudioPlayerState state;

+ void PressPlay() { state.PressPlay(this); }

+ void PressAudioSource() { state.PressAudioSource(this); }

--properties--

- AudioPlayerState State;

}

abstract class AudioPlayerState {

+ {abstract} void PressPlay(AudioPlayer player);

+ {abstract} void PressAudioSource(AudioPlayer player);

}

class StandbyState { }

class MP3PlayingState {

+ void PressPlay(AudioPlayer player) { player.State = new MP3PausedState(); }

+ void PressAudioSource(AudioPlayer player){ player.State = new RadioState(); }

}

class MP3PausedState {}

class RadioState {}

AudioPlayer o--> AudioPlayerState : -state

StandbyState -down-|> AudioPlayerState

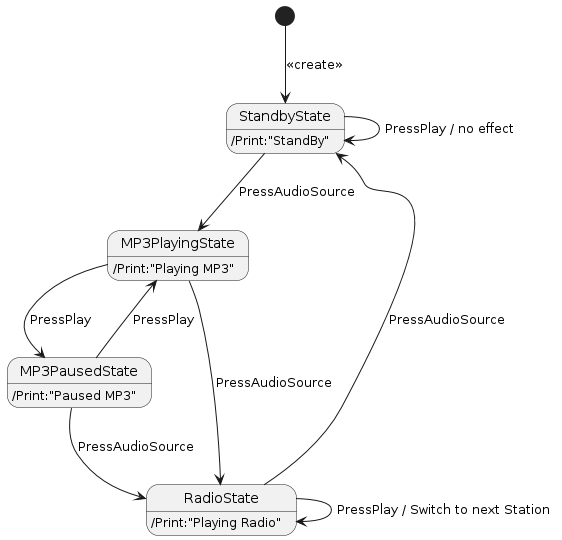
MP3PlayingState -up-|> AudioPlayerState

MP3PausedState -down-|> AudioPlayerState

RadioState -up-|> AudioPlayerState

@enduml

Mivel a fenti ábrára nem minden állapotátmenet fért el, ezért külön UML ábrán mutatjuk az állapotátmeneteket:



A példaprogram állapotgép diagrammja

PlantUML szkript:

@startuml

StandbyState : /Print:"StandBy"

MP3PlayingState : /Print:"Playing MP3"

MP3PausedState : /Print:"Paused MP3"

RadioState : /Print:"Playing Radio"

[\*] --> StandbyState : <<create>>

StandbyState --> StandbyState : PressPlay / no effect

StandbyState --> MP3PlayingState : PressAudioSource

MP3PlayingState --> MP3PausedState : PressPlay

MP3PlayingState --> RadioState : PressAudioSource

MP3PausedState --> MP3PlayingState : PressPlay

MP3PausedState --> RadioState : PressAudioSource

RadioState --> RadioState : PressPlay / Switch to next Station

RadioState --> StandbyState : PressAudioSource

@enduml

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. Kérdésünkre 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 mindent, 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 löttyöt 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.

### 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. 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

{

// értékék 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 mind observerek, van Update() metódusuk, így tudjuk őket értesíteni

public void NotifyObservers()

{

for (int i = 0; i < observers.Count; i++)

{

IObserver observer = observers[i];

observer.Update(temperature, humidity, pressure); // ez push-os

// observer.Update(this); // ez pull-os

}

}

// amikor a Weather Station-től 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 Observert,

//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);

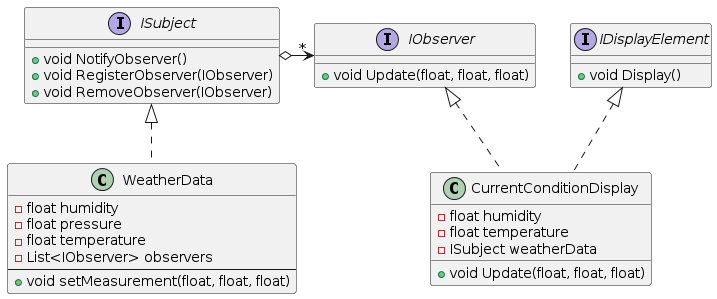
Console.ReadKey();

}

}

}

UML ábra:



Példa UML ábra a megfigyelő tervezési mintára

PlantUML szkript:

@startuml

interface IObserver {

+void Update(float, float, float)

}

interface ISubject {

+void NotifyObserver()

+void RegisterObserver(IObserver)

+void RemoveObserver(IObserver)

}

interface IDisplayElement {

+void Display()

}

class WeatherData {

-float humidity

-float pressure

-float temperature

-List<IObserver> observers

--

+void setMeasurement(float, float, float)

}

class CurrentConditionDisplay {

-float humidity

-float temperature

-ISubject weatherData

+void Update(float, float, float)

}

IObserver <|.down. CurrentConditionDisplay

IDisplayElement <|.down. CurrentConditionDisplay

ISubject <|.down. WeatherData

ISubject o-right-> "\*" IObserver

@enduml

Gyakorlófeladat: Az alábbi leírás szerint készítsünk 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 főárbóctetején 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 hírekről (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 rajongók 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 közt, 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 úgynevezett toló (félig angolul: „push”-os) megoldás lesz. Ha csak referenciát küldenénk, ahonnan a poszt lehúzható, akkor húzó (félig angolul: „pull”-os”) megoldásunk lenne.

Híroldal osztály:

* híresség: A híroldal tulajdonosának neve.
* rajongók: A rajongók listája.
* poszt: Az utolsó poszt szövege.
* Híroldal(híresség): Konstruktor, beállítja a híresség nevét. A rajongók 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: foreach(rajongó raj in rajongók) raj.Értesít(poszt); Vegye észre, hogy ez az úgynevezett műsorszórás, hiszen egy eseményről nem csak egy objektum, hanem minden érdekelt objektum értesül.

### Sablonmetódus – Template Method

A sablonmetódus (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 metódusokat.

A sablonmetódus egy olyan tervezési minta, amit akkor használunk, ha van egy általános receptünk, ami alapján több hasonló dolog is gyártható. Klasszikus példa a teafőzés és kávéfőzés, amit részletesen is ismertetünk. A sablonmetódus tervezési mintát gyakran hasonlítják össze a stratégia tervezési mintával az alábbi mondatokkal:

* Stratégia: Ugyanazt csináljuk, de másképp;
* Sablonmetódus: Ugyanúgy csináljuk, de mást.

A receptben háromféle lépés lehet:

* kötelező és közös: bármit készítünk a recepttel, ez a lépés mindig ugyanaz,
* kötelező, de nem közös: bármit készítünk a recepttel, ez a lépés szükséges, de minden esetben mást és mást 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 ősben konkrétak és azokat általában nem is szabad felülírni. Ilyen a forró ital főzésénél a vízforraló metódus.
* A kötelező, de nem közös lépések az ősben absztrakt metódusok, amit a gyermekosztályok fejtenek ki. Ilyen a forró ital főzésénél az édesítés.
* Az opcionális lépések az ősben horog (angolul: hook) metódusok, azaz van törzsük, de az üres. Ezek a metódusok virtuálisak, hogy aki akarja, az felülírhassa őket.

Mivel a gyermekosztálynak implementálnia kell minden absztrakt metódust, ezért az ilyenek kötelezőek. Igaz, hogy akár az implementáció üres is lehet. Mivel a horog (angolul: 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 horog (angolul: 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 horog (angolul: hook) metódusnak felvenni.

Érdekes megfigyelni, hogy az absztrakt ős és a gyermekosztályai 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 gyermekosztály példányán keresztül hívjuk majd a sablonmetódust, akkor a késői kötés miatt ezen metódusok helyett az őket felülíró gyermekbéli metódusok fognak lefutni.

Példa: A sablonmetódust egy példával szemléltetjük: Nincs is jobb a visszatérő ügyfélnél, és már nekünk is van ilyen. A gyártómetódus kapcsán megismert zsíroskenyérbolt visszatért. Olyan jól ment az üzlet, hogy kávét és teát is elkezdtek árusítani. Két kolléga el is kezdte a munkát, egyikőjük a teát, míg a másik a kávét kapta feladatául. 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: mindkettőjüknél volt vízforraló függvény és kitölt függvény. Ezek a dolgok mindkét 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 kollégánk 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. 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 felűlírhatónak lennie

Console.WriteLine("Vízforralás 98..99..100°C");

}

protected abstract void főz(); // Ki kell dolgoznia 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();

}

}

}

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 tésztát.");

}

// írja meg a hiányzó metódusokat

}

class TurósBuktaSűtés : BuktaSűtés

{

// írja meg a hiányzó metódusokat

}

class LekvárosBuktaSűtés : BuktaSűtés

{

// írja meg a hiányzó metódusokat

}

### Stratégia - Strategy

A Stratégia (angolul: Strategy) egy viselkedési tervezési minta, amely mindig egy, és mindig csak egy változékony metódust emel ki egy osztály hierarchiába, és amely felelősség-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 mindig 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 rekedten 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.

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 HápogásiStratégia

{

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 : HápogásiStratégia

{

public override void Hápog()

{

Console.WriteLine("HápHáp");

}

}

class RekedtHápogás : HápogásiStratégia

{

public override void Hápog()

{

Console.WriteLine("HákrrHákrrr");

}

}

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();

}

}

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());

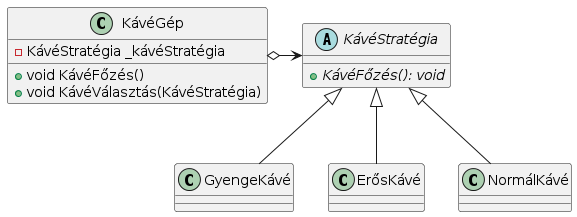
automata.KávéFőzés();

Console.ReadLine();

}

}

UML ábra:



UML ábra a stratégia tervezési mintára

PlantUML szkript:

@startuml

abstract class KávéStratégia{

+ {abstract} KávéFőzés(): void

}

class GyengeKávé{}

class ErősKávé{}

class NormálKávé{}

class KávéGép{

- KávéStratégia \_kávéStratégia

+ void KávéFőzés()

+ void KávéVálasztás(KávéStratégia)

}

KávéStratégia <|-- GyengeKávé

KávéStratégia <|-- ErősKávé

KávéStratégia <|-- NormálKávé

KávéGép o-right-> KávéStratégia

@enduml

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.

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 vezetni, illetve, hogyan lehet megakadályozni, hogy ez a hiba fellépjen a KávéGép osztály kódjának megváltoztatásával.

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 komolyságához méltón. Elkészült a kód és mi az esti tévézés 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 főnők 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.

### Látogató – Visitor

A látogató (angolul: visitor) viselkedési tervezési minta egy vagy több változékony metódust szervez ki 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 (angolul: queue) vagy verem (angolul: stack), ami már előreláthatóan nem fog változni, de lehetőséget akarunk adni arra, hogy könnyen lehessen bővíteni az adatszerkezetet feldolgozó algoritmusok sorát. Ilyenkor érdemes szétválasztani az adatszerkezetet és a feldolgozó metódusokat. Az adatszerkezetbe mindösszesen egy látogató fogadás (angolul: accept visitor) metódus kell, ami fogadja a látogatókat. A látogató fogadás metódus kódja mindig ugyanaz: v.Visit(this).

Ily módon nagyon egyszerű új látogatót írni, de ha megváltozik az adatszerkezet, akkor az összes látogató kódját frissíteni kell. Ez a látogató minta hátránya.

**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 az ágens tájékoztatja megbízóját az eredményről.

Legyen most az a feladatunk, hogy megtudjuk, hány perces a legújabb szuper mozifilm. Két lehetőségünk van:

* Vagy letöltjük a filmet a szolgáltatótól és miután lejött, megnézzük, hány perces. Ez a klasszikus kliens-szerver megoldás.
* Vagy írunk egy kis scriptet, feltöltjük 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 nem nekem kell távolról indítani, nekem csak elküldeni kell, és ha megérkezik, akkor magától elindul.

Látható, hogy a második megoldás sokkal kisebb hálózati adatforgalommal jár. Ez az ágensalapú megoldás előnye. Ugyanakkor ez a módszer távolról se terjedt el, hiszen ki szeretné, hogy mindenféle külvilágból 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, illetve 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.

A jegyzetben találunk még egy alfejezetet erről a tervezési mintáról, ahol tovább finomítjuk eddigi ismereteinket, hiszen ebben a részben még nem fejtettünk ki egy fontos fogalmat, a dupla hivatkozásfeloldás (angolul: double dispatch). 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 AcceptVisitor(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 AcceptVisitor(Visitor v)

{

v.Visit(this);

}

}

public class Levél : BFa

{

int szám;

public int Szám { get { return szám; } }

public Levél(int szám) : base()

{

this.szám = szám;

}

public override void AcceptVisitor(Visitor v)

{

v.Visit(this);

}

}

public abstract class Visitor

{

public abstract void Visit(Fa f);

public abstract void Visit(Levél f);

}

public class SumVisitor : Visitor

{

int sum = 0;

public int Sum { get { return sum; } }

public override void Visit(Fa f)

{

sum += f.Szám;

if (f.Bal != null) f.Bal.AcceptVisitor(this);

if (f.Jobb != null) f.Jobb.AcceptVisitor(this);

}

public override void Visit(Levél f)

{

sum += f.Szám;

}

}

public class MaxVisitor : Visitor

{

int max = int.MinValue;

public int Max { get { return max; } }

public override void Visit(Fa f)

{

if (f.Szám > max) max = f.Szám;

if (f.Bal != null) f.Bal.AcceptVisitor(this);

if (f.Jobb != null) f.Jobb.AcceptVisitor(this);

}

public override void Visit(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.AcceptVisitor(sumv);

fa.AcceptVisitor(maxv);

Console.WriteLine("A fában lévő számok összege: {0}", sumv.Sum);

Console.WriteLine("A fában lévő legnagyobb szám: {0}",maxv.Max);

Console.ReadLine();

}

}

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.

### Dupla hivatkozásfeloldás – Double dispatch

Azt szoktuk mondani, hogy a Látogató tervezési minta az egyetlen, amit józan paraszti ésszel nehéz kitalálni. Minden más tervezési mintát egy kis tapasztalat után és egy kis józan paraszti ésszel bárki képes újra felfedezni. Ennek oka a dupla hivatkozásfeloldás (angolul: double dispatch).

Általában csak egyszeres hivatkozásfeloldásra van szükségünk, a késő kötésre. Ha van egy polimorfikus metódus hívásunk, például: referencia.valamiHasznos(), akkor a késő kötés oldja fel, hogy az osztály hierarchiában lévő több valamiHasznos() nevű metódus közül melyik fog lefutni. Ehhez a késő kötés megnézi, hogy a referencia milyen példányra mutat, és a példány metódusa kerül meghívásra.

Ugyanakkor van egy másik eset is, amikor hivatkozásfeloldásra van szükségünk, ez a metódus túlterhelés (angolul: overloading), amikor egy metódusból több változat van más-más paraméter listával. Például, tegyük fel, hogy a valamiHasznos metódust túlterheltük:

* void valamiHasznos(int i) { … }
* void valamiHasznos(String s) { … }

Most, ha belefutunk egy hívásba: valamiHasznos(x), akkor a x kifejezés típusától függ, hogy melyik változat fog lefutni.

Dupla hivatkozásfeloldás (angolul: double dispatch) esetén mind a két fajta hivatkozásfeloldásra szükségünk van. Például a **referencia.valamiHasznos(x)** hívásnál először késői kötéssel kell meghatároznunk, hogy melyik osztály példányából kell futtatni a metódust, majd az x kifejezés típusának segítségével kell meghatároznunk, hogy melyik túlterhelt változatnak kell futnia.

Pontosan ezt használja ki a Látogató tervezési minta. Nézzük részletesen, hogy hogyan. Látogató tervezési mintát akkor használunk, ha már van egy jól kiforrott, kitesztelt adatszerkezetünk, amit már nem akarunk bővíteni. Ez a fenti példában a BFa osztály hierarchia, ami áll a BFA, a Fa és a Levél osztályokból.

Ahhoz, hogy az adatszerkezetet képes legyen feldolgozni egy látogató, ehhez minden osztály implementálja az AcceptVisitor metódust, méghozzá mindenki teljesen ugyanúgy:

void AcceptVisitor(Visitor v) { v.Visit(this); }

Azaz fogadunk egy látogatót, v, majd rögtön meghívjuk a Visit metódusát önmagunkkal: **v.Visit(this)**. Erről nagyon könnyen felismerhető a Látogató tervezési minta, és itt van a dupla hivatkozásfeloldás. Ahhoz, hogy tudjuk, hogy melyik metódusnak kell lefutnia, tudnunk kell milyen látogatóra mutat a v referencia. És hogy ezen belül a visit metódusnak melyik változata fusson le, ahhoz tudnunk kell a this típusát, ugyanis a Visit-nek az ősön kívül minden adatszerkezet osztályra fel kell készülnie:

* void Visit(Fa f)
* void Visit(Levél f)

Ha a this típusa Fa volt, akkor az első változat fut le, ha Levél, akkor a második.

Tehát a **v.Visit(this)** hívás feloldásához dupla hivatkozásfeloldás kell, azaz a Látogató tervezési minta alapja a dupla hivatkozásfeloldás.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettünk 5 fontos viselkedési tervezési mintát, úgy mint: Állapot, Megfigyelő, Sablonmetódus, Stratégia, Látogató. A viselkedési tervezési minták olyan megoldást biztosítanak, amelyek könnyen bővíthetők új viselkedéssel. Ehhez általában csak egy új alosztállyal kell bővíteni a meglévő osztály hierarchiát.

Ellenőrző kérdések:

1. Kérdés: Mi a viselkedési tervezési minták közös jellemzője?

Válasz: A viselkedési tervezési minták segítenek olyan programok létrehozásában, amelyek könnyen bővíthetők új viselkedéssel.

1. Kérdés: Melyik tervezési mintát kell használni, ha egy eseményre több lehetséges válaszreakciót is szeret adni?

Válasz: A Megfigyelő (angolul: Observer) tervezési minta pont erre jó: egy megfigyelt, ahol az esemény bekövetkezik, több megfigyelő, amely ezeket az eseményeket feldolgozza.

1. Kérdés: Hogyan jelenik meg az IoC a Sablonmetódus tervezési mintában?

Válasz: Nem az gyermek hívja az ősét, hanem az ős a gyermekét, azaz az ősben van olyan metódus, ami konkrét és ez absztrakt metódust hív, ami a gyermekben lesz kifejtve.

1. Kérdés: Mit jelent a horog (angolul: hook) metódus?

Válasz: Olyan virtuális metódus, amelynek van törzse, de a törzse üres, vagy csak egy return utasítás van benne. A Sablonmetódus (angolul: Template Method) tervezési mintában az opcionális lépések horog (angolul: hook) metódusok.

1. Kérdés: Melyik tervezési mintát érdemes használni egy olyan eszköz programjának elkészítésére, amelyen van egy gomb, de a gomb más-más funkcióval bír az eszköz más-más állapotában?

Válasz: Az Állapot tervezési minta pont ennek a feladatnak a megoldására szolgál.

1. Kérdés: Melyik tervezési mintát használjuk, ha van egy változékony metódusunk, pl.: GetÁr(), aminek a kódja gyakran változik.

Válasz: Ilyen esetben a Stratégia a jó választás. A változékony metódust kiemeljük egy új osztály hierarchiába, és mindig csak egy új osztályt adunk hozzá, amikor egy új változatot kell írni.

1. Kérdés: Melyik tervezési mintára mondják, hogy józan paraszti ésszel nehezen kitalálható?

Válasz: Majd minden forrás a Látogató (angolul: Visitor) tervezési mintát jelöli meg, mint a legnehezebben megérthetőt. Ennek oka, hogy dupla hivatkozásfeloldást technikát alkalmaz.

1. Kérdés: Mit mond ki a szétválasztás elve?

Válasz: Az elv kimondja, hogy amit szét lehet választani, azt érdemes is szétválasztani.

1. Kérdés: Milyen előnyei vannak az Állapot viselkedési tervezési mintának?

Válasz: Az előnyök közé tartozik, hogy egységbe zárja az állapotfüggő viselkedést, könnyen új állapotokat vezethetünk be, áttekinthetőbb kódot eredményez, és az állapot objektumokat megoszthatjuk.

1. Kérdés: Mi a hátránya az Állapot viselkedési tervezési mintának?

Válasz: A hátránya, hogy nő az osztályok száma, ezért csak indokolt esetekben érdemes használni.

1. Kérdés: Hogyan foglalható egy mondatba a Stratégia és a Sablonmetódus különbsége?

Válasz: Stratégia esetén ugyanazt csináljuk, de másképp; Sablonmetódus esetén ugyanúgy csináljuk, de mást.

1. Kérdés: Milyen előnyei vannak a Stratégia tervezési mintának az OCP elv szempontjából?

Válasz: A Stratégia tervezési minta segít az OCP elv betartásában, mivel lehetővé teszi új viselkedések hozzáadását a rendszerhez anélkül, hogy a meglévő osztályokat módosítani kellene. Ezáltal a rendszer zárt marad a módosítások előtt, de nyitott új funkciók hozzáadására.

1. Kérdés: Milyen helyzetekben használjuk a látogató tervezési mintát?

Válasz: A látogató tervezési mintát akkor használjuk, ha már van egy kiforrott adatszerkezetünk, de lehetőséget akarunk adni arra, hogy könnyen lehessen bővíteni az adatszerkezetet feldolgozó algoritmusok sorát.

1. Kérdés: Mi a homokozó (angolul: sandbox) szerepe az ágensalapú programozásban?

Válasz: A homokozó egy virtuális szerver, amely korlátozza az ágens által elérhető erőforrásokat és funkcionalitást, így minimalizálva a biztonsági kockázatokat.

1. Kérdés: Mi a dupla hivatkozásfeloldás (angolul: double dispatch)?

Válasz: A referencia.valamiHasznos(x) metódushívás feloldásához fel kell oldani, hogy a referencia hova mutat, ez a késői kötés. Továbbá, ha a valamiHasznos egy túlterhelt metódus, akkor az x kifejezés típusa alapján el kell dönteni, hogy melyik változatát kell hívni.

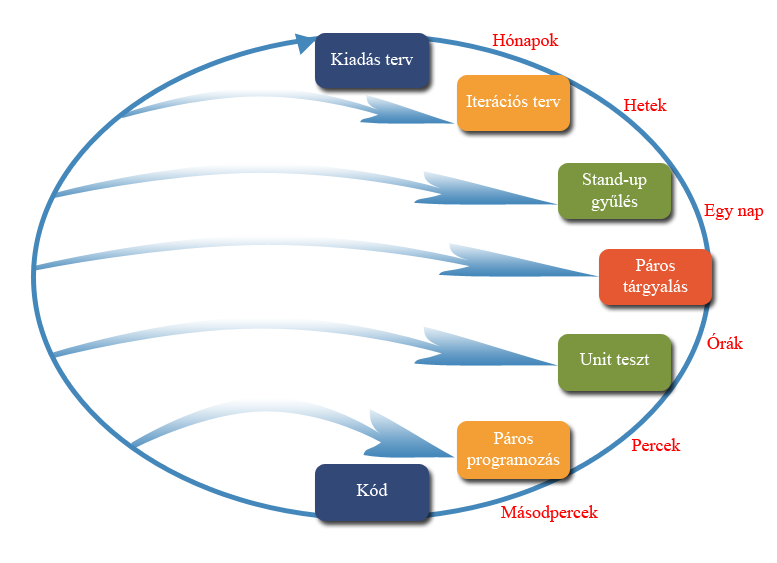
# Practices: Jól bevált módszerek

A jól bevált módszerek (angolul: best practices) fő forrása az extrém programozás (angolul: eXtreme Programming, röviden: XP) nevű Agilis módszertan. Az XP akkor használható sikeresen, ha megrendelő és a szoftverfejlesztő cég akár napi szinten, vagy még sűrűbben tud egyeztetni. Ez lényegében csak akkor fordul elő, ha egy cégen belül egy osztály megrendel a cég saját informatikai osztályától egy belső fejlesztést. Emiatt maga az extrém programozás nem terjedt el, de elterjedtek a módszerei, mint úgynevezett jól bevált módszerek.

Ennek az az oka, hogy az extrém programozás attól extrém, hogy ezen a módszertanon belül, ha valamit csinálunk, akkor extrém mód jól csináljuk, illetve, ha valamit nem csinálunk, akkor azt extrém mód nem csináljuk. Az extrém programozás előírja például, hogy csináljunk tesztelést, ennek megfelelően extrém jól csináljuk a tesztelést, úgynevezett Tesztvezérelt fejlesztést (angolul: Test Driven Development – TDD) csinálunk. És miért pont Tesztvezérelt fejlesztést? Mert a sokéves tapasztalat szerint ez a legextrémebb, legjobb tesztelés! És ha ez a legjobb, akkor ezt érdemes lenne máshol is alkalmazni! Ezen gondolatmenet és tapasztalat útján terjedtek el az extrém programozás módszerei, és lettek jól bevált módszerek.

Persze nem minden XP módszer terjedt el. Például a páros programozás az XP egyik módszere, és aki kipróbálta, az dicsérni is szokta, mégsem terjedt el széles körben a szoftvercégek gyakorlatában.

Nézzük először az XP módszereit, aztán vizsgáljuk meg részletesebben azokat, amik széles körben elterjedtek:



Az XP jól bevált módszerei

Az XP módszertan módszerei:

* Páros programozás (angolul: pair programming): 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.
* Tesztvezérelt fejlesztés (angolul: test driven development): Már a metódus elkészítése előtt megírjuk a hozzá tartozó egységteszteket, 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 érthető, 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 félreértés.
* Kódszépítés (angolul: refactoring): A már letesztelt, működő kódban lehet szépíteni, az esetleg lassú, rugalmatlan, vagy egyszerűen csak csúnya részeket. A kódszépítés előfeltétele, hogy legyen sok egységteszt. A szépítés során nem szabad megváltoztatni a kód funkcionalitását, de a szerkezet, pl. egy metódus törzse, szabadon változtatható. A szépítés után minden egységtesztet le kell futtatni, nem csak a megváltozott kódhoz tartozókat, hogy lássuk, a változások okoztak-e hibát.

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

A jól bevált módszerek másik forrása a Robert C. Martin nevéhez köthető Tiszta kód (angolul: Clean Code) nevű könyv és videósorozat, ami az Agilis szoftverfejlesztés filozófiáját tárgyalja sok-sok könyvön és videón keresztül, anélkül, hogy valamelyik konkrét Agilis módszertan mellé letenné a voksot.

Ebben a fejezetben kisebb-nagyobb részletességgel tárgyaljuk azokat a módszereket, amik szerintünk jól bevált módszernek tekinthetők.

## 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 témával Robert C. Martin Tiszta kód című könyve.

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 is, ha saját magunk fejlesztjük tovább, akkor is egy idegen programozónak írjuk a kódunkat, mert gyakran megfigyelhető, hogy egy néhány hónappal korábban írt programról már nem ismerjük fel, hogy azt saját magunk 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, illetve használhatunk öndokumentáló megjegyzést a szerződések rögzítésére (lásd: LSP), de nem lehet benne olyan megjegyzés, ami egy mezőt vagy egy kódrészletet magyaráz. Ha ilyen van a kódunkban, azzal beismerjük, hogy nem használunk beszédes változó neveket, nehezen érthető részek vannak a kódban, 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. Egy ilyen megjegyzést senki se mer törölni, mert lehet, hogy egy későbbi hibajavításhoz fontos lesz.

A tiszta kód ajánlást ad a metódusok méretére. Egy metódus legyen 5-6 soros, csak 1-2 if utasítást tartalmazzon, 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 (belsőállapot-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 használjunk logikai paramétert, illetve ne használjunk kimeneti paramétert.

### Cserkészszabály

A tiszta kód egyik alapelve a cserkészszabály (angolul: boy scout rule), amely így szól:

* Mindig hagyd a táborhelyet tisztábban, mint ahogy találtad.

Szoftverfejlesztés esetén ez 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.

### 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 cégünket, végül tönkre megyünk.

Rothadó kód egy szép tiszta kódból lesz. A fő elvünk kimondja, 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 mindig 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 kezelni, de az a változás sohasem következik be.
* Homályosság (angolul: opacity): Nehezen olvasható, nehezen érthető kód.

## Lecke 12.: Egységteszt

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk az egységteszt alapjaival, illetve, hogy az egyes IDE-kben hogyan kell egységtesztet írni.

Szükséges eszközök, források: Visual Studio, IntelliJ

Feldolgozási idő: 20 perc.

Témakörök: Egységteszt készítése IntelliJ-ben JUnit5 csomagot használva, Egységteszt készítése Visual Studio-ban.

A lecke tartalma: Ebben a leckében áttekintjük, hogyan kell egységtesztet, vagy más szóval unit-tesztet írni két nagyon elterjedt integrált fejlesztési környezetben (angolul: IDE). Ezek az IntelliJ és a Visual Studio. Ebben a leckében nem térünk ki a JUnit5, illetve a C# UnitTest minden részletére, csak az alapokat mutatjuk be.

### Bevezetés

Az egységteszt, vagy más néven unit-teszt a metódusok tesztje. Egy egységtesztben 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, és ha ugyanaz, akkor az egységteszt zöld lesz, ha nem, akkor piros.

Egy példa egységteszt, először Javában:

@Test

public void TestMethod1() {

SzámolóGép target = new SzámolóGép();

int a = 2;

int b = 2;

int expected = 4;

int actual = target.Összead(a,b);

Assertions.assertEquals(expected,actual);

}

Illetve C#-ban ugyanaz:

[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 kimenetet (a kódban: actual). Majd a kettőt összehasonlítjuk.

Az egységtesztek legnagyobb előnye, hogy egyszerre el tudjuk indítani az összes tesztesetet, egyetlen gombnyomással. Így a regressziós teszt nagyon egyszerűvé válik, hiszen egy-egy változtatás után csak le kell futtatnunk az összes egységtesztet, ami csupán egy gombnyomás!

### Egységteszt készítése JUnit5 és Maven használata esetén

Ahhoz, hogy egységtesztet, vagy más néven unit-tesztet tudjunk írni Javában, ahhoz kell egy ezt támogató keretrendszer. Napjainkban a legelterjedtebb keretrendszer a JUnit5, amit külső függőségként kell hozzáadni a projekthez. Ezt talán legegyszerűbb a Maven építés-automatizálási segédeszköz (angolul: build-automation utilities) használatával megtenni.

Maven esetén a pom.xml nevű állományba kell felsorolni a külső függőségeket. JUnit5 használatához ezeket a sorokat kell beszúrni a pom.xml-be:

<dependencies>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.9.2</version>

<scope>test</scope>

</dependency>

</dependencies>

Ezek után egy frissítést kell futtatni. IntelliJ IDE esetén az a pom.xml-ben jobb klikk (helyi menü) -> Maven -> Reload project menüvel lehetséges, vagy a pom.xml ablak jobb felső sarkában megjelenő Reload gomb segítségével. Ezután már a következő kis program futtatható lesz:

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

import org.example.SzámolóGép;

public class Test1 {

@Test

public void TestMethod1() {

SzámolóGép target = new SzámolóGép();

int a = 2;

int b = 2;

int expected = 4;

int actual = target.Összead(a, b);

assertEquals(expected, actual);

}

}

Figyeljük meg, hogy szükség van néhány import-ra. Az első import rögtön nagyon érdekes, hiszen ez egy osztály statikus metódusát importálja import static kulcsszavak segítségével. Az első import lehetővé teszi, hogy az elvárt és az aktuális érték összehasonlítható legyen assertEquals segítségével.

A második import ahhoz szükséges, hogy használhassuk a @Test annotációt. Ez az az annotáció, ami egy metódust unit-tesztté tesz. A harmadik import ahhoz kell, hogy a SzámolóGép osztályt tesztelni tudjuk.

Ezen túl ebben a kis példában megvan minden, ami egy klasszikus unit-teszthez kell:

* target,
* fix input paraméterek,
* elvárt érték,
* aktuális érték,
* a kettő összehasonlítása.

### Egységteszt készítése Visual Studio 2013 vagy újabb esetén

Ahhoz, hogy egységtesztet, vagy más néven unit-tesztet tudjunk 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 UnitTestProject1-en belül a References-en jobb klikk, majd Add Reference… -> Solution –> Projects, az Application neve melletti jelölő négyzetbe pipa, majd OK.
3. A UnitTest1.cs fájlba beírni egy using utasítást, hogy hivatkozni tudjunk 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:

<https://learn.microsoft.com/en-us/visualstudio/test/getting-started-with-unit-testing?view=vs-2022>.

### Egységteszt készítése Visual Studio 2008 és 2010 esetén

Egységtesztet, 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 egységtesztet szeretnénk készíteni.

Egy példa egységteszt amelyet a fenti eljárással generáltunk a maximum értéket 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.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük, hogyan kell egységtesztet, vagy más szóval unit-tesztet írni két nagyon elterjedt integrált fejlesztési környezetben (angolul: IDE). Ezek az IntelliJ és a Visual Studio. Ebben a leckében nem tértünk ki a JUnit5, illetve a C# UnitTest minden részletére, csak az alapokat mutattuk be.

Ellenőrző kérdések:

1. Kérdés: Java esetén melyik a legelterjedtebb unit-tesztelési környezet?

Válasz: Jelen pillanatban a JUnit5 csomag a legelterjedtebb.

1. Kérdés: Milyen 5 pontból áll általában egy unit-teszt?

Válasz: Target, fix input paraméterek, elvárt érték, aktuális érték, a kettő összehasonlítása.

1. Kérdés: Maven használata esetén hova kell írni a JUnit5 függőségeit?

Válasz: A POM.XML fájlba kell írni a <dependencies> részbe.

## Lecke 13.: Tesztvezérelt fejlesztés – Test Driven Development – TDD

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk a TDD alapjaival, alapelveivel, jelentőségével, miszerint a TDD a szoftverfejlesztés kettős könyvelése.

Szükséges eszközök, források: Robert C. Martin Tiszta kód című könyve.

Feldolgozási idő: 100 perc.

Témakörök: TDD alapelvei, Piros – Zöld – Piros, Piros – Zöld – Kék – Piros, Kettős könyvelés.

A lecke tartalma: Ebben a leckében áttekintjük a Tesztvezérelt Fejlesztés (angolul: Test Driven Development - TDD) és a Tiszta Kód közötti kapcsolatot. A Tiszta Kód célja könnyen olvasható, könnyen karbantartható forráskód írása. Előbb-utóbb minden forráskód elkezd bűzölögni, majd rothadni. Ez ellen néha-néha refaktorálnunk, szépítenünk kell a kódot. A refaktorálás előfeltétele, hogy legyen sok-sok unit-tesztünk, hogy legyen bátorságunk hozzányúlni a kódhoz. Ebben segít a TDD, amelynek alapelvei: 1.: Ne írj üzemi kódot, kivéve, hogy egy hibás tesztet kielégíts. 2.: Csak annyi tesztet írj, amely elegendő egy hiba demonstrálásához. 3.: Csak annyi üzemi kódot írj, hogy kielégítsd a tesztet. A TDD másik neve: Piros – Zöld – Piros, illetve, ha kódszépítési lépés is van, akkor: Piros – Zöld – Kék – Piros.

### Bevezetés

A tesztvezérelt fejlesztés (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 egységtesztet, vagy más néven unit-tesztet, hanem épp fordítva, először írjuk az egységtesztet, úgy, hogy a tesztelt metódusnak még a feje sincs kész, majd csak ezután írjuk meg a metódust.

A TDD még ettől is továbbmegy, azt írja elő, hogy a metódusból csak annyit írjunk meg, hogy épp átmenjen a legutoljára megírt egységteszten. Ezután írjunk egy új egységtesztet a már félkész metódushoz, ami egy eddig le nem fedett esetet tesztel, aztán írjuk meg a metódusba ezt az esetet. És így tovább, amíg minden estet le fedünk tesztesettel, illetve kóddal a metódusban.

Tehát a TDD lépései így foglalhatók össze:

* Egységtesztet írunk először, csak utána metódust.
* Minden egységteszt az eddig nem tesztelt lehető legegyszerűbb esetet írja le.
* A metódusból csak annyit írunk meg, hogy épp átmenjen az összes egységteszten.

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 üzemi kódot, kivéve, hogy egy hibás tesztet kielégíts.
* „Write only enough of a test to demonstrate a failure.” Azaz: Csak annyi tesztet írj, amely elegendő egy hiba demonstrálásához.
* „Write only enough production code to pass the test.” Azaz: Csak annyi üzemi 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 egységtesztet.

Ha betartjuk a TDD ajánlásait, akkor annak rengeteg pozitív hatása van. Lássuk ezeket:

* Így nem felejtünk el egységtesztet 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 szívvel merünk hozzányúlni 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 spagetti kóddá, vagy ami még rosszabb, bűzhödt, rothadó kóddá.
* A tesztek a kód nagy részét lefedik, így a program magas minőségében biztosak lehetünk.
* Sokkal kevesebb időt kell hibakereséssel, nyomkövetéssel töltenünk.
* Az egységteszt a legjobb, programozóknak szóló dokumentáció, így könnyebb megérteni a kódunkat.

Továbbá, minden programozónak két feje, azaz gondolkodási módja van. A programozó fejünk így gondolkodik: „Én egy programozó zseni vagyok, rövid és gyors kódot írok!”. A tesztelő fejünk így gondolkodik: „Mindegy, hogy van implementálva, csak működőképes legyen, tudjam tesztelni.”

A TDD egyik nagy előnye, hogy először a sokkal pragmatikusabb tesztelő fejünkkel gondolkodunk. Például szem előtt tartjuk, hogy az osztályaink legyenek publikusak, hiszen csak publikus osztályokat lehet tesztelni. Lássuk be, a programozó fejünk még sose gondolt arra, hogy egy osztály publikus legyen vagy sem.

### TDD és a tiszta kód

A tiszta kód alapja a TDD. Ha TDD-t használunk, akkor van sok-sok egységtesztünk, ami bátorságot ad, hogy hozzányúljunk a kódhoz. Ha látjuk, hogy kezd romlani a kódunk, akkor merjük azt refaktorálni. Emiatt a TDD a tiszta kód alapja!

Habár a fenti bekezdés igaz és velős, mégis ez az alfejezet nagyon rövid lett, ezért megkérdeztük ChatGPT 3.5 AGI-t, hogy fejtse ki a véleményét ebben a témában. A prompt ez volt:

Kérlek, egyetemi jegyzet stílusban írj erről a témáról: TDD és a tiszta kód kapcsolata.

ChatGPT 3.5 válasza (néhány iteráció után):

### Tesztvezérelt Fejlesztés (TDD) és Tiszta Kód Kapcsolata

A Tesztvezérelt Fejlesztés (Test-Driven Development, TDD) és a Tiszta Kód elve között szoros kapcsolat van, amely alapvető fontosságú a hatékony és fenntartható szoftverfejlesztéshez. A TDD egy olyan fejlesztési módszer, amelyben a fejlesztő először ír tesztet, majd csak ezután ír üzemi kódot, hogy a teszt sikeresen lefusson. Ez a módszer fokozza a fejlesztés folyamatának fókuszát a funkcionalitásra és a végtermék elvárt viselkedésére.

A TDD gyakorlása során a tesztek szorosan meghatározzák a kívánt funkciókat és elvárt viselkedést, ezáltal irányítják a kód implementációját. Ez a folyamat arra ösztönzi a fejlesztőket, hogy csak azt a kódot írják meg, ami szükséges a tesztek sikeres lefuttatásához, így minimalizálva a fölösleges kód mennyiségét és javítva a kód olvashatóságát.

A Tiszta Kód elve a szoftverfejlesztésben arra törekszik, hogy a kód könnyen érthető, karbantartható és kiegészíthető legyen. Ez magában foglalja a jól strukturált, jól elnevezett osztályok és metódusok használatát, valamint az egyszerű és kifejező kód írását. A tiszta kód elvei segítenek megőrizni a kód hosszú távú érthetőségét és fenntarthatóságát.

A TDD és a Tiszta Kód szoros kölcsönhatása azt eredményezi, hogy a TDD által generált tesztek arra ösztönzik a fejlesztőket, hogy a kód tisztasági elveit kövessék. A tesztek meghatározzák a kód funkcionalitását és elvárt viselkedését, míg a tiszta kód elvei segítenek abban, hogy ezeket a célokat hatékonyan és olvasható módon valósítsák meg. Ennek eredményeként a TDD és a Tiszta Kód együttműködése elősegíti a magas minőségű, karbantartható és kiegészíthető kód létrehozását, ami hosszú távon előnyös a szoftverfejlesztés során.

---

ChatGPT válaszát elolvasva, ami alapvetően jó, mégis kicsit csalódtunk, hiszen, nem emelte ki a lényeget, miszerint a unit-tesztek az az életbiztosításunk, ami ahhoz kell, hogy legyen bátorságunk a kódot tisztán tartani, refaktorálni, mielőtt az rothadó kóddá válna.

### Piros – Zöld – Piros

Tudjuk, hogy a sikertelen egységteszt, más néven unit-teszt jelzése minden keretrendszerben piros, a sikeresé zöld. Azt is tudjuk, hogy a TDD előírja, hogy a metódusból (az üzemi kódból) csak annyit írjunk meg, hogy épp átmenjen a legutoljára megírt egységteszten. Ezután írjunk egy új egységtesztet a már félkész metódushoz, ami egy eddig le nem fedett esetet tesztel, aztán írjuk meg a metódusba ezt az esetet. És így tovább, amíg minden esetet lefedünk tesztesettel, illetve üzemi kóddal a metódusban.

Azaz először lesz egy piros egységteszt, 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.

### TDD esettanulmány

Lássunk egy egyszerű példát TDD használatára. TDD segítségével fogjuk lefejleszteni a bináris fában lévő számok összegét kiszámító metódust. Minden egységteszt után közöljük az egész bináris fa osztályt.

A lenti példában, ami több körre van osztva, az egységteszteket a UnitTest1.cs fájlba kell másolni, a BFa osztályt pedig a Program.cs nevű fájlba.

**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; }

}

**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; }

}

**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;

}

}

**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;

}

}

### 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 egységtesztet, 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 konstruktorban 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;

}

### A TDD a szoftverfejlesztés kettős könyvelése

A szoftverfejlesztés nagyon érzékeny, könnyű helytelen kódsort írni. Szerencsés esetben a kódsor csak szintaktikailag helytelen. Ilyenkor a fordító kiírja, hogy ez és ez a sor hibás, 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 máris az egész elromlik. A könyvelés területén ezt a kettős könyveléssel védik 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.

### Összegzés és ellenőrző kérdések

Ebben a leckében áttekintettük a Tesztvezérelt Fejlesztés (angolul: Test Driven Development - TDD) és a Tiszta Kód közötti kapcsolatot. A Tiszta Kód célja könnyen olvasható, könnyen karbantartható forráskód írása. Előbb-utóbb minden forráskód elkezd bűzölögni, majd rothadni. Ez ellen néha-néha refaktorálnunk, szépítenünk kell a kódot. A refaktorálás előfeltétele, hogy legyen sok-sok unit-tesztünk, hogy legyen bátorságunk hozzányúlni a kódhoz. Ebben segít a TDD, amelynek alapelvei: 1.: Ne írj üzemi kódot, kivéve, hogy egy hibás tesztet kielégíts. 2.: Csak annyi tesztet írj, amely elegendő egy hiba demonstrálásához. 3.: Csak annyi üzemi kódot írj, hogy kielégítsd a tesztet. A TDD másik neve: Piros – Zöld – Piros, illetve, ha kódszépítési lépés is van, akkor: Piros – Zöld – Kék – Piros.

Ellenőrző kérdések:

1. Kérdés: Mit jelent a Piros – Zöld – Piros?

Válasz: A Piros – Zöld – Piros a TDD egy másik neve. Az elnevezés onnan ered, hogy a TDD alapelveit betartva először egy unit-tesztet írunk, ami piros lesz. Aztán megírjuk az ezt kiegészítő üzemi kódot, így ez a unit-teszt zöld lesz. Ha még nincs kész a metódus, akkor írunk egy újabb unit-tesztet erre a metódusra, ami megint piros lesz. Ezt ismételjük, amíg el nem készül a metódus.

1. Kérdés: Mit jelent a Piros – Zöld – Kék – Piros?

Válasz: A piros a sikertelen unit teszt, a zöld a sikeres unit teszt, a kék a refaktorálás színe. Ezeket a lépéseket ismételgetjük a TDD alapelvei szerint, amíg el nem készül az adott metódus.

1. Kérdés: Miért hívjuk a TDD módszert a szoftverfejlesztés kettős könyvelésének?

Válasz: Mivel a program minden kis részletéhez van unit-tesztünk és üzemi kódunk is, ezért, a TDD a szoftverfejlesztés kettős könyvelése. Előnye: Ha vagy az egyik, vagy a másik helyen elrontjuk az üzleti logikát, akkor arról hibajelentést kapunk

1. Kérdés: Miért érdemes először a tesztet megírni, és csak utána az üzemi kódot?

Válasz: Ennek rengeteg előnye van, talán a legfontosabb, hogy így nem felejtünk el unit-teszteket írni, és így lesz sok-sok unit-tesztünk, ami a kód szépítés előfeltétele, a kód szépítés pedig ahhoz kell, hogy ne kezdjen el bűzölögni, majd rothadni a kódunk.

1. Kérdés: Miért mondják azt, hogy egy programozónak két feje van?

Válasz: Minden programozó tud programozóként és tesztelőként is gondolkodni, de a két gondolkodásmóddal alapvetően másként közelítünk meg egy szoftverfejlesztési problémát. A programozó fejünk egy csúcsminőségű megoldást szeretne, ami valószínűleg elég bonyolult is. A tesztelő fejünk egy egyszerű, könnyen tesztelhető megoldást szeretne.

1. Kérdés: A TDD módszer melyik P betűhöz tartozik: Principles, Patterns, Practices?

Válasz: A TDD egy jól bevált módszer, azaz a Practices szóhoz tartozik.

1. Kérdés: Mi a unit-tesztelés extrém változata?

Válasz: A TDD, hiszen TDD esetén unit-tesztelést csinálunk extrém jól.

## Lecke 14.: Viselkedésvezérelt Fejlesztés – Behaviour-Driven Development – BDD

### A lecke célja és tartalma

Célok, megszerezhető kompetenciák: A lecke végére megismerkedünk és gyakorlati tapasztalatot kapunk a BDD technológiában példákon keresztül.

Szükséges eszközök, források: Java SDK, IntelliJ IDEA, Maven build tool, Cucumber for Java plugin az IntelliJ-hez. Kapcsolódó weboldalak:

* [https://www.jetbrains.com/idea/download](https://www.jetbrains.com/idea/download/?section=mac)
* <https://cucumber.io/docs/installation/>
* <https://maven.apache.org/install.html>
* <https://www.oracle.com/java/technologies/downloads/>

Feldolgozási idő: 50 perc.

Témakörök: Bevezetés a felhasználói sztorik készítésbe, a BDD módszertanba, a Gherkin nyelv alapjaiba és a Cucumber használatába.

A lecke tartalma: 2000 környékén Daniel Terhorst-North szerette volna a Test-Driven Development módszertanát továbbfejleszteni, mivel úgy érezte, hogy vannak hiányosságai, melyet a fejlesztésben résztvevő három fél közti kollaboráció definiálatlansága okoz. Mind a három félnek megfelelő információval kell rendelkeznie a feladatról, ugyanazokat a folyamatokat kell követniük és lehető legrövidebb idő alatt kell a feladatot megfelelően kivitelezniük. Ennek eredményeképpen, létrehozta a JBehave eszközt a Java nyelvhez, hogy viselkedéseket (behaviour) tudjon definiálni tesztek helyett. 2006-ban megjelentetett egy ismertetőt a BDD evolúciójáról amiből teljesebb képet kaphatunk annak kialakulásáról (<https://dannorth.net/introducing-bdd/> ).

Egy User Story-ban leginkább olyan mondatokat fogalmazunk meg, melyek a „Mint ..., azért, hogy ... „ (“As a…, so that…”) sémát követik, azért hogy egységes kialakítása legyen. Rachel Davies felfedezte, hogy ezek leegyszerűsíthetőek olyan szavakra, mint “Given, When, Then” amik végül a BDD-hez használt Gherkin nyelv kulcsszavai lettek.

Egy csoport programozó elkezdte a JBehave-et Ruby nyelvre átírni, ami neve RSpec volt (https://rspec.info/about/). A fejlesztők egyike, Aslak Hellesøy külön projektbe kezdett azért, hogy a RSpec Story Runner eszközét saját ötleteivel kiegészíthesse. Készíteni szeretett volna hozzá egy sokkal jobb Story Runner verziót, végül ez lett a Cucumber.

(History of BDD: <https://cucumber.io/docs/bdd/history/>)

### Bevezetés

A viselkedésvezérelt fejlesztés (angolul: Behaviour-Driven Development - BDD) a TDD édestestvére, habár nem a tesztelésre helyezi a fő fókuszt, hanem a „Három Amigó” közös munkájára.

A TDD szoftverfejlesztési technika összefüggésben áll az extrém programozással (angolul: eXtreme Programming - XP), az automatizált tesztekkel, melyek első teszteléskor még hibát jeleznek, hiszen az üzleti logika nincs készen. De a fejlesztés előrehaladtával, ahogy a kódunk kiteljesedik, a tesztek is sikerrel zárulnak.

A teszt alapú technikák előnye, hogy a teszteket a későbbiekben is fel tudjuk használni annak ellenőrzésére, hogy változtatásaink nem rontják el az addigi munkánkat, vagyis a programunk továbbra is hiba nélkül fut. A nehézséget leginkább a tesztek megfelelő megfogalmazása okozza, azaz meg kell tudnunk határozni a teljesítés kritériumait, amit az üzleti logika megvalósít. Amennyiben megköveteljük a mindenre kiterjedő tervezést, akkor megsértjük az agilis módszertan iteratív megközelítését. Ugyanakkor rendelkezésre állnak olyan eszközök, mint a felhasználói történetek (angolul: User Story), ahol mindennek az alapját le tudjuk fektetni. A BDD ezen definíciókat használja és konvertálja olyan viselkedésekké, amit a kódunknak meg kell valósítania, tehát a tényleges és elvárt viselkedés összehasonlítható.

Megtévesztő is lehet a TDD és a BDD kapcsolata, ugyanakkor látnunk kell azt, hogy a BDD nem szimplán tesztek írásáról szól, azok csak eszközök ahhoz, hogy a „Három Amigó” (angolul: Three Amigos) munkáját összekösse. A „Három Amigó”

* az üzleti elemző, esetleg a termék tulajdonos (angolul: business analyst / product owner),
* a fejlesztő (angolul: developer), és
* a tesztelő (angolul: tester).

A BDD feladata, hogy összekösse a „Három Amigót” és olyan automatizált folyamatot alkosson, amivel a követelmények lefektetése után implementálni tudják azt a viselkedést, amit a felhasználói sztorikban lefektettek, és ennek tesztelésére legyenek automatikusan generált tesztek.

### Mi is az a BDD?

A 2.2.8. Felhasználói sztori fejezetben megjelenik egy példa az epic-re, ami felbontható kisebb felhasználói történetekre (angolul: user story):

„Éva és Peti elmennek autót vásárolni.

Évának megtetszik egy piros sportautó.

Zoli, az eladó, rögzíti Éva adatait az *érdeklődő adatai rögzítése* képernyőn.

Zoli még megmutat néhány autót Évának, de Évának annyira tetszik a piros sportautó, hogy végül meg is veszi.

Ehhez Zoli elkéri Éva adatait, amit rögzít a *vásárló adatainak rögzítése* menüpont alatt, majd kinyomtatja a számlát.

Éva fizet, és megkapja a piros sportautó kulcsait.”

| „Éva és Peti elmennek autót vásárolni.  Évának megtetszik egy piros sportautó.  Zoli, az eladó, rögzíti Éva adatait az érdeklődő adatai rögzítése képernyőn.  Zoli még megmutat néhány autót Évának, de Évának annyira tetszik a piros sportautó, hogy végül meg is veszi.  Ehhez Zoli elkéri Éva adatait, amit rögzít a vásárló adatainak rögzítése menüpont alatt, majd kinyomtatja a számlát.  Éva fizet, és megkapja a piros sportautó kulcsait.” |
| --- |

Ez a megfogalmazás még nem egészen fogható fel user story-nak, hiszen a megjelenő felhasználók szerepköre nincs konkretizálva, van olyan, ahol egyszerre több felhasználó is megjelenik egyező szerepkörrel, vagy összevon akár több történetet is. Ugyanakkor ez az a leírás, amit a megrendelőtől megkaphatunk és pont emiatt nagyon hasznos, hiszen ebből már kialakíthatók a felhasználói történetek. Ezek pedig rövid ismertetést tartalmaznak azon viselkedésről, amit a megrendelők tapasztalni szeretnének a programunk működése során. Ezt a leírást általában az ügyféllel kapcsolatot tartó termék tulajdonos (angolul: product owner) készíti el az ügyféllel, vagy kéri meg azt ennek elkészítésére.

A megkapott leírást már át lehet alakítani felhasználói sztorikká. A sztorik pedig egy-egy feladatra koncentrálnak, konkrét szerepkörökkel jelennek meg bennük felhasználók és tartalmazzák a legfontosabbat, a cél definiálását. Viszont ezek könnyű feldolgozhatóságához egységesíteni kell megfogalmazásukat. Tehát a fenti példát így lehet átfogalmazni:

„Éva, mint vevő el akar menni egy autókereskedőhöz, azért hogy autót vásároljon.

Zoli, mint eladó el akarja menteni az érdeklődő Éva adatait azért, hogy rögzítse az érdeklődő adatai adatbázisban későbbi kapcsolatfelvételre.

Zoli, mint eladó el akarja menteni az érdeklődő Éva kívánságait, azért hogy a megfelelő típust tudja ajánlani számára.

Zoli, mint eladó meg akar mutatni néhány autót Évának, azért hogy nagyobb legyen a választék.

Éva, mint vevő a piros sportautót akar választani, azért hogy olyan autója legyen mint a kedvenc színe.

Zoli, mint eladó, át akarja tenni az érdeklődő adatai adatbázisból a vásárló adatbázisba Éva adatait, azért hogy számlát készíthessen.

Éva, mint vevő számlát akar kapni, azért hogy kifizethesse autóját.”

| „Éva, mint vevő el akar menni egy autókereskedőhöz, azért hogy autót vásároljon.  Zoli, mint eladó el akarja menteni az érdeklődő Éva adatait azért, hogy rögzítse az érdeklődő adatai adatbázisban későbbi kapcsolatfelvételre.  Zoli, mint eladó el akarja menteni az érdeklődő Éva kívánságait, azért hogy a megfelelő típust tudja ajánlani számára.  Zoli, mint eladó meg akar mutatni néhány autót Évának, azért hogy nagyobb legyen a választék.  Éva, mint vevő a piros sportautót akar választani, azért hogy olyan autója legyen mint a kedvenc színe.  Zoli, mint eladó, át akarja tenni az érdeklődő adatai adatbázisból a vásárló adatbázisba Éva adatait, azért hogy számlát készíthessen.  Éva, mint vevő számlát akar kapni, azért hogy kifizethesse autóját.” |
| --- |

Az egységesítés követi a következő felépítést:

“Mint … , azt akarom hogy …, azért mert … “

“As a [person], I [want to], [so that].”

Lehet elsőre ez a felépítés nem olyan olvasmányos, mint a megrendelő által megfogalmazott eredeti szöveg, de még mindig eléggé olvasható és már konkretizálja felhasználói szerepekre a történeteket, továbbá megmagyarázza miért van ezekre szükség. Készítői kapnak egy sablont és meg kell határozniuk minden részét annak. Ha bármelyik rész kimarad, mert nem tudják leírni, akkor gyanús, hogy nincs is szükség annak megvalósítására. Tehát egy előzetes szűrést végez a fejlesztők számára, hiszen ezeken először az üzleti elemző (angolul: business analyst), vagy a termék tulajdonos (angolul: product owner) dolgozik.

Az egységesítés segíti a fejlesztők munkáját azzal, hogy mindig ugyanazt a három szakaszt kell keresniük, nem kell más módon értelmezniük a leírtakat. Kicsit még dolgozzuk át az előbbi megfogalmazást:

| „Adott, hogy Éva egy érdeklődő.  Amikor Éva autót akar választani,  Akkor Éva belép az autókereskedőhöz.    Adott, hogy Zoli egy eladó.  És rendelkezik az adatbázis eléréshez jogosultságokkal.  Amikor érdeklődő adatait megkapja,  Akkor Zoli elmenti azokat az adatbázisba.    Adott, hogy Zoli egy eladó.  És fel tudja venni az érdeklődő kedvenc színét az adatbázisba.  Amikor megkap egy keresett színt,  Akkor Zoli elmenti azt az adatbázisba.    Adott, hogy Zoli egy eladó.  És ki tudja keresni az adott színű autókat az adatbázisból.  Amikor kiválaszt egy színt,  Akkor a rendszer kilistázza az adott színű autókat.    Adott, hogy Éva egy vevő.  És kedvenc színe a piros.  Adott, hogy Zoli egy eladó.  És rendelkezik az adatbázis eléréshez jogosultságokkal.  Amikor Zoli megadja a piros színt,  Akkor a rendszer csak a piros színű autókat listázza.    Adott, hogy Zoli egy eladó.  Amikor egy érdeklődő vásárol,  Akkor az adatai átkerülnek a vásárló adatbázisba    Adott, hogy Éva egy vevő.  Amikor Éva fizetni akar,  Akkor számlát kell készíteni a számára.” |
| --- |

Ez a leírás is hasonlít az eredeti “Mint … , azt akarom hogy …, azért mert … “ megfogalmazáshoz, de sokkal következetesebb szavakat és lehetőségeket biztosít az egyes felhasználói helyzetek (történetek) leírására. Ha pedig mindezeket a kulcsszavakat angolra fordítjuk, akkor nemzetközileg használható megfogalmazást alakíthatunk ki, amik, mint metódus nevek jelennek meg tesztjeinkben.

Például:

Scenario: sales person filters cars

Given that Zoli is a sales person

And has ROLE\_SALES role to access the database

When Zoli supply red colour

Then the system lists only red cars.

| Scenario: sales person filters cars  Given that Zoli is a sales person  And has ROLE\_SALES role to access the database  When Zoli supply red colour  Then the system lists only red cars. |
| --- |

Ezt az átalakítást általában már a fejlesztő (angolul: developer) végzi, de az üzleti elemző (angolul: business analyst) vagy a termék tulajdonos (angolul: product owner) validálja azt az eredeti kívánalmak szerint. Ez az ellenőrzés lényeges mindenki számára, mert lehetővé teszi, hogy az utolsó automatizált lépés olyan eredményt hozzon, ami a megrendelő elvárásainak megfelel.

Ahogy már fentebb említettük, érdemes angol nyelven írni ezeket, mivel az egyes mondatok mint metódusok jelennek meg. Amennyiben pedig nemzetközi környezetben dolgozunk, el kell fogadnunk, hogy a programozás úgymond hivatalos nyelve az angol, hiszen a programozási nyelvek kulcsszavai is az angol nyelvből erednek. Későbbi példáinkban is ezt fogjuk a Gherkin kapcsán használni.

Fenti példánk még mindig hasonlít az eredeti felhasználói sztorihoz, de már sokkal inkább viselkedéseket fogalmaz meg. Formátuma egységes, tartalma következetes, rendelkezik a megfelelő kiindulási adatokkal és ez az, amit a BDD során el szeretnénk érni. Mégpedig egyrészt olyan megfogalmazást készíteni, ami a Business Analyst vagy a Product Owner számára olvasható és össze tudják vetni az eredeti user story-val, de már sokkal inkább a fejlesztő gondolatmenetét követi, viszont ennek egységes formátuma már feldolgozható egy szoftver által.

A megfogalmazásban használt Scenario, Given, And, When és Then kulcsszavak a Gherkin nyelv leggyakrabban használt parancsai, ami pedig értelmezni tudja mindezeket, az a Cucumber eszköz.

Nézzük át mi mindent csináltunk az előbbiekben!

Először a termék tulajdonos (angolul: product owner) és a megrendelő egyeztetett, ezt a leírást a termék tulajdonos (angolul: product owner) átalakította felhasználói sztorivá, ezt pedig a fejlesztő lefordította Gherkin nyelvre, amit viszont a termék tulajdonos (angolul: product owner) ellenőrzött. Amit még nem láttunk az az utolsó lépés, amikor a Cucumber segítségével a teszter elkészíti az elvárt viselkedés ellenőrzéséhez szükséges kódokat.

Ezt a három fázist a BDD a következőként fogalmazza meg (lásd: <https://cucumber.io/docs/bdd/>):

1. Felfedezés (angolul: Discovery)
2. Kialakítás (angolul: Formulation)
3. Automatizáció (angolul: Automation)

Bár a három lépés nem tartalmazza az ügyféllel történő kezdeti egyeztetést, de ezt a Felfedezés fázisba be lehet vonni és akkor a Három Amigó részvételével megfogalmazható minden User Story. Így az egységesítésben már a megrendelő is részt vehet és a fejlesztői oldal is ki tudja alakítani a megrendelővel közös nyelvezetet.

Összességében látható, hogy bár a BDD a TDD eszközén, a teszten alapul, de nem a tesztelésről szól. Sokkal inkább ezen három lépés segítségével a feladat megfogalmazásán alapuló viselkedés kialakítását és az automatikus elfogadást célozza meg. Mindezek mellett a tesztekkel meghatározunk olyan ismereteket a rendszerről, ami egyben dokumentálja is azt.

Emlékezzünk, hogy a Kiáltvány az agilis szoftverfejlesztésért (angolul: The Agile Manifesto), többek közt előnyben részesíti „a működő szoftvert az átfogó dokumentációval szemben” valamint „a változás iránti készséget a tervek szolgai követésével szemben, lásd: Kiáltvány az agilis szoftverfejlesztésért, <http://agilemanifesto.org/iso/hu/manifesto.html>. A BDD ebben sokat segít, hiszen, ha valamelyik történet változik, akkor csak azt kell módosítani, de rendelkezünk olyan átfogó tesztekkel, amikkel ellenőrizni lehet az elvárt működést.

Az agilis programozás a működő kódra helyezi a hangsúlyt, de a működés validálására azonban szükség van tesztelésre. Ezek megírása viszont nem szokott az első dolga lenni minden fejlesztőnek, hiszen a működő kód írására koncentrálnak. Viszont a BDD folyamat terméke a teszt is, és a működő kód is, ami így megéri az időráfordítást.

És bár a változások követése lényeges, de ez elég kötetlenné tud válni, ha a megrendelő szabadon ötletelhet. Amennyiben tennivalója akad azon túl, hogy egy-egy telefonhívással naponta többször változtat, akkor már sokkal átgondoltabban fog a feladathoz állni. Ebben segít a user story lefektetése.

A BDD fogalmát a Cucumber dokumentációja jól összefoglalja:

„A BDD olyan módszer egy szoftverfejlesztő csapat számára, amivel az üzletemberek és fejlesztők közti különbségeket át lehet hidalni a következők segítségével:

Szerepek közötti együttműködés ösztönzése a megoldandó probléma közös megértésének kialakítása érdekében,

Gyors, kis iterációkban végzett munka a visszacsatolás és az információ áramlás növelése érdekében,

Rendszerdokumentáció készítése, amely automatikusan ellenőrizve van a rendszer viselkedéséhez.

Ezt úgy érjük el, hogy az együttműködés konkrét, valós példákra alapozva készül, melyek a rendszer tényleges viselkedését illusztrálják. Ezeket a példákat használjuk arra, hogy eligazodjunk az ötlettől a megvalósításig, a folyamatos együttműködés során.” Forrás: <https://cucumber.io/docs/bdd/>

### A Gherkin nyelv bemutatása

Ahogy már szó volt róla az előző fejezetben, a Cucumber egy eszköz, ami képes a Gherkin nyelven megírt fejlesztői sztorikat lefordítani forráskóddá. Az első user story a következő volt:

Scenario: sales person filters cars

Given that Zoli is a sales person

And has ROLE\_SALES role to access the database

When Zoli supply red colour

Then the system lists only red cars.

| Scenario: sales person filters cars  Given that Zoli is a sales person  And has ROLE\_SALES role to access the database  When Zoli supply red colour  Then the system lists only red cars. |
| --- |

Milyen kulcsszavakra láttunk ebben példát? Megjelenik a Scenario, Given, And, When és a Then.

A Scenario, mint az angol szó jelentése is mondja egy forgatókönyvet ír le. Ennek címét a Scenario-t követő kettőspont után írhatjuk le. A Scenario szó helyett használhatjuk az Example-t is, ami ugyanazt valósítja meg, de vannak helyzetek, amikor jobban beleillik a mondat környezetébe mint a Scenario.

A Scenario úgynevezett lépésekből (angolul: Step) épül fel, amiket a Given, When, Then, And, But kulcsszavakkal definiálunk soronként új kulcsszóval indulva.

Minden leírás egy Given segítségével indul, ami a kezdőállapotot definiálja. Természetesen, ha összetett a probléma, akkor több ilyen Given állapotot is megadhatunk.

| Example: Lunch requirements  Given that I am hungry  Given that I am thirsty  Given that I am driving after lunch  When I order meal  Then I should order food and beverage  But I shouldn't order alcohol |
| --- |

A When határozza meg azt a lépést, amivel a Scenario végrehajtása elindul mint trigger esemény és a végén a Then fogja a When előtti lépéseket lekezelni.

Viszont több Given sor egymás után nagyon hivatalos hangvételű és nem biztosít logikai kapcsolatot az állapotok között. Ezt az And és But szavakkal tudjuk megtenni, amik a Given helyettesítői. Viszont olvashatóbbá teszik a Scenario tartalmát.

Írjuk át az előbbi példát And és But segítségével!

| Example: Lunch requirements  Given that I am hungry  And I am thirsty  But I am driving after lunch  When I order meal  Then I should order food and beverage  But I shouldn't order alcohol |
| --- |

Nem volt még szó arról hogyan is lehet ezeket a Scenario-kat megadni a Cucumber számára. Ehhez egy úgynevezett Feature fájlra lesz szükségünk, aminek a kiterjesztése .feature .

Nézzünk egy példát!

| Feature: Bridge-keeper test.  In Monty Python and the Holly Grail the bridge-keeper at the  Bridge of Death asks three questions to let someone path the bridge. One question is about the favourite colour.    Scenario: What is your favourite colour?  Given that it is blue  When I supply blue colour as my answer  Then I should be told ‘Right. Off you go.’ |
| --- |

Vagy vegyünk egy technológiai beállítottságú példát:

| Feature: Gross price calculator  In Hungary most product has 27% VAT included in its price.  Adding VAT to net price should result gross price.    Scenario: What is the gross price of a product?  Given that the net price is 100  When I calculate gross price  Then I should get result 127 |
| --- |

A fájl kiterjesztése mellett előírás az is, hogy a Feature kulcsszóval kezdődjön a tartalma, majd a kettőspont után meg kell adnunk a megvalósítandó funkció nevét.

Bár a Feature csak egyszer szerepelhet, de több Scenario-t is tartalmazhat. Illetve, mint a példában is láthatjuk a Feature sora és az első Scenario (vagy Background, vagy Rule) közé több soros leírást is adhatunk a fájlban leírt funkcióról. Ezt a leírást nem használja a Gherkin, de a későbbiekben a szoftver dokumentálásához, vagy az elvárt viselkedés megértéséhez hasznos lehet.

Előfordulhat, hogy olyan Scenario-kat fogalmazunk meg, melyeknek közös lépései vannak, akkor ezen közös részeket kiemelhetjük egy Background definícióba. Minden Scenario előtt a Background-ban tárolt lépések meg lesznek hívva, emiatt ezt a legelső Scenario (vagy Example) előtt kell létrehozni. Abból pedig, hogy minden Scenario számára közös lesz adódik az, hogy csak egyetlen Background lehet egy Feature fájlban.

Például:

| Feature: Testing user roles.  Background:  Given an administrator with username “admin”  And a user with username “user”    Scenario: Setting up a new user as an administrator  Given I am logged in with username “admin”  When I enter to the system  Then I should see “Add new user”    Scenario: Setting up a new user as a user  Given I am logged in with username “user”  When I enter to the system  Then I should not see "Add new user" |
| --- |

Előfordulhatnak ismétlődő Scenario-k, amik csak egy-egy paraméterben térnek el, de más-más eredményt adnak. Természetesen lemásolhatjuk az elsőt, majd módosítgatjuk szükség szerint. De ez nem jó gyakorlat, mivel így ha valamit elírunk az eredetiben, azt az összes variánsban javítani kell, ha valahol ez kimarad, akkor problémát okozhat. Sokkal jobb lenne olyan módszert használni ahol a változókat felsorolhatnánk a rendszer által adott reakciókkal együtt. Erre ad megoldást a Scenario Outline (vagy Scenario Template) amivel táblázatos formában adhatunk meg értékeket.

Például:

Scenario Outline: PIN code entry

Given a pinpad to enter a PIN code

When I enter <pin>

Then I should get <result> as a reply

Examples:

| pin | result |

| 1234 | false |

| 1111 | true |

| 0000 | false |

| Scenario Outline: PIN code entry  Given a pinpad to enter a PIN code  When I enter <pin>  Then I should get <result> as a reply    Examples:  | pin | result |  | 1234 | false |  | 1111 | true |  | 0000 | false | |
| --- |

Sokkal átláthatóbbá válik ezáltal a példa és könnyebb is változtatni rajta, így kisebb a hibalehetőség.

A táblázat fejlécében a változók nevei vannak felsorolva, amikre a szövegben a < és > jelek között hivatkozhatunk, a táblázat oszlopait pedig a | karakter segítségével jelölhetjük ki.

Amennyiben egy Step leírása többsoros definíciót igényel, azt Doc String megadásával tehetjük meg, amit három darab idézőjellel tudunk definiálni.

| Scenario: Entering a multiline recipe  Given that we have a recipe  """  1. Fill up a kettle with water  2. Boil the water  3. Put a teabag into a mug  4. Pour the boiling water into the mug  5. LEave the teabag int the water for 30 seconds  6. Remove the teabag  7. Your tea is ready  """  When I enter it into the text entry field  Then I should see the whole recipe and not just the first line |
| --- |

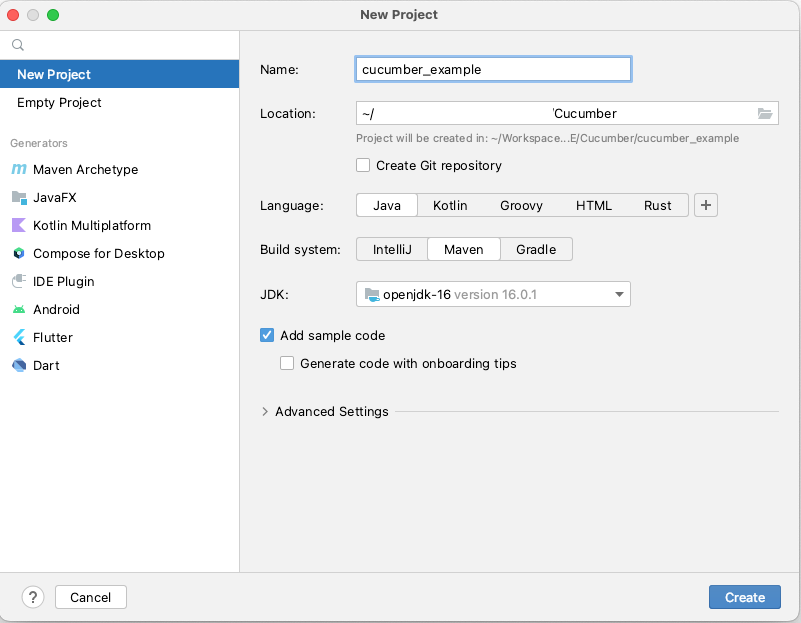
A fentebbi példákon át láthatjuk, hogy a Gherkin nyelv egyszerű, de ugyanakkor következetes szintaktikával rendelkezik. Az általa leírt forgatókönyvek jól olvashatók és a szakemberek mellett laikusok számára is könnyen olvasható.

A Gherkin teljes leírását annak referencia oldalán érhetjük el:  
 https://cucumber.io/docs/gherkin/reference/

### A Cucumber bemutatása

A BDD harmadik fázisa az Automation. Ez az ami során a Gherkin nyelven megfogalmazott viselkedések átalakításra kerülnek végrehajtható tesztekké és ezt a Cucumber nevű eszköz végzi el. Kipróbálásához pedig az IntelliJ pluginját fogjuk használni.

Ahhoz, hogy a BDD-t ki tudjuk próbálni, el kell készítenünk egy Java projektet. Ehhez indítsuk el az IntelliJ-t! (Amennyiben volt már projekt megnyitva, akkor a File menü Close parancsával zárjuk azt be.) A megjelenő Welcome to IntelliJ ablakban kattintsunk a New Project gombra, ennek hatására megjelenik a New Project ablak:



1. Kép

New Project ablak

Adjuk meg a projekt adatait.

A projekt neve legyen cucumber\_example. Telepítve kell lennie egy JDK-nak is, ezt a JDK résznél választhatjuk ki. A build eszköz legyen Maven, és a forráskódunk nyelve pedig Java. Győződjünk meg róla, hogy az Add sample code engedélyezve van, mivel ezzel az alap Java osztályokat is elkészíti a fejlesztői környezet. Ha mindent megadtunk, akkor kattintsunk a Create gombra!

A projekt az elkészülte után betöltődik az IntelliJ-be és a szerkesztő felületen megnyílik a Main osztályunk, ami tartalmazza a main() belépési pontot.

Ahhoz, hogy a Cucumbert használni tudjuk, a pom.xml projekt konfigurációs fájlt is szerkesztenünk kell. A projekt létrehozása után tartalma a következőre hasonlíthat:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<artifactId>cucumber\_example</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<maven.compiler.source>16</maven.compiler.source>

<maven.compiler.target>16</maven.compiler.target>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

</project>

Adjuk a következőket a </properties> és </project> tag-ek közé. Ezek segítségével fogja a Maven letölteni mindazokat a függőségeket, melyekre a projektnek szüksége lesz.

<dependencyManagement>

<dependencies>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-bom</artifactId>

<version>7.17.0</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

<groupId>org.junit</groupId>

<artifactId>junit-bom</artifactId>

<version>5.10.2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit-platform-engine</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-suite</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.13.0</version>

<configuration>

<encoding>UTF-8</encoding>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.2.5</version>

</plugin>

</plugins>

</build>

A szerkesztés után a változtatások érvényesítéséhez kattintsunk a következő képen látható Load Maven Changes gombra.

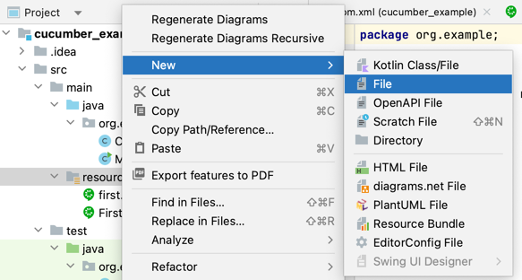


2. Kép

Load Maven Changes

Ezzel tudjuk a pom.xml változásait érvényesíteni.

Következő lépésben készítsük el azt a .feature fájlt, amiben definiálni fogjuk az első BDD Feature-t. Ehhez jobb egérgombbal kattintsunk a src/main/java/resource könyvtáron a projekt ablakban. A megjelenő felugró menüből válasszuk a New majd File menüpontokat.



3. Kép

New File

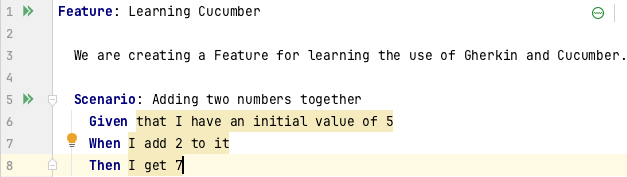
Hozzunk létre egy FirstFeature.feature fájlt.

A fájl neve legyen FirstFeature.feature. Az elkészült fájlunk megjelenik a src/main/java/resources útvonalon és tartalma természetesen üres lesz.

Másoljuk át bele a következőket!

| Feature: Learning Cucumber  We are creating a Feature for learning the use of Gherkin and Cucumber.  Scenario: Adding two numbers together  Given that I have an initial value of 5  When I add 2 to it  Then I get 7 |
| --- |

Oldalt a sorszámok mellett láthatunk két darab kettős nyilat, egyik a Feature: mellett, a másik pedig a Scenario: mellett jelenik meg. A Feature melletti minden Scenario-t lefuttat, ha pedig kiválasztunk egy Scenario-t akkor csak azt.



4. Kép

Scenario-k futtatása

Run ‘Feature: FirstFeature’

Indítsuk el a Feature: mellett találhatót (1. sor), ehhez a kettős nyílra kell kattintani, majd a megjelenő felugró ablakból ki kell választani a Run ‘Future: FirstFeature’ menüpontot.

És ezzel végre is hajtottuk az első Cucumber tesztünket! Ami viszont a következőhöz hasonló hibával fog leállni:

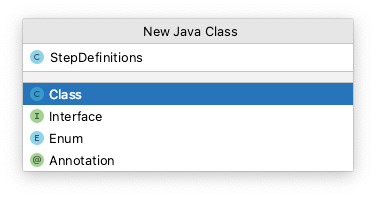
| Step undefined  You can implement this step and 2 other step(s) using the snippet(s) below:    @Given("that I have an initial value of {int}")  public void that\_i\_have\_an\_initial\_value\_of(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  }  @When("I add {int} to it")  public void i\_add\_to\_it(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  }  @Then("I get {int}")  public void i\_get(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  }    Step skipped  Step skipped  Undefined scenarios:  …  1 Scenarios (1 undefined)  3 Steps (2 skipped, 1 undefined)  0m0,561s |
| --- |

Mivel még semmilyen kódot nem hoztunk létre, így a hiba, amit kapunk „1 Scenarios (1 undefined) and 3 Steps (2 skipped, 1 undefined)” azt jelenti, hogy egy Scenario-ból 2 Step át lett ugorva és egy pedig nincs definiálva. De ez az elvárt viselkedés, hiszen nincs még egyetlen tesztünk sem elkészítve. Ugyanakkor segítséget is kapunk a hiba javításához: “You can implement missing steps with the snippets below”.

| @Given("that I have an initial value of {int}")  public void that\_i\_have\_an\_initial\_value\_of(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  }    @When("I add {int} to it")  public void i\_add\_to\_it(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  }    @Then("I get {int}")  public void i\_get(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  } |
| --- |

Hajtsuk is végre ezt! Kattintsunk jobb egérgombbal az src/test könyvtáron a projekt ablakban, majd a felugró ablakban New és a Package menüpontokat válasszuk ki. Célszerű ugyanazt a csomagnevet megadni, amit a Main-nél is használunk (ez alapból az: org.example).

Amint megjelenik a csomag a test könyvtárban, máris létrehozhatjuk a szükséges tesztosztályokat. Jobb egér kattintás a csomagnéven, majd New és Java Class a felugró ablakból. Hívjuk az új osztályunkat a következőnek: StepDefinitions.



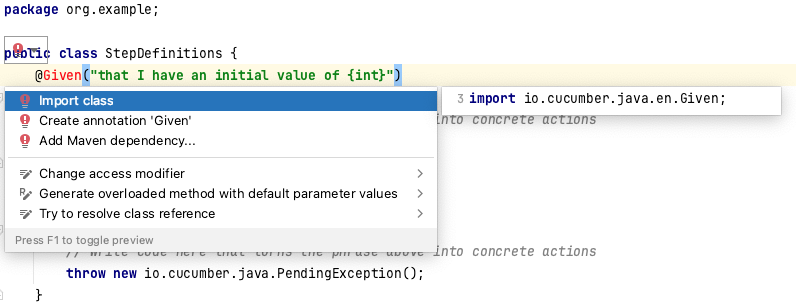
5. Kép

New / Java Class

Válasszuk az osztály létrehozást, majd névnek adjuk meg a StepDefinitions-t.

Amint a StepDefinitions osztály elkészült, másoljuk bele a hibába javasolt hiányzó sorokat!

A Given, When és a Then piros színnel ki lesznek emelve, mivel hiányoznak az őket tartalmazó csomagok importjai. Ehhez fogadjuk el a következő képen látható úgynevezett Quick Fix gyorsjavításokat.



6. Kép

Importáljuk a megfelelő csomagokat

Quick Fix javaslatokat biztosít számunkra a hiba megoldására.

Vagy ha úgy gyorsabb, akkor a következő import sorokat másoljuk az osztály elejére:

| import io.cucumber.java.en.Given;  import io.cucumber.java.en.Then;  import io.cucumber.java.en.When; |
| --- |

Amint a javításokkal elkészülünk ismételten futtathatjuk a teszteket.. Továbbra is elbukik mind a három, hiszen nincs valódi tartalmuk.

Nézzük meg a StepDefinitions osztály tartalmát!

| @Given("that I have an initial value of {int}")  public void that\_i\_have\_an\_initial\_value\_of(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  }    @When("I add {int} to it")  public void i\_add\_to\_it(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  }    @Then("I get {int}")  public void i\_get(Integer int1) {  // Write code here that turns the phrase above into concrete actions  throw new io.cucumber.java.PendingException();  } |
| --- |

A három darab Gherkin lépés három darab metódussá lett átkonvertálva, felhasználva a lépések definícióit a metódus nevek létrehozásához. Mindegyik meg lesz jelölve valamelyik annotációval a következők közül: @Given, @When and @Then. Illetve ahol a leírásban számot érzékelt a Cucumber, ott egy integer egész változót hoz létre.

Folytassuk a tényleges üzleti logika implementálásával!

Készítsünk egy új Java osztályt Calculator névvel és a következő tartalommal (jobb egérkattintás a Man osztályt tartalmazó csomag nevén, majd New / Java Class és az osztály neve legyen Calculator:

| package org.example;  public class Calculator {  public int firstNumber;  public int secondNumber;  public int add(){  return firstNumber+secondNumber;  }  } |
| --- |

Ezt követően lépjünk vissza a StepDefinitions osztályba, és módosítsuk a 3 metódust a következők szerint:

| package org.example;    import io.cucumber.java.en.Given;  import io.cucumber.java.en.Then;  import io.cucumber.java.en.When;    import static org.junit.jupiter.api.Assertions.assertEquals;    public class StepDefinitions {    Calculator calculator = new Calculator();  int actualResult = 0;    @Given("that I have an initial value of {int}")  public void that\_i\_have\_an\_initial\_value\_of(Integer int1) {  calculator.firstNumber = int1;  }    @When("I add {int} to it")  public void i\_add\_to\_it(Integer int1) {  calculator.secondNumber=int1;  actualResult = calculator.add();  }    @Then("I get {int}")  public void i\_get(Integer int1) {  assertEquals(7, actualResult);  } |
| --- |

Figyeljünk oda, hogy a org.junit.jupiter.api.Assertions.assertEquals importálása is megjelenjen a forrás elején. Erre amiatt van szükség, hogy az assertEquals() metódus le tudjon futni a @Then lépésben. (Ez a sor fogja a tényleges teszt futtatást megvalósítani.)

Most már nyugodtan futtathatjuk a tesztet, aminek az eredménye a következőhöz hasonlatos kell, hogy legyen:

1 Scenarios (1 passed)

3 Steps (3 passed)

0m0,505s

| „1 Scenarios (1 passed)  3 Steps (3 passed)  0m0,505s |
| --- |

Ezek szerint 1 Scenario lefutott, ami 3 db lépésből áll. Ezek alapján a Calculator osztály megvalósítja az elvárt működést. Az üzleti logika implementálva lett és tesztelve a megtervezett elfogadási tesztek alapján. Ugyanakkor a Feature fájlunk egy alap dokumentációt is ad a projekthez.

### Összegzés és ellenőrző kérdések

A BDD alapja az automatikus teszt készítés, de nem magáról a tesztelésről szól, hanem betölti a “Három Amigo” közti kommunikációs rést. És bár a BDD használatához szükségünk van szoftver eszközökre, de ettől még nem értékeljük jobban ezen eszközöket az egyénektől és a személyes kommunikációtól.

A Gherkin nyelv képes leírni a User Story-k tartalmát, amit viszont a projekt résztvevői és a Cucumber is fel tud dolgozni.

Összességében pedig a felhasznált technológiákkal agilisek maradunk!

Ellenőrző kérdések:

1. Kérdés: Mi az a BDD?

Válasz: Egy eszköz a fejlesztők, tesztelők és üzletemberek számára, hogy megvalósítsák azt a kollaborációt, amivel létrehozhatják a szükséges követelményeket egy félig technikai nyelven.

1. Kérdés: Hasonlatos a Gherkin a User Story-hoz?

Válasz: Igen, hasonlít hozzá. De amíg a User Story mondatai olyanokkal indulnak, hogy „Mint egy .... , azért hogy ...”, addig a Gherkin a “Given, When, Then” szavakat használja.

1. Kérdés: Szükség van speciális szerkesztőre a .feature fájlok szerkesztésére?

Válasz: Nem, egy hétköznapi szövegszerkesztővel is módosíthatjuk, de a fájl kiterjesztésének .feature-nek kell lennie.

1. Kérdés: Tartalmazhat több Given lépést egy Gherkin Scenario?

Válasz: Igen, de a jobb megértés miatt célszerűbb az And és But alternatíváit használni.

1. Kérdés: Amikor első alkalommal futtatjuk a Cucumber-t az adott Feature definíción, akkor megáll hibával?

Válasz: Igen, mivel még nincs egyetlen teszt sem létrehozva.

## 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 értékadást 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 követhetők a naplóból, hármas szinten minden metódushívásnál készítünk naplóbejegyzést, négyes szinten szinte minden értékadásról 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 ügyfélnél 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. Ettől többet a Pillanatkép (angolul: Memento) tervezési minta leírásából tudhatunk meg.

## Aspektusorientált programozás

Az aspektusorientált programozás alapgondolata az, hogy használjunk tiszta osztályokat, olyan osztályokat, amiket nem szennyez be a naplózás, a jogosultság kezelés, és más ilyen, mindenhol előjövő feladatok. Az egyszerűség kedvéért vegyük a naplózást (általában minden jegyzetben ez a példa).

Naplózni kell a hallgató osztályban és a kurzus osztályban is. Ez azt mutatja, hogy a naplózás 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 osztályra gondolunk, akkor eszünkbe juthat a Java nyelv POJO fogalma. Ez a Plain Old Java Object rövidítése. A Java programozók ezt olyan értelemben használják, hogy olyan osztály, amiben nincs JEE (Java Enterprise Edition) specifikus kódrészlet, nincs benne naplózás és más egyéb „fertőzés”. A POJO megfelelője C#-ban a POCO, Plain Old C# Object.

Az aspektusorientált programozás fogalmai:

* Aspektus (angolul: aspect): Ide kell kiszervezni az átmetsző követelményeket.
* Szövés (angolul: weaving): Az a folyamat, ami során az aspektusokat beleszőjük a programba.
* Kapcsolódási pontok (angolul: join point): Ezeken a pontokon lehet beleszőni a programba az aspektusokat. Általában metódusok előtt és után érdemes aspektusokat beleszőni a programba, de akár a metódus helyére is kerülhet az aspektus, azaz lecserélhető a metódus kódja.
* Pontszűrő (angolul: pointcut): Itt rendeljük össze az aspektusokat és a kapcsolódási pontokat, azaz megmondjuk, hogy melyik aspektust melyik kapcsolódási pontnál kell beleszőni 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 Solution Explorer-ben a projektre jobb klikk után előjövő helyi menüben tehetjük meg.

Figyeljük meg, hogy a fenti példában a Hasznos Osztály egy POCO, amit attribútumokkal láttunk el. Az attribútumok az osztály, vagy a metódus neve elé írt szögletes zárójelek közötti 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] attribútummal jelölünk. Az osztály, amit létrehozunk, végső soron egy attribútum lesz, így az osztály nevének XYZAttribute alakúnak kell lennie és az XYZ-t attribútumként használhatjuk. A mi esetünkben az XYZ helyén Trace áll. Az attribútumnak lehet paramétere is, ami abból látszik, hogy a konstruktornak van-e paramétere. A mi esetünkben egy paramétere van.

Az aspektuson belül hozzuk létre a kapcsolódási pontokat (angolul: 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 (angolul: pointcut) ez esetben a Trace attribútum. Az a metódus, ami elé odaírjuk ezt a attribútumot, azelőtt le fog futni az OnEntry, utána pedig az OnExit. Ez esetben naplózás történik. Gondoljunk bele, mennyivel egyszerűbb így a naplózás kezelése!

A szövés automatikusan történik, amikor lefordítjuk a projektet. A PostSharp rendszer a köztes kódba fordítja bele az aspektusainkat, így az ilyen kód nyomonkövetése nehéz.

Végezetül, vegyük észre, hogy itt is a Separation of Concerns elvet használtuk. A naplózás felelősségét és a naplózással bepiszkított metódusokat szétvágtuk naplózó aspektusra és POCO osztályokra, amiket később majd szövéssel rak össze a PostSharp.

## 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. Ezt elérni dinamikus teszteléssel nagyon költséges, mert 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.

### 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 közötti átmenetek.

Ezeket a hibákat könnyebb felülvizsgálattal megtalálni, mint más technikákkal:

1. szabványoktól / kódolási szabályoktól való eltérések,
2. követelményekkel kapcsolatos hibák, pl. nincs minden funkcionális követelményhez funkció,
3. tervezési hibák, pl. az adatbázis nincs harmadik normálformában,
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 szakértő bevonásával, rövid idejű),
4. inspekció (külsős szakértő bevonásával, hosszú idejű).

### 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 egységteszt. A szépítés során nem szabad megváltoztatni a kód funkcionalitását, de a szerkezet, pl. egy metódus törzse, szabadon változtatható. A szépítés után minden egységtesztet 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 tapasztaltabb fejlesztő, ritkán a menedzsment,
3. hatékonysága függ az átnéző személyétől, minél tapasztaltabb, annál több hibát vehet észre,
4. célja a korai, költséghatékony hiba felderítés.

### Felülvizsgálat – Átvizsgálás

Az átvizsgálás már egy kicsivel 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 megjegyzéseikkel 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 megvalósítani. 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, megértése, azokban hibakeresés.

### Felülvizsgálat – Technikai felülvizsgálat

Technikai felülvizsgálatra általában akkor kerül sor, ha a szoftver teljesítményével nem vagyunk elégedettek. Azt általában könnyű megtalálni a felhasználói visszajelzések és úgynevezett 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 leak) 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.

### Felülvizsgálat – Inspekció

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 inspekció esetén leírjuk pontosan, hogy milyen gyors legyen.

Az inspekció szó abból jön, hogy a probléma megoldásához általában nem elég csak a szoftver egy részét átvizsgálni, hanem az egész forráskódot adatbázissal együtt inspekció alá kell vonni. Inspekciót alkalmazunk akkor is, ha egy régi (esetleg már nem támogatott programozási nyelven íródott) kódot akarunk szépíteni / átírni, hogy ismét rugalmasan lehessen bővíteni.

Az inspektornak nagy tekintélyű szakembernek kell lennie, mert az általa javasolt változtatások általában nagyon fájdalmasak, 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 inspekció esetén, mint technikai felülvizsgálat esetén.

Az inspekció 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 úgynevezett PoC-ot (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 inspekció 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.

### 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.akármi;” 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.akármi;” 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.

### 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 be kell illeszteni a fordítás folyamatába. Ezután a statikus elemző felhívja a figyelmünket a tipikus programozói hibákra. Ezek általában programozási nyelv-specifikusak, de léteznek nyelvfüggetlenek, pl. a Sonar vagy a Yasca rendszer, amelyek egy-egy plugin segítségével adaptálhatóak a kedvenc nyelvünkhöz.

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: <https://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-ot.

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); }

Ebben az esetben a „There is an apparent infinite recursive loop” figyelmeztetést kapjuk nagyon helyesen, mivel egy rekurzív függvényt írtunk bázisfeltétel nélkül, és így semmi sem 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 tételbizonyítót.

# Ábrajegyzék

* [1. ábra: A szoftverfejlesztés életciklusa](#bookmark=id.pxg7fj4b9k8t)
* [2. ábra: Példa használati eset diagramra](#bookmark=id.jqz2kncuum5q)
* [3. ábra: Rendszerhatár első példája (információs rendszer kezelése)](#bookmark=id.f0dztec4rza)
* [4. ábra: Rendszerhatár második példája (egységek feladatainak körülhatárolása)](#bookmark=id.58q2xom4u52z)
* [5. ábra: Folyamatábra egy főnév szereplői körének eldöntésére](#bookmark=id.6ynbcqwumump)
* [6. ábra: Folyamatábra egy cselekedet meghatározására (használati eset-e](#bookmark=id.tjvb6nqtci4n))
* [7-9. ábrák: Példák sima, illetve üzleti használati esetekre](#bookmark=id.w4kif7y00neg)
* [10. ábra: Használati eset példa bal- és jobboldali szereplőkre](#bookmark=id.ixzt57ae072p)
* [11. ábra: Nyílfajták demonstrációja a használati eset diagramokban](#bookmark=id.hjnucztjbp16)
* [12-17. ábra: A 11. ábra kisebb diagramokra](#bookmark=id.22bo2uwvsyuz) való bontása