Szoftvertechnológia

© Szabolcsi Judit
2012

Tartalom

A rendszernek csak a külső viselkedését írják le, és kerülni kell benne a rendszer tervezésének jellemzőit. A felhasználói követelményeket természetes nyelven írják, így a következő problémák

(Ajánlott irodalom: Ian Somerville: Szoftverrendszerek fejlesztése. Második, bővített, átdolgozott kiadás, Panem Kiadó, Budapest 2007.*)*

ÁTTEKINTÉS

I. Szoftver és szoftvertervezés

I.1. Mi a szoftvertervezés?

Napjainkban a legtöbb ország számítógép alapú, összetett rendszerektől függ. Egyre több termék foglal magában számítógépeket és áll valamilyen formában szoftveres irányítás alatt. A szoftverek ezekben a rendszerekben a teljes költségnek nagy és egyre nagyobb hányadát teszik ki. (Érdekesség: 1969-ben az US Apollo programnak kevesebb, mint 10MB szoftverkódra volt szüksége ahhoz, hogy embert juttasson a Holdra, az amerikai űrállomásprogramhoz (Colombus) már kb. 100 MB-ra van szükség és az ember Marsra juttatásához még többre lesz.) Emiatt a szoftverek költséghatékony módon történő előállítása fontos a nemzeti és nemzetközi gazdaság működéséhez.

El kell fogadnunk, hogy **a szoftver termékké vált**, és mint minden termék esetében, **az előállításához technológiára van szükség. A technológia valamely termék gyártási eljárásainak összessége.** A szoftver egyfajta termék, tehát:

- **van szolgáltatási funkciója**
- **van minősége**
- **van előállítási költsége**
- **van előállítási határideje**

Ezek a tervezési paraméterek. A szoftvertechnológiának biztosítania kell a tervezési paramétereknek megfelelő termék előállítását. Természetesen a program előállításának problémái nem a kis – egy ember által, fejben is megtervezhető – programok esetén jelentkeznek, hanem a nagyméretű programrendszereknél. Ezek jellemzői:

- nagy bonyolultságú rendszer (azaz fejben tartva nem kezelhető az összes osztály, objektum, stb.)
- több ember fejleszti (team - csoport)
- hosszú élettartamú (így számos változatát kell előállítani, követni, karbantartani, dokumentálni)

A nagyméretű programok előállítására alkalmas szoftvertechnológiákkal a szoftvertervezés (Software Engineering) nevű mérnöki tudományág foglalkozik. A szoftvertervezés a szoftvertermék minden lehetséges vonatkozását érinti, vagyis nemcsak a technikai folyamatokat, hanem a projektmenedzselést, a szoftverfejlesztést támogató eszközöket, valamint az elméleteket és módszereket is. A tervezés nem más, mint a körülmények figyelembevételével a legmegfelelőbb és leghatékonyabb módszer kiválasztása.

A szoftvertervezés célja a szoftverrendszerek költséghatékony fejlesztése.

Mi a rendszer? **A rendszer egymással kölcsönösen kapcsolatban lévő komponensek jól átgondolt, egy adott cél elérése érdekében együtt dolgozó együttese.**

Minket csak a szoftvereket is tartalmazó rendszerek érdekelnek. Ezek két kategóriába sorolhatók:

- **Technikai számítógép-alapú rendszerek**
  Hardver- és szoftverkomponensekből állnak, de eljárásokból és folyamatokból nem. Pl. mobiltelefon és a legtöbb PC-s szoftver. Ezeknek az a sajátosságuk, hogy az őket használó

egyén vagy szervezet céljainak az elérését lehetővé tevő tudáshalmaz NEM része ennek a rendszernek. Pl. egy szövegszerkesztő nem tartalmazza egy regény megírásához szükséges „információkat”.

- **Szociotechnikai rendszerek**
  Ez a bővebb kategória, tartalmaz egy vagy több technikai rendszert, de ezen túl egy tudáshalmazt is arról, hogy ezekkel az eszközökkel hogyan érhető el a cél. Ezek a rendszerek a szoftver- és hardverkomponensek mellett egy jól definiált folyamattal rendelkeznek, és az azt működtető emberek is részei. Pl. egy regényt megjelentető kiadói rendszer. Ezek a rendszerek általában nemdeterminisztikusak, vagyis ugyanarra a bemenetre nem mindig ugyanazt a kimenetet fogják szolgáltatni, hiszen az emberek nem mindig ugyanúgy reagálnak ugyanarra az ingerre.

Mi a továbbiakban az olyan szociotechnikai rendszerekkel foglalkozunk, amelyek szoftver- és hardverelemeket tartalmaznak és szoftver által megvalósított interfésszel rendelkeznek a felhasználók felé. A szoftvertervezőknek számos ismerettel kell rendelkezniük a szociotechnikai rendszerek tervezésével kapcsolatban.

A szoftvertervezés fiatal tudománynak számít, hiszen ez a fogalom először 1968-ban, egy később „szoftverkrízisnek” nevezett probléma megoldása céljából tartott konferencián hangzott el először. Ez a szoftverkrízis az (akkor még) erős harmadik generációs hardverek bevezetésének köszönhető. Azok teljesítménye ugyanis az addig megvalósíthatatlannak tűnő alkalmazásokat megvalósítható feladatokká tette, így a szoftverek nagyságrendekkel nagyobbak és bonyolultabbak lettek elődeiknél.

I.2. Mi a szoftver?

A szoftver a számítógépes programok, a hozzájuk kapcsolódó dokumentációk és konfigurációs adatok összessége. A szoftvertermékeknek két fő csoportja van: általános termékek és rendelésre készített (egyedi igényeknek megfelelő) termékek.

Az általános termékeket nevezik dobozos szoftvereknek is. Ezeket egy fejlesztő szervezet készíti és adja el a piacon bármely vevőnek. Itt a vevők közvetlenül nem befolyásolhatják a termék jellemzőit, a szoftverspecifikációt a gyártó cég tartja kézben. Ilyenek a játékok, az operációs rendszerek, az adatbázis-kezelők, a szövegszerkesztők, a különböző rajz- és tervezőprogramok, fordítóprogramok és a projektmenedzselési eszközök.

A rendelésre készített termékek esetében a megrendelő igényei szerint kell a terméket kifejleszteni. Itt a megrendelő adja meg a specifikációt (vagy legalábbis annak a vázlatát) és az elkészült szoftverterméket ez alapján ellenőrzi. Ilyenek lehetnek: könyvelőprogramok, egyéni üzleti folyamatokat támogató rendszerek, forgalomirányító (pl. légi, vasúti), elektromos eszközök vezérlőrendszerei vagy ellenőrző rendszerek.

A kétfajta termékcsoport közötti választóvonal egyre inkább elmosódik, mivel egyre több szoftvercég fejleszt általános termékeket, amiket aztán a vásárlók igénye szerint testre szab. A vállalatirányítási rendszerek (ERP – Enterprise Resource Plannig), mint pl. az SAP jó példa erre. Ezeket tekinthetjük egy **harmadik csoportnak** is, amely részben az általános termékek, részben a rendelésre készítettek tulajdonságaival rendelkezik.

II. A szoftverfolyamat

A szoftverfolyamat tevékenységek és kapcsolódó eredmények olyan sora, amely egy szoftvertermék előállításához vezet. Ezek történhetnek a szoftverfejlesztés kezdeteitől (nulláról indulunk), de napjainkban sokkal gyakoribb az, hogy egy már meglévő szoftvert kell módosítani vagy kiegészíteni. A szoftverfolyamatok – mint minden szellemi folyamat – összetettek és emberi nézetektől és döntésektől függenek. Emiatt – mivel emberi kreativitást igényel – a szoftverfolyamat automatizálását jobbára csak a teljesen mechanikus részekre sikerült kiterjeszteni. A CASE (Computer-Aided Software Engineering – számítógéppel segített szoftver mérnökség) eszközök szolgálnak a folyamat automatizálására (ezekről majd később bővebben).

Bár számos különböző szoftverfolyamat létezik, van **négy** olyan **alapvető tevékenység**, amely mindegyikben megtalálható:

1. **Szoftverspecifikáció:** a szoftver funkcionális és nem funkcionális tulajdonságait írja le. Funkcionális tulajdonság az, hogy konkrétan mit csinál, milyen funkciói vannak. (Nem funkcionális tulajdonság lehet pl.: a szoftver sebességére, memória-felhasználására, egyszerre kiszolgált felhasználók létszámára vonatkozó megkötések, a dokumentáltsága, vagy az elvárt megbízhatóság és védettségi szint.)
2. **Szoftvertervezés és implementáció:** a specifikációnak megfelelő szoftvert elő kell állítani.
3. **Szoftvervalidáció:** meg kell mutatni, hogy azt fejlesztettük-e ki, amit az ügyfél kívánt.
4. **Szoftverevolúció:** a szoftvert úgy kell alakítani, hogy a megrendelő által kért változtatásoknak minél könnyebben eleget tudjunk tenni.

II.1. Szoftverspecifikáció

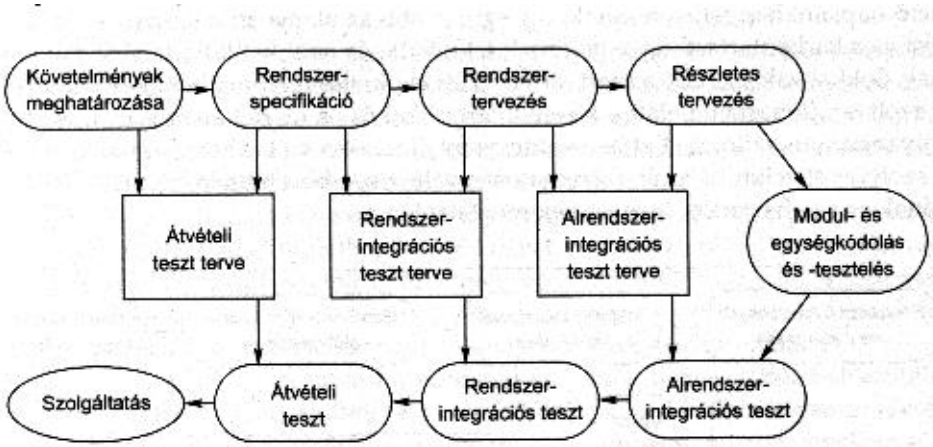
Itt kell meghatároznunk, hogy milyen szolgáltatásokat követelünk meg a rendszertől, és hogy a rendszer fejlesztésének és működtetéseinek milyen megszorításait alkalmazzuk. Ezt a tevékenységet gyakran **követelménytervezésnek is hívják**. Ez a rész a szoftverfolyamat különösen kritikus szakasza, mivel az itt vétett hibák később elkerülhetetlenül problémákat okoznak (a tervezésben és az implementációban). **A szoftverspecifikáció a követelménydokumentum előállítását eredményezi**. A követelmények általában két szinten kerülnek kifejtésre: a végfelhasználóknak és ügyfeleknek valamint a fejlesztőknek.
A megvalósíthatósági tanulmányban meg kell becsülni, hogy a felhasználók kívánságai kielégíthetőek-e az adott szoftver- és hardvertechnológia mellett. Mennyibe fog kerülni a rendszer? Ennek az elemzésnek relatíve olcsónak (esetleg ingyenesnek) és gyorsnak kell lennie. A tanulmány eredménye, hogy érdemes-e folytatni a munkát.
A követelmények feltárása és elemzése már meglévő hasonló rendszerek megfigyelésén, valamint a leendő felhasználókkal való megbeszéléseken alapul. Itt egy vagy több rendszermodellt, sőt prototípust is készíthetünk.
A követelményspecifikáció az elemzési tevékenységekből összegyűjtött információk egységes dokumentummá való alakítására szolgáló művelet.
A követelményvalidáció tevékenysége ellenőrzi, hogy mennyire valószerűek, konzisztensek és teljesek a követelmények. Az itt feltárt hibákat a dokumentumok módosításával ki kell javítani.

II.2. Szoftvertervezés és implementáció

Az implementáció nem más, mint a rendszerspecifikáció futtatható rendszerré történő konvertálása. Ez magában foglalja a szoftver tervezését és a programozást. A tervezési folyamat általános modellje:
Architekturális tervezés: a rendszert felépítő alrendszereket és a köztük lévő kapcsolatokat azonosítja és dokumentálja.
Absztrakt specifikáció: az alrendszerek szolgáltatásainak absztrakt specifikációjának megadása és azok a megszorítások, amelyek mellett a szolgáltatások működnek.
Interfész tervezése: az alrendszerek interfészeinek megtervezése és dokumentálása (a specifikáció egyértelmű legyen!)
Komponens tervezése: A szolgáltatásokat el kell helyezi a különböző komponensekben és meg kell tervezni az interfészeket.
Adatszerkezet tervezése: Meg kell határozni és részletesen meg kell tervezni a rendszer implementációjában használat adatszerkezeteket.
Algoritmus tervezése: Meg kell tervezni és pontosan meg kell határozni a szolgáltatásokhoz szükséges algoritmusokat.

II.3. Szoftvervalidáció

Célja: hogy megmutassa, a rendszer egyezik saját specifikációjával, és hogy a rendszer megfelel a rendszert megvásárló ügyfél elvárásainak. A validációnál két fő technikát használnak: a



4.10. ábra. A szoftverfolyamat tesztelési fázisai

szoftverátvizsgálásokat és a tesztelést.
Mivel a validáció költséges folyamat, a tervezését már a fejlesztési folyamat elején el kell kezdeni. A tesztelési és fejlesztési tevékenységek összekapcsolása a teszttterveken keresztül (V-modell):

II.4. Szoftverevolúció

A nagy és összetett rendszerek hosszú élettartamúak. Ezalatt változnak: egyrészt korrigálni kell az eredeti rendszer követelményeinek hibáit, másrészt a felmerülő új követelményeket is bele kell építeni. A rendszer számítógépei ezalatt valószínűleg kicserélődnek, a rendszert használó szervezetek is megváltoznak. A szoftverevolúció fontos, miután a cégek jelentős része teljesen a

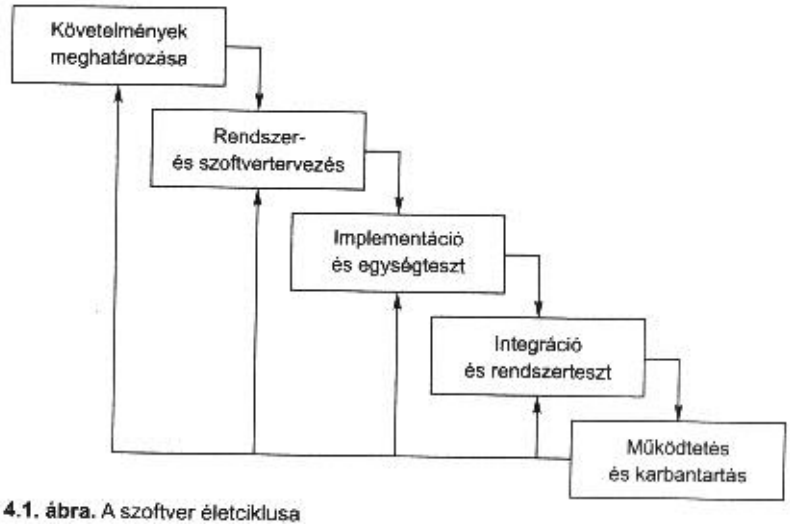
szoftverrendszerétől függ. A nagy cégeknél szoftverköltségvetés akár 90%-át is kiteheti az evolúciós költség.

II.5. A szoftverfolyamat modelljei

A szoftverfolyamat modellje a szoftverfolyamat absztrakt reprezentációja. (A szoftverfolyamat egy bizonyos perspektívából adódó egyszerűsített leírása.) Minden modell más és más szempontból mutat be egy folyamatot. A modellek nem pontos, részletes leírásai a folyamatnak, csak nagyvonalú áttekintést adnak róla.

II.5.1. Vízesés modell (Szoftver életciklus modell)

Ez volt az első publikált modell. Széles körben használatos a gyakorlatban. A következő fázis addig nem indulhat el, amíg az előző be nem fejeződött. Ez a modell akkor működik jól, ha a követelmények teljesen ismertek. Hátrányai: a projekt munkájának megszervezés nehézkes; új szolgáltatások utólagos bevezetése drága. (A visszafelé mutató nyilak azt jelzik, hogy ha valamilyen probléma lépett fel az egyik fázisban és ott nem sikerült megoldani, akkor a közvetlenül előtte lévő fázisba kell visszamenni és ott kell megoldani.)



4.1. ábra. A szoftver életciklusa

II.5.2. Evolúciós modell

Az alapötlet az, hogy ki kell fejleszteni egy kezdeti implementációt (prototípust), azt a felhasználókkal véleményeztetni, majd sok-sok verzión át addig finomítani, amíg megfelelő nem lesz.

Ezt is használják a gyakorlatban. Ez a modell a felhasználó kívánságait jobban kielégítő programot eredményez. A rövid élettartamú kis (<100.000 programsor) és közepes (<=500.000 programsor) rendszerek fejlesztéséhez ideális. Hátrányai: a folyamat nem látható; a rendszerek gyakran szegényesen strukturáltak; a gyors fejlesztés rendszerint a dokumentáltság rovására megy.

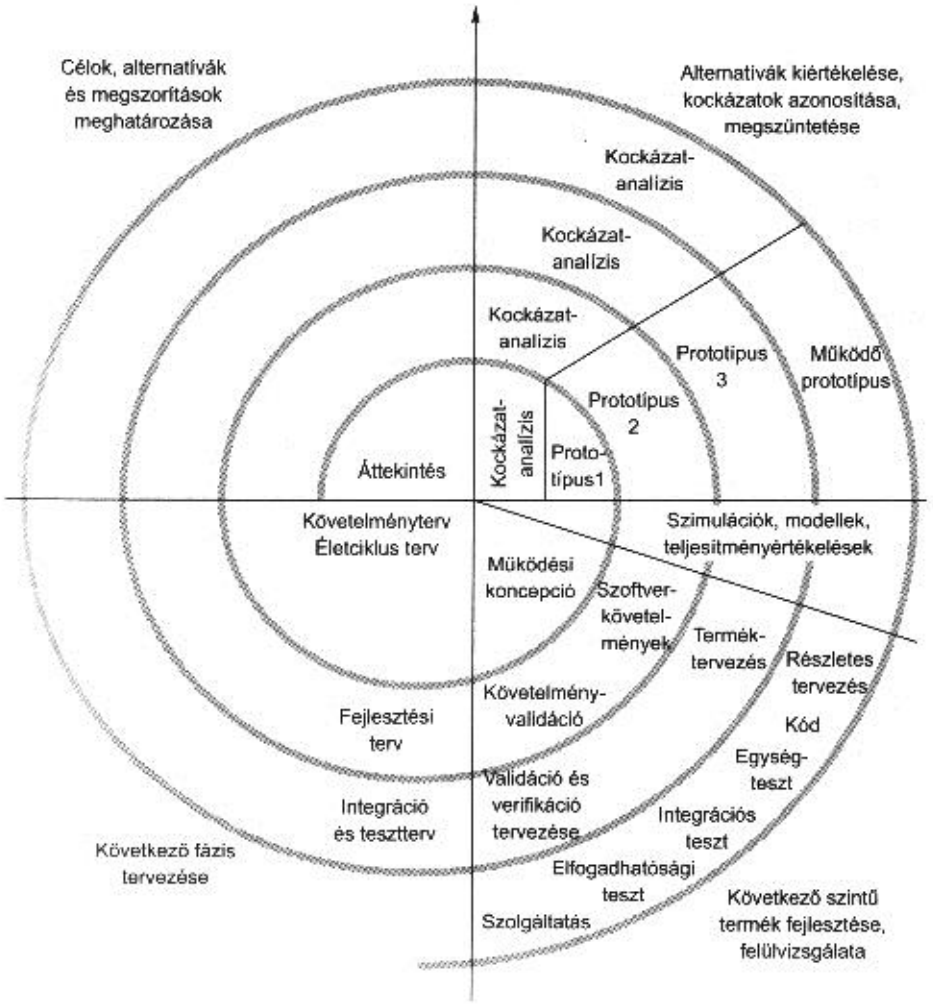
II.5.3. Formális rendszerfejlesztési modell

Kapcsolódik a vízesés modellhez, de a fejlesztési folyamat alapja a rendszerspecifikáció futtatható programmá történő transzformálása formális matematikai eszközökkel.

Legismertebb példája az IBM által kifejlesztett Cleanroom-módszer, ahol minden egyes fejlesztési szakasz után annak hibátlanságát formális módszerekkel bebizonyítják (így nincs tesztelés). A

formális rendszerfejlesztést nem használják széles körben, (hátrányai) mert speciális szakértőket kíván és nem lesz sokkal jobb és olcsóbb a termék, mint más módszereknél. Érdemes viszont használni a szigorú biztonságosságot, megbízhatóságot, és védelmet igénylő termékeknél.

II.5.4. Boehm-féle spirális modell



4.5. **ábra.** A szoftverfolyamat Boehm-féle spirális modellje (© 1988 IEEE)

A szoftverfolyamatot nem tevékenységek és a köztük található esetleges visszalépések sorozataként tekinti, hanem spirálként. Minden egyes körben a spirál a szoftverfolyamat egy-egy fázisát reprezentálja. A legbelső kör a megvalósíthatósággal foglalkozik, a következő a rendszer követelményeinek meghatározásával, aztán a rendszertervezéssel, stb. A spirál minden egyes ciklusát négy szektorra osztjuk fel: célok, alternatívák meghatározása; kockázat becslése és csökkentése; a fázis termékének megvalósítása és validálása; következő fázis tervezése.
A spirális modell a kockázati tényezőkkel explicite számol. A spirális modellben nincsenek rögzített fázisok, és felölelhet más folyamatmodelleket is (vízesés, evolúciós, formális transzformáció). Hátrányai: a modell alkalmazása bonyolult, munkaigényes feladat; a párhuzamos foglalkoztatás csak a 3. szektorban lehetséges.

II.5.5 Újrafelhasználás-orientált fejlesztés (komponens alapú modell)
Ez a módszer nagymértékben az elérhető újrafelhasználható szoftverkomponensekre támaszkodik. A komponensek lehetnek teljes rendszerek, pl. egy szövegszerkesztő, vagy kisebb egységek (osztályok, modulok, stb.) Előnye: lecsökkenti a kifejlesztendő részek számát, így csökkenti a költségeket és a kockázatot. Ez általában a kész rendszer gyorsabb leszállításhoz vezet. Hátrányai: a követelményeknél hozott kompromisszumok elkerülhetetlenek, és ez olyan rendszerhez vezethet, ami nem felel meg a felhasználó valódi kívánságának.
A fejlesztés szakaszai:
Komponenselemzés: Adott a követelményspecifikáció, ami alapján megkeressük, hogy milyen kész komponensek valósítják meg. A legtöbb esetben nincs egzakt illeszkedés, és a kiválasztott komponens a funkcióknak csak egy részét nyújtja.
Követelménymódosítás: A követelmények elemzése a megtalált komponensek alapján. A követelményeket módosítani kell az elérhető komponenseknek megfelelően. Ahol ez lehetetlen, ott újra a komponenselemzési tevékenységet kell elővenni, és más megoldást keresni.
Rendszertervezés újrafelhasználással: A rendszer szerkezetét kell megtervezni, vagy egy már meglévő vázat felhasználni. A tervezőknek figyelembe kell venniük, hogy milyen újrafelhasznált komponensek lesznek, és úgy kell megtervezni a szerkezetet, hogy ezek működhessenek. Ha nincs megfelelő újrafelhasználható komponens, akkor új szoftverrészek is kifejleszthetők.
Fejlesztés és integráció: A nem megvásárolható komponenseket ki kell fejleszteni és a COTS (Commercial-Off-The-Shelf – kereskedelemben kapható)-rendszerekkel össze kell kapcsolni. A rendszerintegráció itt sokkal inkább tekinthető a fejlesztési folyamat részének, mint különálló tevékenységnek.

II.5.6. RUP
Az **RUP (Rational Unified Process)** jó példája a modern folyamatmodelleknek, amelyek az UML-re épülnek. Ez egy igen összetett folyamatmodell, ami **3 perspektívából és 4 fázisból áll**. A három perspektíva:

- dinamikus perspektíva, amely a modell fázisait mutatja
- statikus perspektíva, amely a végrehajtandó folyamattevékenységeket mutatja
- gyakorlati perspektíva, amely gyakorlatokat javasol a folyamat alatt

A négy fázis:

1. Indítás. Ennek a célja a rendszer üzleti vonatkozásának tisztázása. El kell dönteni, hogy hoz-e nyereséget a rendszer.
2. Előkészítés. Ennek a célja a probléma megértése. A fázis befejezése után rendelkezni fogunk a szoftver fejlesztési tervével.
3. Létrehozás. Rendszerterv, programozás, tesztelés.
4. Átadás. A rendszer fejlesztői környezetből felhasználói környezetbe való átültetése. A fázis eredménye egy dokumentált szoftverrendszer, mely megfelelően működik a megrendelő céges környezetében.

Az RUP legfontosabb újdonságai a fázisok és a munkafolyamatok szétválasztása, és az, hogy a felhasználói környezetben való üzembe helyezés a folyamat része. A fázisok dinamikusak és konkrét célokkal rendelkeznek, a munkafolyamatok statikusak és olyan technikai tevékenységeket tartalmaznak, amik a teljes fejlesztésen keresztülhúzódnak és nem rendelhetők hozzá az egyes fázisokhoz.

III. Automatizált folyamattámogatás

III.1. A CASE-eszközök szerepe

A számítógéppel támogatott szoftvertervezéshez (Computer-Aided Software Engineering - CASE) használt szoftvereket nevezzük CASE-eszközöknek.

A szoftverfolyamatban a következő tevékenységeket támogatják a CASE eszközök:

Követelményspecifikáció során: grafikus rendszermodellek, üzleti és domain (a modellezni kívánt terület) modellek megtervezése.

Elemzés/tervezés során: adatszótár kezelése, mely a tervben található egyedekről és kapcsolataikról tartalmaz információt; felhasználói interfész generálását egy grafikus interfészleírásból, melyet a felhasználóval együtt készíthetünk el.; a terv ellentmondásmentesség-vizsgálata

Implementáció során: automatikus kódgenerálás (Computer Aided Programming - CAP); verziókezelés

Szoftvervalidáció során: automatikus teszt-eset generálás, teszt-kiértékelés, -dokumentálás

Szoftverevolúció során: forráskód visszafejtés (reverse engineering); régebbi verziójú programnyelvek automatikus újrafordítása újabb verzióba.

Mindegyik fázisban alkalmazható: automatikus dokumentumgenerálás; projektmenedzsment támogatás (ütemezés, határidők figyelése, erőforrás-tervezés, költség- és kapacitásszámítás, stb.)

A CASE-eszközök korai pártolói azt jósolták, hogy a szoftverek minőségében és a termelékenységben nagyságrendi javulást okoznak ezek az eszközök, de valójában csak 40% körüli a javulás.

Az eredményességet két tényező korlátozza:

- A szoftvertervezés lényegében tervezői tevékenység, amely kreatív gondolkodást igényel. A létező CASE-eszközök automatizálják a rutintevékenységeket és hasznosítják a mesterséges intelligencia bizonyos technológiáit, de ez utóbbival még nem értek el átütő eredményt.
- A legtöbb szervezetben a szoftvertervezés csoportos tevékenység, és a benne résztvevők rengeteg időt töltenek a csapat más tagjaival való eszmecserével. A CASE-technológia ehhez nem nyújt túl nagy segítséget.

III.2. A CASE-eszközök fajtái

Három szempontból csoportosítjuk a CASE-eszközöket:

- funkcionális nézőpontból
- folyamat nézőpontból
- integrációs nézőpontból

Funkcionalitásuk alapján: tervezői, szerkesztő, változtatáskezelő, konfigurációkezelő, prototípuskészítő, módszertámogató, nyelvi feldolgozó, programelemző, tesztelő, nyomkövető, dokumentációs, újratervezési eszközök

Az eszközök által támogatott folyamat alapján:

	Specifikáció	Tervezés	Implementáció	Verifikáció és validáció
Tervezőeszközök	*	*	*	*
Szerkesztőeszközök	*	*	*	*
Változtatáskezelő eszközök	*	*	*	*
Konfigurációkezelő eszközök		*	*	*

Prototípus-készítő eszközök	*			*
Módszertámogató eszközök	*	*		
Nyelvi feldolgozó eszközök		*	*	
Programelemző eszközök			*	*
Tesztelőeszközök			*	*
Nyomkövető eszközök			*	*
Dokumentációs eszközök	*	*	*	*
Újratervezési eszközök			*	

Integrációs nézőpontból:

1. Tools (eszközök): az egyedi folyamatlépéseket támogatják.
2. Toolkits (eszközkészletek): néhány fejlesztési fázist támogatnak, eszközök többé-kevésbé integrált halmaza
3. I-CASE Integrated Workbench (környezetek): a szoftverfolyamat mindegyik részét támogatják, általában integrált eszközkészleteket tartalmaznak.

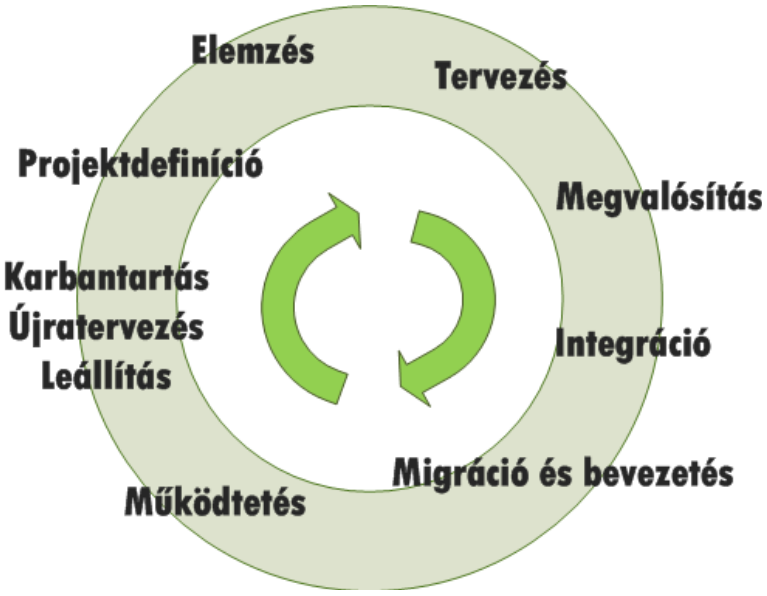
Néhány konkrét CASE-program:

Enterprise Architect, BridgePoint Suite, Cool termékcsalád, iUML, Objectory, Paradigm Plus, PTECH, SystemArchitect, ObjectMaker, GDPro, StP Software Through Pictures, With Class 2000, ARIS, Forte ADE, Together és a Rational programcsalád.

IV. Projektmenedzsment

IV.1. A szoftver életciklus

Ahogy más műszak termékeknek, a szoftvereknek is van életciklusuk: kigondolják, megtervezik, előállítják, üzembe helyezik, használják, karbantartják, végül leállítják, üzemen kívül helyezik őket. Ezt a ciklust mutatja be vázlatosan a lenti ábra. A rajz „kezdőpontja” a projektdefiníció lehetne, de mivel a régi rendszer leállítása után elég valószínű, hogy kell egy új, ezért valóban körkörös a folyamat, igazából nincs sem eleje sem vége.



(Az ábra forrása: Tarczali Tünde: UML diagramok a gyakorlatban – tankonyvtar.hu)

IV.2. Vezetői tevékenységek

Projekt: egy időben behatárolt erőfeszítés, egy egyedi termék, szolgáltatás vagy eredmény létrehozása céljából. (PMBOK GUIDE magyarul: Projektmenedzsment útmutató, Akadémia Kiadó 2009.)
Az adott cégnél mindig vannak folyamatosan elvégzendő feladatok (könyvelési, pénzügyi, stb.), amelyek a cég fennmaradását szolgálják. Emellett a cég vezetése indíthat egy vagy több projektet is. A projektek egy-egy szoftvertermék vagy szolgáltatás létrehozása céljából indulnak. A projekteket természetesen menedzselni kell.

A menedzselés szükségessége igen fontos megkülönböztető tulajdonság a professzionális szoftverfejlesztés és az amatőr programozás között. A szoftverprojekt menedzserének dolga, hogy biztosítsa a szoftverprojekt megfelelését a költségvetési és ütemezési megszorításoknak, valamint azt, hogy olyan terméket szállítsanak le, amely képes hozzájárulni az üzleti célok eléréséhez. A jó menedzselés nem tudja garantálni a projekt sikerességét, a rossz azonban legtöbb esetben a projekt bukását okozza. (Tehát szükséges, de nem elégséges feltétel.)

A szoftvermenedzserek más mérnöki tervezési munkák menedzseléséhez hasonló feladatot végeznek, azonban van néhány különbség:

- *A szoftver nem kézzelfogható valami.* Egy építőmérnöki projekt vezetője látja a terméket a fejlesztés alatt. Ha a munka ütemezése csúszik, az látszik a terméken is. A szoftver azonban megfoghatatlan, nem látható vagy tapintható, így a menedzserek nem látják annak haladását, csak a mások által előállított dokumentációra támaszkodhatnak.
- *Nincsenek régi, jól bevált, szabványos szoftverfolyamatok.* A többi mérnöki tudományág már hosszú évek óta kipróbált, tesztelt módszerekkel dolgozik (pl. hídépítési módszerek), a szoftverfolyamatok azonban még eléggé újak, így nem tudjuk nagy bizonyossággal előre megmondani, hogy mikor fog egy szoftverfolyamat fejlesztési problémákba botlani.
- *A nagy szoftverprojektek gyakran „különálló" projektek.* A nagy szoftverprojektek gyakran különböznek minden más korábbi projekttől, így sokkal nehezebb felkészülni a problémákra. Ráadásul a gyors technológiai váltások a korábbi tapasztalatokat gyorsan elavulttá teszik.

A szoftverprojekt vezetőjének feladatai:

- indítványokat írni: erre általában a szerződés megkötéséhez van szükség
- a projektet megtervezni és ütemezni: a tevékenységek azonosítása, a mérföldkövek és a leszállítható részeredmények meghatározása; költségbecslés
- az embereket kiválogatni: itt gyakran előfordul az ideálistól való eltérés, mert vagy a költségvetés nem teszi lehetővé a jól képzett szakemberek alkalmazását, vagy nem áll rendelkezésre kellő számú megfelelő szakember (sem a szervezeten belül, sem kívül), vagy maga a fejlesztő szervezet vezetősége kívánja úgy, hogy tapasztalatlan munkaerőt alkalmazzunk, hogy betanulhassanak.
- a projekt költségét figyelemmel kísérni
- a projektet figyelni és felülvizsgálni: ez egy folyamatos tevékenység; össze kell hasonlítani a valódi és tervezett haladást és költségeket. A nem formális figyelés, vagyis a tagokkal való napi beszélgetés sokszor tisztább képet ad a problémákról
- beszámolókat írni és előadni: mind a megrendelő, mind a saját szervezet felé

IV.3. Projekt tervezése

A projektterv részei:

1. *Bevezetés.* Ez a projekt céljainak tömör leírása, a megszorításokkal együtt.
2. *Projekt szervezet.* A projektcsapat összeállításának módja, a részt vevő emberek neve és azok szerepe.
3. *Kockázatelemzés*. A lehetséges kockázatok leírása, azok bekövetkezésének valószínűsége, és a csökkentésükre ajánlott stratégiák.
4. *Hardver- és szoftvererőforrás követelmények*. Milyen hardver és szoftver kell a fejlesztéshez. Ha a hardvert meg kell vásárolni, akkor árat is meg kell adni.
5. *Munka felosztása.* Tevékenységekre való felosztás, mérföldkövek és részeredmények meghatározása.
6. *Projekt ütemterve.* A tevékenységek közötti függőségek meghatározása, határidők becslése, a résztvevők feladatokhoz rendelése.
7. *Figyelési és jelentéskészítési mechanizmusok.* A menedzser által előállított jelentések fajtái, határideje, a projektfigyelési technikák.

Ez a projektterv csak a fejlesztési folyamattal foglalkozik, de a menedzsernek **más típusú terveket** is el kell készítenie. Ezek a következők:

Terv	**Leírás**
Minőségi terv	Meghatározza azokat a minőségi eljárásokat és szabványokat, melyeket a projektben használni kell.
Validációs terv	Meghatározza a megközelítési módot, az erőforrásokat és ütemterveket, melyeket a rendszer validációjához használni kell.
Konfigurációkezelési terv	A konfigurációkezelési eljárások meghatározása.
Karbantartási terv	Előre megadja a rendszer karbantartási követelményeit, a karbantartási költségeket és a szükséges erőfeszítéseket.
Munkaerő-fejlesztési terv	Hogyan kell a projektcsapat tagjainak szaktudását és tapasztalatait fejleszteni.

A projekttervezési folyamat:

Megállapítani a projekt megszorításait
A projekt paramétereinek egy kezdeti összegzését elkészíteni
Definiálni a projekt részeredményeit és mérföldköveit
amíg a projekt nincs kész, vagy nem vonták vissza, addig
    felvázolni a projekt ütemtervét
    elindítani az ütemtervnek megfelelő tevékenységeket
    várakozni (kb. 2-3 hét)
    átvizsgálni a projekt előrehaladását
    felülvizsgálni a projekt paramétereinek becslését
    frissíteni a projekt ütemtervét
    újra tárgyalni a projekt megszorításait és részeredményeit
    ha probléma merül fel elindítani a technikai felülvizsgálatokat és a lehetséges átdolgozásokat
ciklus vége

A projekttervezés iteratív folyamat, csak akkor válik teljessé, ha maga a projekt is teljessé

vált. A bölcs projektvezető nem feltételezi, hogy minden jól fog menni. Bizonyos fajta problémák csaknem mindig felmerülnek a projekt ideje alatt, ezért a kezdeti feltevéseknek és ütemterveknek inkább pesszimistának kell lenniük.

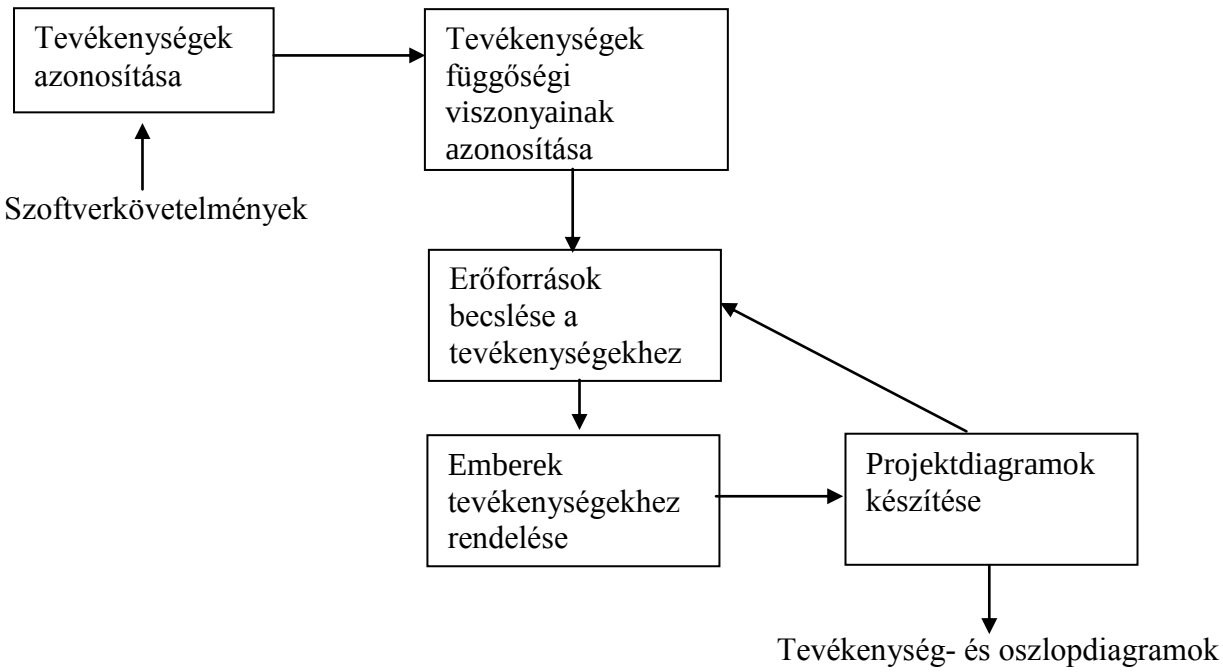
Mérföldkövek és részeredmények:

A menedzsereknek információra van szükségük. A szoftver nem kézzelfogható dolog, így az információk csak dokumentum formájában biztosíthatók, melyek a fejlesztés alatt álló szoftver állapotát írják le.

A mérföldkő a szoftverfolyamat tevékenységeinek egy ellenőrző pontja, egy logikai szakasz vége. Egy vagy több olyan részfeladat után helyezzük el, ahol a részfeladatok eredményes befejezése nélkül nem lehet továbbhaladni. Minden mérföldkőnél érdemes formális kimenetet készíteni. Az olyan határozatlan mérföldköveket, mint pl. a „kód 80%-a kész" lehetetlen ellenőrizni, így értelmetlenek és használhatatlanok. A mérföldkövek meghatározásához a szoftverfolyamatot alapvető tevékenységekre és hozzájuk kapcsolódó kimenetekre kell tördelni.

A **részeredmények** a projekt olyan eredményei, amelyek átadhatók a megrendelőnek. Ezek általában mérföldkövek is, de a mérföldkő nem szükségszerűen részeredmény.

IV.4. Projekt ütemezése



Az ábra a projekt ütemezési folyamatát mutatja, a szükséges tevékenységek egymásutánját.

A projekt ütemtervét legtöbbször diagramok halmazaként (oszlopdiagramok és tevékenységhálók) adják meg, ennek automatizálásra szolgál pl. a Microsoft Solutions Framework vagy a Microsoft Project.

A projekttevékenységek normális esetben legalább egy hétig tartanak, ennél kisebb fázisokra nem érdemes bontani. Szintén hasznos, ha a tevékenységek nem tartanak tovább 8-10 hétnél. Ha egy tevékenység ennél hosszabb, akkor ajánlatos részekre bontani.

Az ütemterveknél érdemes figyelembe venni, hogy a projekten dolgozó emberek megbetegedhetnek, felmondhatnak, a hardverek meghibásodhatnak, az elengedhetetlen hardvereket vagy szoftvereket csak később kapjuk meg, stb.

Ian Sommerville szerint a becslésre érdemes még 30%-ot rászámolni a nem várt problémák miatt,

és további 20%-ot az olyan dolgok miatt, amikre a tervezéskor nem gondoltunk.

IV.5. Kockázatkezelés

A projektmenedzser egyik nagyon fontos feladata a projekt ütemezése közben fellépő, vagy a fejlesztett szoftver minőségében változást előidéző kockázatok becslése és az azok elkerülése érdekében tett lépések elvégzése. **A kockázatok azonosítását és az azok hatásának minimalizálása érdekében történő tervek felvázolását együtt kockázatkezelésnek nevezzük.** Három fő kockázati kategória van:
projekt-, termék- és üzleti kockázat.
A **projektkockázat** kihatással lehet az ütemtervre vagy az erőforrásokra. A **termékkockázat** a fejlesztett szoftver minőségére vagy teljesítményére gyakorol hatást. Az **üzleti kockázat** a szervezetre van hatással. Ez nem egy egymást kizáró osztályozás, hiszen pl. ha egy tapasztalt programozó elhagyja a projektet az tekinthető projektkockázatnak, hiszen a rendszer leszállítása késhet, tekinthető termékkockázatnak, mert az őt helyettesítő kevésbé tapasztalt programozó nem tud olyan minőségű terméket előállítani, de tekinthető üzleti kockázatnak is, hiszen a szervezet így kevesebb tapasztalattal rendelkezik.
Néhány gyakran előforduló kockázati tényező és a kockázat típusa:

Kockázat	**Típusa**
munkaerő távozása	projekt
vezetőség megváltozása	projekt
hardver elérhetetlensége	projekt
követelmények megváltozása	projekt és termék
specifikáció késése	projekt és termék
méret alábecslése	projekt és termék
CASE-eszköz alulteljesítése	termék
technológia megváltozása	üzleti
versenyképes termék kerül piacra, mielőtt a rendszer elkészülne	üzleti

A kockázatkezelés folyamatát négy részre lehet bontani:
1. kockázat azonosítása; 2. kockázat elemzése; 3. kockázat tervezése; 4. kockázat figyelése

A kockázat azonosítása fázisban csapatmunkaként vagy egyszerűen a menedzser tapasztalatait hasznosítva egy listát állítunk össze a lehetséges kockázatokról. Egy másik, gyakorlatibb tipizálást és néhány konkrét példát mutat a következő táblázat:

Kockázattípus	Lehetséges kockázatok
technológiai	A rendszerhez használt adatbázis nem tud mp-ként annyi tranzakciót feldolgozni, mint amit elvárunk tőle. A használt programozási nyelv nem képes hatékonyan támogatni a feladathoz szükséges módszereket.
emberi	Lehetetlen a megfelelő szakértelemmel rendelkező munkatársakat összetoborozni. A kulcsfontosságú munkaerő megbetegszik.
szervezeti	A szervezetben lévő pénzügyi problémák a projekt költségvetésének csökkentését okozzák. A projekt vezetősége megváltozik.
eszköz	A CASE eszköz által generált kód nem hatékony. A különböző típusú CASE-eszközöket nem lehet integrálni.

követelmény	A követelmények szükséges megváltoztatása a terv nagyobb léptékű átdolgozását kívánja meg. A megrendelők nem képesek megérteni, hogy az általuk kívánt szolgáltatások miért lennének olyan drágák.
becslési	A szoftver kifejlesztéséhez szükséges időt alábecsülték. A hibák számát alulbecsülték.

A kockázat elemzése fázisban minden egyes azonosított kockázatot át kell tekinteni és meg kell ítélni annak valószínűségét és komolyságát. Ezek általában nem konkrét értékek, csak sávok: a **valószínűség**: nagyon kicsi (<10%), kicsi (10-25%), mérsékelt (25-50%), magas (50-75%) vagy nagyon magas (>75%); **a kockázat hatása**: nem jelentős, elviselhető, súlyos vagy katasztrofális. Az elemzési folyamat eredményeit súlyosságuk szerint táblázatba rendezzük és ebből a táblázatból kiválasztjuk azt a néhány tételt, amit a projekt során végig figyelemmel kell kísérni.

A kockázat tervezése fázisban az előző fázisban kiválasztott néhány kulcskockázat kezelési stratégiáját határozzuk meg. Ezek a stratégiák három nagy csoportba sorolhatók: elkerülési stratégiák; minimalizációs stratégiák; vészhelyzeti tervek.

A kockázat figyelése fázisban az azonosított kockázatok bekövetkezési valószínűségének és hatásának a változását figyeljük.

A projekt megvalósítása során felmerülő kockázatok				
Megnevezése	**Bekövetkezési valószínűség[1]**	**Hatása a projektre[2]**	**Kockázat mértéke[3]**	**Kezelésének módja**
Projekt indítási szakaszának kockázatai				
Projektszervezet felállítása elhúzódik	1	3	3	A közbeszerzési dokumentáció megfelelő elkészítése, a közbeszerzési eljárás hibamentes lebonyolítása, a nyertes szervezettel való szerződéskötés lehető leghamarabbi (törvény által megszabott időkeretet figyelembe véve) megvalósítása
A meglévő dokumentumok feldolgozása a szükségesnél hosszabb ideig tart, emiatt a projektmenedzsment szervezet nem a kellő információk birtokában tudja megkezdeni a munkáját	1	2	2	A projektmenedzsment szervezet tagjai számára a szerződéskötést követően rögtön biztosítani kell a szükséges dokumentumokat, a projektmenedzsment szervezet tagjainak pedig kellő időt kell szánniuk az átadott dokumentumok tanulmányozására.
Projekt tervezési szakaszának kockázata				
A PAD tartalmilag hiányos, nem teljes körű, így a projekt működését szabályozó dokumentum nem tudja betölteni a szerepét	1	3	3	A PAD elkészítése során körültekintően kell megtervezni a projekt működését szabályozó eljárásokat, a dokumentumnak minden, a projekt működésével kapcsolatos területre kell rendelkeznie egyértelmű eljárási renddel.
Projekt végrehajtási, monitoring és kontrolling szakaszának kockázatai				
Közbeszerzési eljárás sikertelensége	2	3	6	Közbeszerzési feltételek körültekintő kidolgozása, határidők pontos betartása, közbeszerzési szakértő alkalmazása
Az Európai Uniós támogatások kifizetése akadozóvá válik, likviditási problémák lépnek fel.	1	3	3	A Kedvezményezett nincs ráhatással a kockázati tényezőre. Lobbi tevékenységet gyakorolhat, de eredménye nem garantált. Az esetlegesen felmerülő likviditási problémák kezelését banki források igénybe vételével lehet megvalósítani.

[1] Bekövetkezési valószínűség értéke: 1 – alacsony, 2 – közepes és 3 - magas valószínűség
[2] Hatása a projektre értéke: 1 – kicsi, 2 – közepes és 3 – nagy
[3] Kockázat mértéke= *Bekövetkezési valószínűség* és *Hatása a projektre* mező értékének szorzata. Ha értéke 1-3 között van: alacsony kockázat, ha 4 és 6 között van: közepes a kockázat, ha az értéke 9: kritikus a kockázat.

Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: R. A. Maksimchuk – E. J. Naiburg: UML földi halandóknak. Kiskapu Kiadó, Budapest 2006. és Harald Störrle: UML 2. Panem Kiadó, Budapest 2007.)

V. UML

V.1. Mi is az az UML?

A szoftvertervezés (szoftvermérnökség – software engineering) olyan szoftvertechnológiákkal foglalkozik, amelyek olyan programok előállítására alkalmasak, amiket:
- nem egy ember, hanem többen együtt, csoportban (team) fejlesztenek
- nem készülnek el rövid idő alatt, hanem hónapokig-évekig fejlesztik őket
- nem tarthatók fejben, hanem összetettek, bonyolultak, ezért részekre kell bontani és modellezni kell őket.

Az UML mind a három fentebb felsorolt szempontból segíthet.

UML (Unified Modeling Language) – egységes modellező nyelv, ez egy szabványos grafikus jelölőrendszer. **Nem programozási nyelv**, hanem a fejlesztők, a tesztelők, a menedzserek és a megrendelők közötti kommunikációt segíti. Gondoljanak az elektromos eszközök kapcsolási rajzaira, amiket mindenki, aki megtanulta a jelölésrendszert, ugyanúgy értelmez. Vagy párhuzamot vonhatunk a zenei kottával is, ami szintén egy szabványos (zenei) jelölésrendszer, aminek megszületése előtt a zenedarabokat kizárólag hallás után tudták megtanulni a zenészek.
Az UML az szeretne lenni a programtervezésben, ami a zenében a kotta, vagy a villamosmérnökök között a kapcsolási rajz.

Az UML-nek van még egy nagy előnye: **szabványos**. Az OMG nevű csoport (Object Management Group – www.omg.org Technology menüpont) gondoskodik róla, hogy elkészüljenek az újabb és újabb verziók, és ezeket szavazással elfogadják, majd a weboldalon közzétegyék. Az OMG tagjai pl.: az Adobe, Fujitsu, HP, Hitachi, Microsoft, NASA, SAP, egyetemek, cégek és még sokan mások.
Jelenleg az **UML 2.4.1-es** verzió az aktuális – az OMG oldalán mindig meg lehet nézni (és le is lehet tölteni) a legutoljára elfogadott verziót.

Tehát mire jó az UML:
- szemléltetésre
- a fejlesztői csoporton belül, illetve a fejlesztők és a megrendelők közötti kommunikációra
- specifikálásra
- megvalósításra
- dokumentálásra

V.2. Az UML részei

A szoftverfejlesztési folyamat sikeréhez három dolog szükséges:
- Jelölésrendszer (notation) – ez lehet az UML
- Folyamat (process) – pl. RUP (Rational Unified Process)
- Eszköz (tool) – pl. Enterprise Architect, Rational Rose, Visual Studio 2010 osztálydiagram készítő része

Mi ezek közül csak a jelölésrendszerrel foglalkozunk.

Az UML diagramokból áll. **A modell és a diagram nem ugyanazt jelenti.** Az UML földi halandóknak c. könyv alapján (14. old.): „Noha első látásra hasonlónak tűnhetnek, a diagramok és a modellek különböznek egymástól. A modell elvont ábrázolás, amely a modellezett dolog céljának meghatározásához szükséges összes elemet (üzleti, kapcsolati, rendszer- és egyéb tényezőket) tartalmazza. A diagram ezzel szemben konkrét rálátást ad valamire, amit meghatározott környezetben akarunk megérteni. A diagram csupán a modell egészének vagy részének egy adott nézőpontja. Egy bizonyos modellezési elem csak egyszer szerepelhet a modellben, de ugyanaz az elem több diagramban is megjelenthet. […]
A modellek (tehát) több diagramból állnak, a diagramok pedig az elemek és azok más elemekkel való kölcsönhatásának ábrázolásai."

A diagramokon és a modelleken kívül az UML rendelkezik még ún. metamodellel is. **A meta-modell a modell modellje.** Az UML metamodellje az UML modellek nyelvét és szerkezetét írja le. Az UML modellek számos különböző elemből tevődnek össze, és a metamodell határozza meg ezeknek az elemeknek a tulajdonságait, azt, hogy milyen módon kapcsolódhatnak és hogy mit jelent egy ilyen kapcsolat.
A legtöbb műszaki nyelv, pl. az SQL is rendelkezik metamodellel.

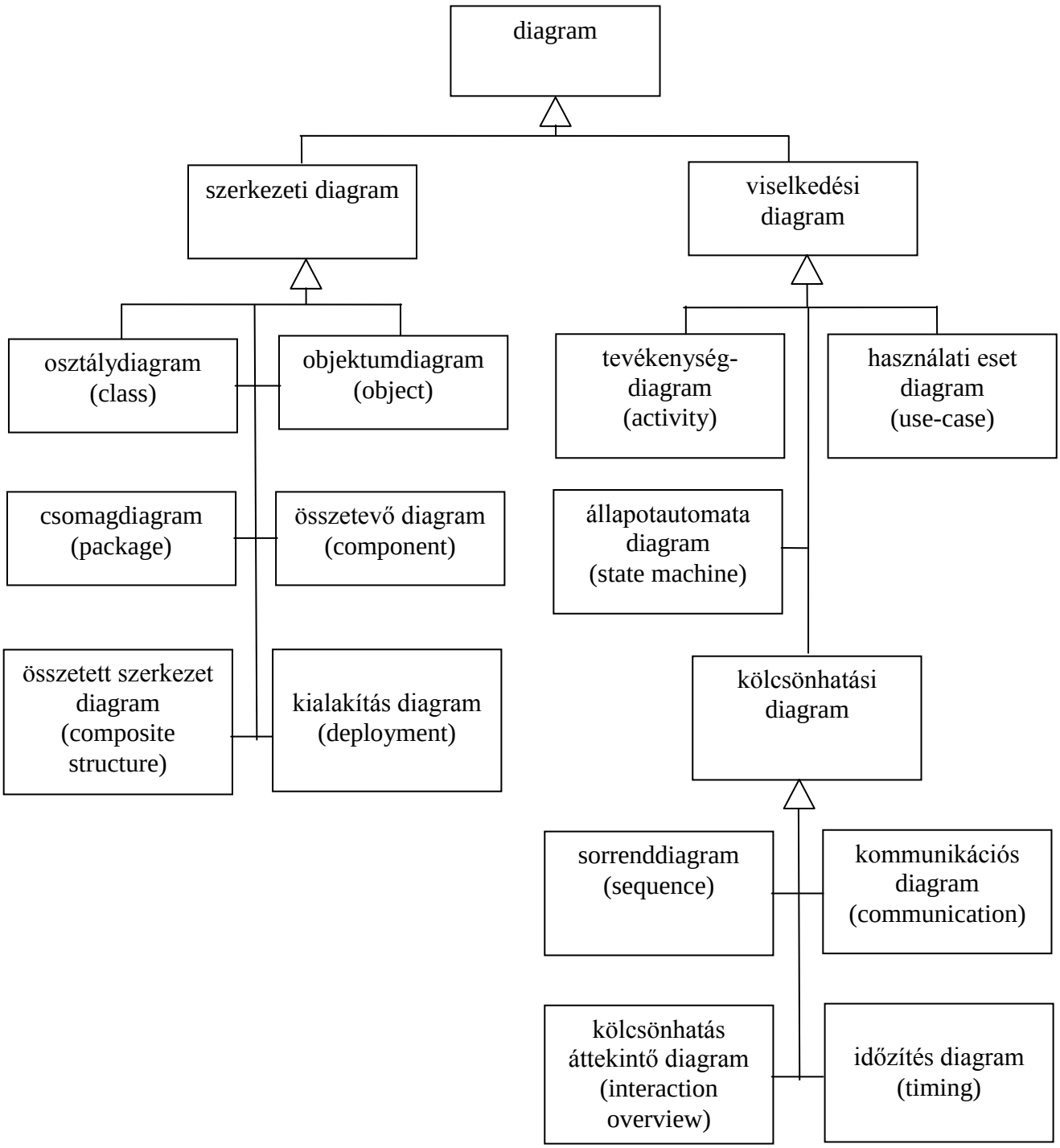
Nézzük tehát az UML 2.x (2.0, 2.1, 2.2, 2.3) diagram-fajtáit. Alapvetően két nagy csoportba oszthatjuk őket**: a szerkezeti (statikus) és a viselkedési (dinamikus) diagramok**. A szerkezeti diagramok nem törődnek az időbeli változással, ők a modellezett rendszer állapotát egy adott időpillanatban mutatják be. Ezzel szemben a viselkedési diagramok folyamatában, változásában mutatják ugyanazt a modellezett rendszert.

Szerkezeti diagramok:
- **osztálydiagram (class)**
- **objektumdiagram (object)**
- **csomagdiagram (package)**
- **összetevő diagram (component)**
- **összetett szerkezet diagram (composite stucture)**
- **kialakításdiagram (deployment)**

Viselkedési diagramok:
- **tevékenység diagram (activity)**
- **használati eset vagy feladat diagram (use-case)**
- **állapotautomata vagy állapotgép diagram(state machnie)**

Kölcsönhatási diagramok:
- **sorrend diagram (sequence)**
- **kommunikációs diagram (communication)**
- **időzítés diagram (timing)**
- **kölcsönhatás áttekintő diagram (interaction overview)**

A következő oldalon látható ábra maga is egy UML osztálydiagram, amely az általánosítás relációval kapcsolja össze pl. a diagram és a szerkezeti diagram osztályokat.



Röviden ismerkedjünk meg a fenti diagramok szerepével!

Osztálydiagram: az UML modellezésben leggyakrabban használt diagramfajta. A rendszerben található állandó elemeket, azok szerkezetét és egymás közötti logikai kapcsolatát jeleníti meg. Általában a rendszer logikai és fizikai felépítésének ábrázolására szolgál. Az UML-ben található osztály és a programozási nyelvek osztályfogalma különböző! Az UML-ben az osztály fogalomnak

legalább négy jelentése van:

osztály mint fogalom	Ez az elemzési/ tervezési fázisban gyakori, ahol a szakterület fogalmait nevezzük osztálynak.
osztály mint típus	Ez már programozási nyelv közelibb; az objektumok az osztály típus értékei, példányai.
osztály mint objektumhalmaz	Az osztály itt csak egy csoportosítás, az azonos felépítésű objektumok halmaza.
osztály mint implementáció	Az OOP nyelvekben az osztály egyszerűen csak egy implementáció (kód) is lehet, amin az objektumai osztoznak.

Ez a négy jelentés különböző erősséggel van jelen, aszerint, hogy az osztályt a szoftver életciklus melyik fázisában használjuk:

osztályok felhasználási módja	elemzési fázisban	tervezési fázisban	megvalósítási fázisban
fogalom	**igen**	esetleg	nem
típus	esetleg	**igen**	**igen**
objektumhalmaz	nem	**igen**	**igen**
implementáció (kód)	nem	esetleg	**igen**

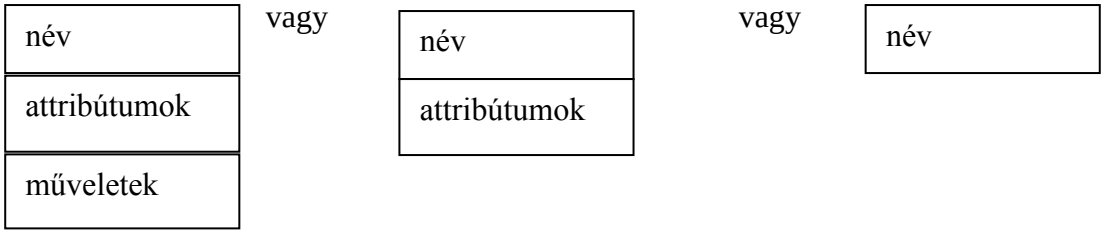
Objektumdiagram: a rendszer egy adott időpontban érvényes pillanatképét határozza meg. Az osztálydiagramból származtatjuk.
Csomagdiagram: a csomagok olyan modellelemek, amelyek más modellelemek csoportosítására szolgálnak, és ezeket valamint a köztük lévő kapcsolatokat ábrázolja ez a fajta diagram.
Összetevő diagram: az összetevő vagy komponens a rendszer fizikailag létező és lecserélhető része, feltéve, hogy az új komponens csatlakozási felülete (interfésze) megegyezik a régivel. (Mint a LEGO-kockák.) Ez a diagram főleg implementációs kérdések eldöntését segíti. A megvalósításnak és a rendszeren belüli elemek együttműködésének megfelelően mutatja be a rendszert.
Összetett szerkezeti diagram: A modellelemek belső szerkezetét mutatja.
Kialakítás diagram: A fizikai (kész) rendszer futásidejű felépítését mutatja. Tartalmazza a hardver és a szoftverelemeket is.

Tevékenységdiagram: A rendszeren belüli tevékenységek folyamatát jeleníti meg. Általában üzleti folyamatok leírására használjuk.
Használati eset/feladat diagram: A rendszer viselkedését írja le, úgy, ahogy az egy külső szemlélő szemszögéből látszik.
Állapotautomata diagram: Az objektumok állapotát és az állapotok közötti átmeneteket mutatja, valamint azt, hogy az átmenetek milyen esemény hatására következnek be.
Kommunikációs diagram: Az objektumok hogyan működnek együtt a feladat megoldása során, hogyan hatnak egymásra.
Sorrenddiagram: Az objektumok közötti üzenetváltás időbeli sorrendjét mutatja.
Időzítés diagram: A kölcsönhatásban álló elemek részletes időinformációit és állapotváltozásait vagy állapotinformációit írja le.
Kölcsönhatás áttekintő diagram: Magas szintű diagram, amely a kölcsönhatás-sorozatok közötti

vezérlési folyamatról ad áttekintést.

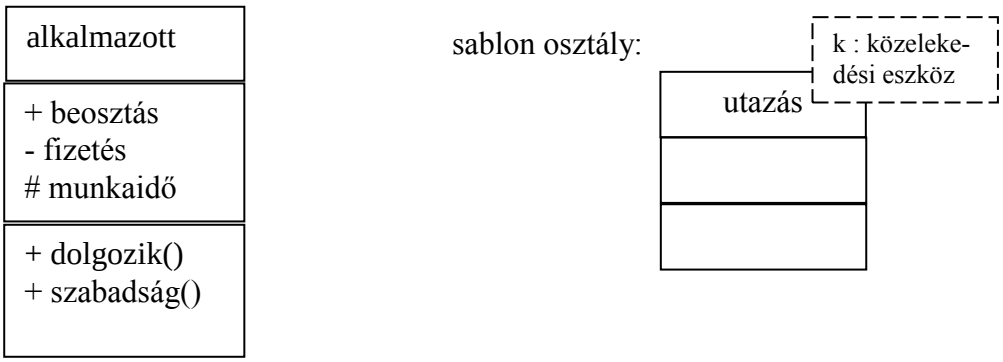
V.3. Osztálydiagram

Egyszeresen összefüggő gráf, amelynek csomópontjai osztályokat, élei pedig relációkat fejeznek ki. Az osztály jele egy általában három részre osztott téglalap, ahol a felső sávba az osztály nevét, a középsőbe az osztály attribútumait, az alsóba pedig az osztály műveleteit írjuk.



Néha csak az osztály nevére vagy a nevére és az attribútumaira vagyunk csak kíváncsiak, ekkor nem rajzoljuk ki a téglalap mind a három részét, csak egyet vagy kettőt. Ezek a fentebb látható egyszerűsített jelölések, de az osztály szabványos jele a három részre osztott téglalap!

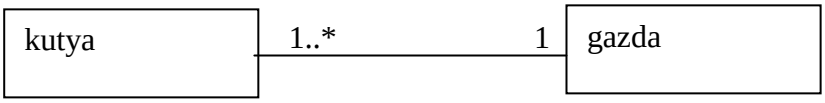
Az attribútumok és műveletek láthatóságai: + public, # protected, - private, ~ package. Pl.:



A statikus adattagokat vagy műveleteket aláhúzással jelöljük, az absztrakt osztály neve pedig dőlt betűs.

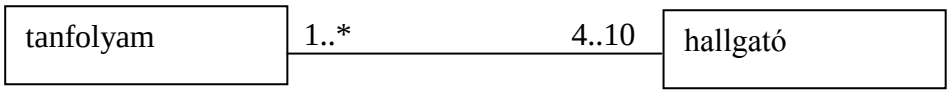
Az osztályok közötti kapcsolatok:
- asszociáció/társítás (association)
- aggregáció/rész-egész kapcsolat (aggregation)
- általánosítás (generalization)
- függőség (dependency)
- megvalósítás (realization)

Asszociáció: Valamilyen használati kapcsolat a két osztály között, amelyek egymástól függetlenek, de legalább az egyik ismeri/használja a másikat.

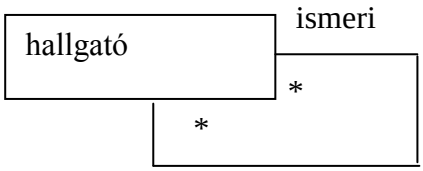


(Egy kutyának pontosan egy gazdája van, és minden gazdának legalább egy, legfeljebb akárhány

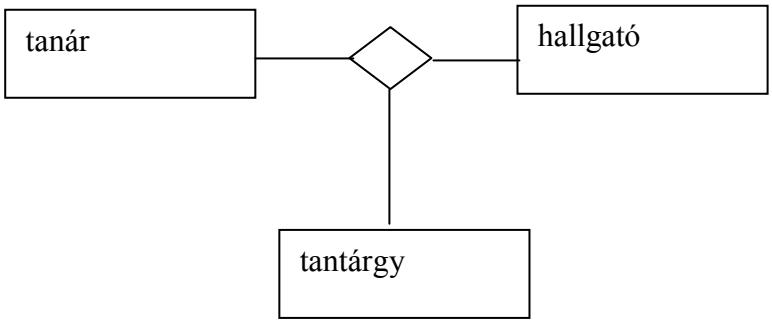
kutyája van. Attól lesz gazda, hogy van legalább egy kutyája.)
A vonalra a **multiplicitást** írjuk:
Egy-egy kapcsolat: ha nem írunk semmit a vonalra, az pontosan 1-1-et jelöl.
Írhatunk ilyet is: 0..1
Egy-sok kapcsolat: * vagy i..* vagy i..j
Sok-sok kapcsolat:



(Minden hallgató jár legalább egy tanfolyamra, és egy tanfolyam legalább 4 fős, legfeljebb 10 fős lehet.)
Reflexív asszociáció: amikor egy osztály saját magával van kapcsolatban.

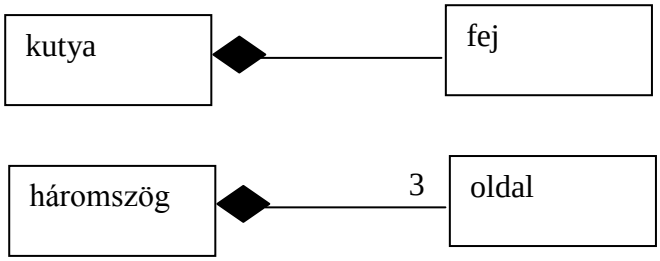


Többes asszociáció:

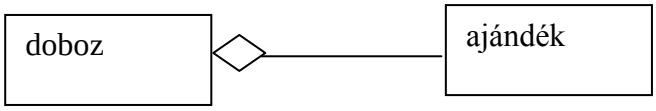


Aggregáció: Erősebb kapcsolat, mint az asszociáció. Egész-rész kapcsolat. Két fajtája van: gyenge és erős. A gyenge tartalmazásnál, ha elvágjuk a kapcsolatot, a részek akkor is „életképesek" maradnak, az erős tartalmazásnál (kompozíció) viszont külön-külön működésképtelenek.

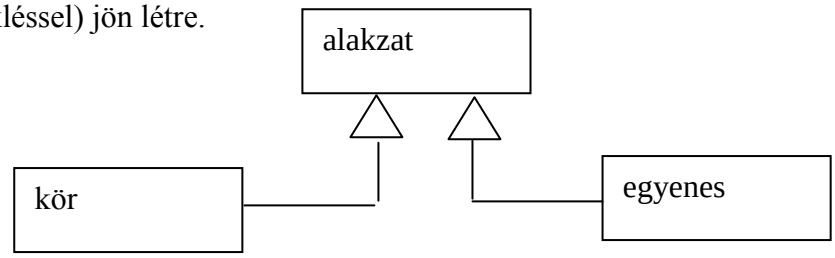
Kompozíció (erős tartalmazás):



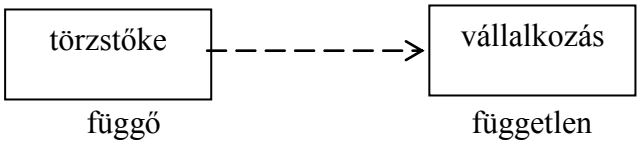
Gyenge tartalmazás:



Általánosítás: a reláció azt fejezi ki, hogy a speciális osztály az általánosból származtatással (örökléssel) jön létre.

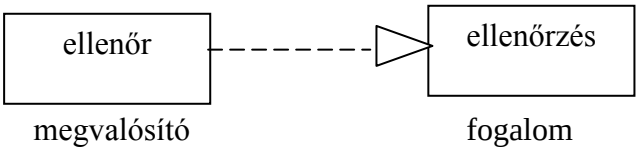


Függőség: Két elem közötti kapcsolat, ahol az egyik változása befolyásolja a másikat.

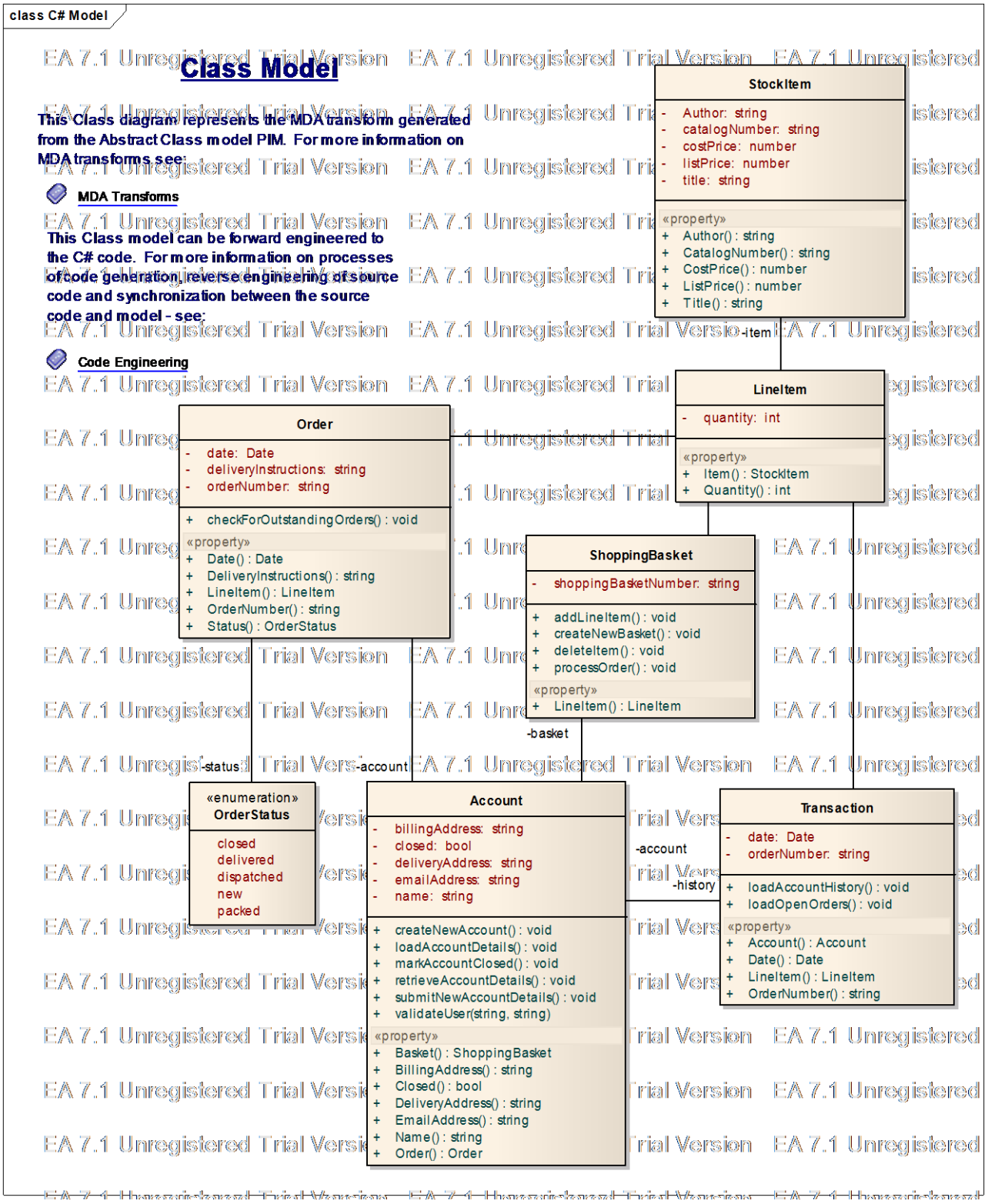


A vállalkozás fejlődésével/csődbe jutásával párhuzamosan változtathatja a törzstőkéje összegét.

Megvalósítás: A fogalom és annak megvalósítója közötti kapcsolat.



Egy összetettebb osztálydiagram az Enterprise Architect 7.1-ből:



Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: R. A. Maksimchuk – E. J. Naiburg: UML földi halandóknak. Kiskapu Kiadó, Budapest 2006. és Harald Störrle: UML 2. Panem Kiadó, Budapest 2007.)

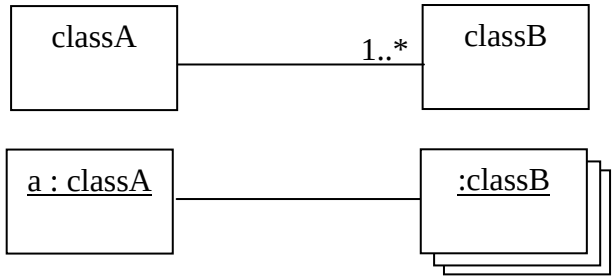
V.4. Objektumdiagram

A rendszerben egy adott időpillanatban szereplő objektumok pillanatképét jeleníti meg. Itt az osztályokat példányosítjuk, ezek a példányok lesznek az objektumok.

Az objektum rajzjele:

Az első objektum a hallgató osztály Péter nevű „példánya”.
A másik példa egy név nélküli, árucikk típusú objektumot mutat be, amit az attribútumai értékeivel jellemzünk. (Kód, név és ár attribútumai vannak.)

Az osztály- és az objektumdiagram kapcsolata:



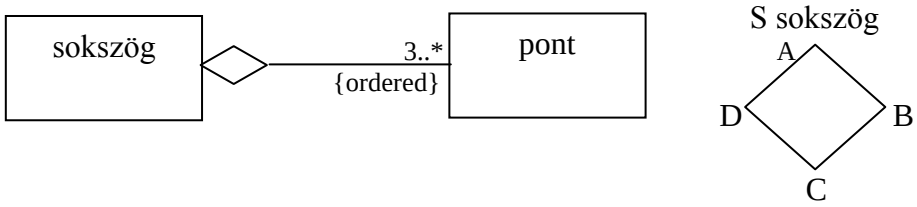
Az első rajz az osztálydiagram, ahol egy-sok típusú asszociációs kapcsolatban van a classA és a classB osztály. Ebből a második rajzon látható objektumdiagram készülhet, ahol az „a” nevű classA típusú objektum van összekapcsolva egy név nélküli, classB típusú multiobjektummal. Ami az osztálydiagramon reláció (itt a példában az asszociáció), azt az objektumdiagramon összekapcsolásnak nevezzük. Az objektumdiagram az osztálydiagram relációjának számosságát is mutatja, hiszen az „a” objektum „sok” (1..*) classB típusú objektummal van összekapcsolva.

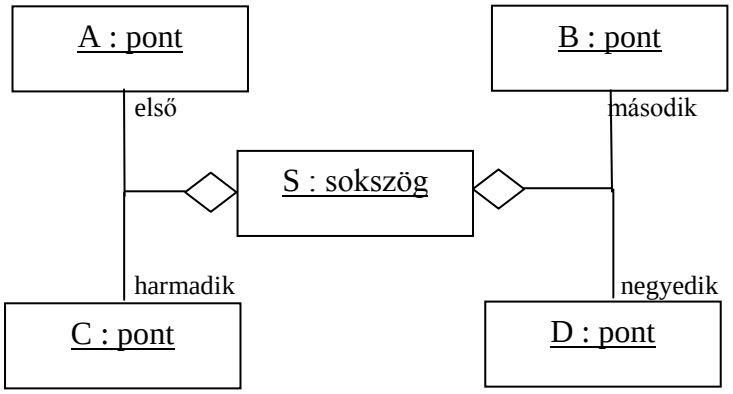
Másik példa:

Az első ábrán egy osztálydiagram látható, ahol a személy osztály saját magával van asszociációs kapcsolatban (reflexív asszociáció). Az anya/apa doboz neve **minősítő**, és az osztály objektumainak egy részhalmazát határozza meg. (Nyilván nem minden személy szülő, csak a személyek egy részhalmaza, csoportja.) A gyerek felirat az asszociációs vonal osztály felőli végén egy **szerepkör**.

A szerepkör azt írja le, hogy az adott relációban mi a szerepe a személy osztály tagjainak. Az asszociáció számossága: egy gyereknek 0, 1 vagy 2 (vér szerinti, élő) szülője van, és minden szülőnek *, vagyis akárhány (vér szerinti, élő) gyereke van. Ehhez az osztálydiagramhoz egy lehetséges objektumdiagram egy négytagú családot ábrázol, ahol Zsolt és Lujza személyeknél látható a minősítő (apa, illetve anya), Zoli és Évi pedig gyerekek (az ábrán gy betűvel rövidítettem a gyerek szerepkört, helyhiány miatt).
Látható tehát, hogy az objektumdiagramban az osztálydiagram osztálya(i)ból példányosított objektumok szerepelnek, olyan összekapcsolással, amit szintén az osztálydiagram relációi határoznak meg.

Még egy utolsó példa:

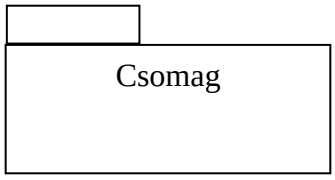




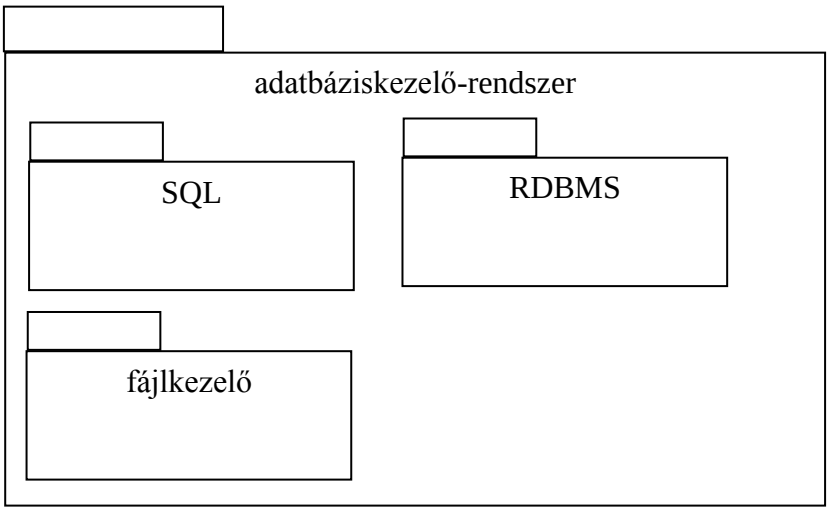
Az osztálydiagram sokszög és pont osztályát egy tartalmazás reláció köti össze. A reláció számossága 3..* vagyis egy sokszöghöz legalább 3 pont kell. Az {ordered} egy megszorítás, azt jelenti, hogy a pontok sorrendje fontos, nem cserélhetők fel.
Az osztálydiagram mellett látható egy S nevű négyszög. A négyszög által meghatározott objektumdiagramot készítettem el. Az {ordered} megszorítást az objektumdiagramban a tartalmazás relációk pont felőli végére írt szerepkör (első, második, harmadik, negyedik) helyettesíti.

V. 5. Csomagdiagram

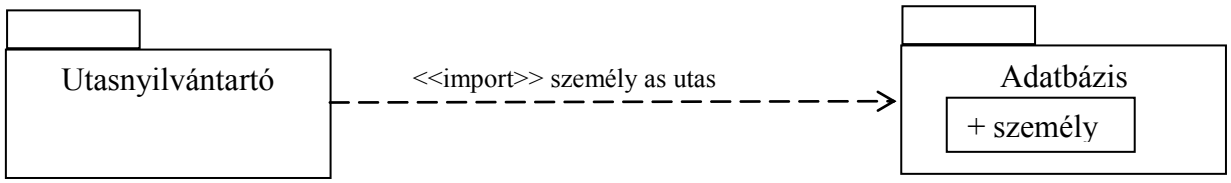
Az UML-elemek csoportosítására, közös névtérben való elhelyezésére alkalmas.
Csomag:



A csomagok tartalmazhatják akár egymást is:

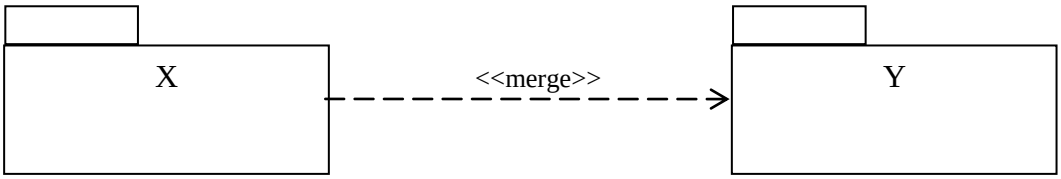


A csomagbeli elemekhez láthatóságot is rendelhetünk, de csak public és private-ot. Ha nincs megadva láthatóság, akkor az elem nyilvánosnak számít. A csomag összes nyilvános eleme minden más névtérből elérhető a teljes minősített név segítségével (teljes minősített név: gyökércsomag:_neve::csomag_neve::elemnév, ha a csomagokat egymásba ágyazzuk). A teljes minősített név használata azonban sokszor kényelmetlen. Egyrészt ezek a nevek nagyon hosszúak, másrészt a csomaghierarchia struktúrája így minden névbe bele lesz építve, ami a hierarchia átszervezését megnehezíti. Ezért kívánatos, hogy egyfajta relatív úttal is hivatkozni lehessen az elemekre. Ezt teszi lehetővé az **importkapcsolat:**



Az ábra Utasnyilvántartó csomagja importálja a személy osztályt (publikus/public – ezért van előtte a plusz jel) az Adatbázis csomagból, és egyúttal utasnak nevezi át a saját névterében.

Egy másik, csomagok között használható kapcsolatfajta a beolvasztás (merge):



Itt az X csomag olvasztja be Y-t, vagyis az Y csomag tartalmát Y::elemnév néven belerajzolhatjuk az X-be.

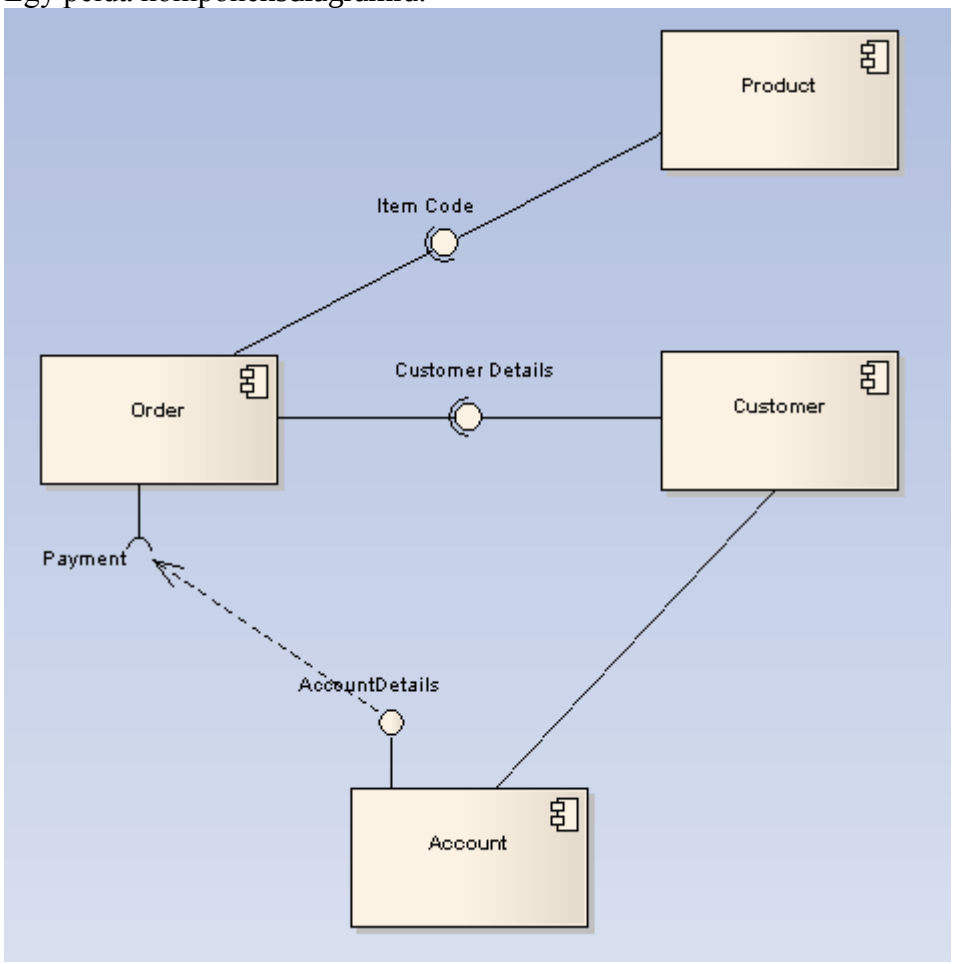
V.6. Összetevő diagram

Komponens: a rendszer fizikailag létező és kicserélhető része, feltéve, hogy az új komponens csatlakozási felülete (interfésze) megegyezik a régivel.

A komponens nagysága változó lehet, az egy-két osztályt tartalmazó kis méretű komponenstől az egész alrendszert tartalmazó nagy méretűig. (A komponens lehet egy EJB vagy Corba komponens is.)
A komponens egy egységbezárt, önálló, teljes és ezáltal cserélhető egység, amely függetlenül működtethető, telepíthető és összekapcsolható más komponensekkel.

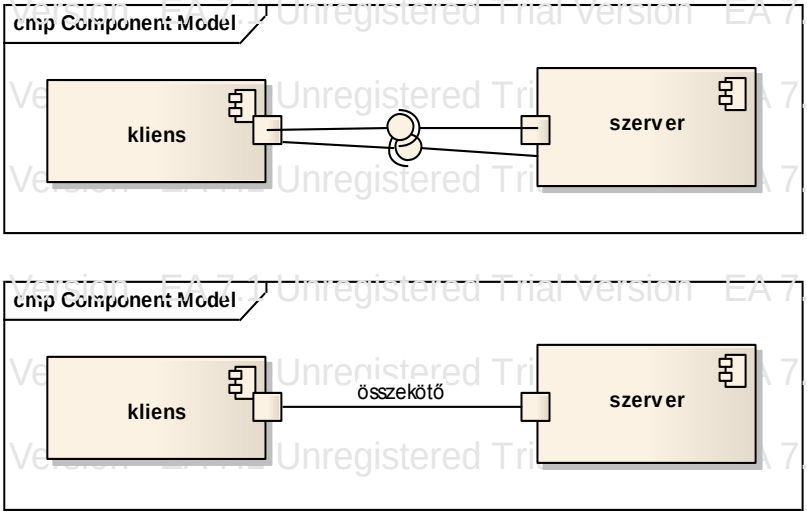
Az osztály és a komponens fogalma nagyon hasonló az UML-ben, már csak azért is, mert a komponens az UML-metamodellben (metamodell – ami leírja a modell és a benne található diagramok kapcsolatát és jelentését) osztályok egy alosztálya, tehát rendelkezik az osztályok összes jellemzőjével.

Egy példa komponensdiagramra:



A komponenseket vagy az ábrán a téglalapok jobb felső sarkában látható rajzjellel vagy a <<component>> sztereotípiával lehet megjelölni. A komponens rendelkezhet elvárt interfésszel (itt az Order (Rendelés) komponens rendelkezik három elvárt interfésszel), illetve nyújtott interfésszel (a gömb; az Account (Számla), a Customer (Vásárló) és a Product (Termék) komponensek rendelkeznek vele).
Az interfészeket **csatlakozóba** (Port) foghatjuk össze (a rajzjele egy kis téglalap). Egy csatlakozó a komponens összes interakcióját összefogja, a nyújtott és az elvárt interfészeket, sőt ezek használati protokollját is. Egy bonyolult csatlakozót akár állapotautomataként (State Machine) is fel tudunk majd rajzolni (amikor megismerkedünk az állapotautomatákkal).

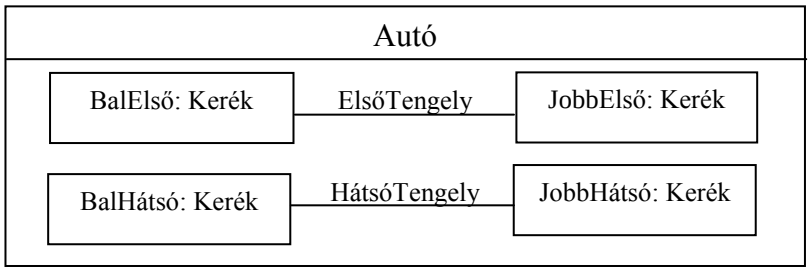
A következő két ábra ugyanazt jelenti. A kilens és a szerver nevű komponensek egy-egy összeillő nyújtott és elvárt interfésszel rendelkeznek. A második ábra ezt egyszerűbben mutatja be, az interfészeket és a kapcsolatukat egy összekötő (Connector) helyettesíti.



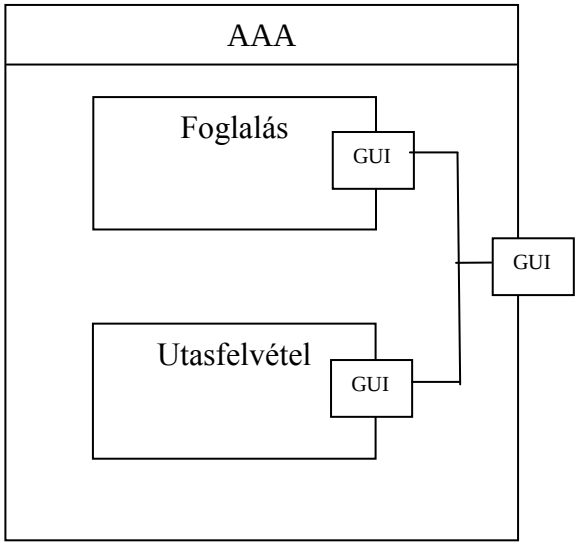
V.7. Összetett szerkezeti diagram

A teljes rendszer vagy egy összetettebb osztály felépítését írja le.

Strukturált osztály: az osztály belső szerkezetét is megmutatja.

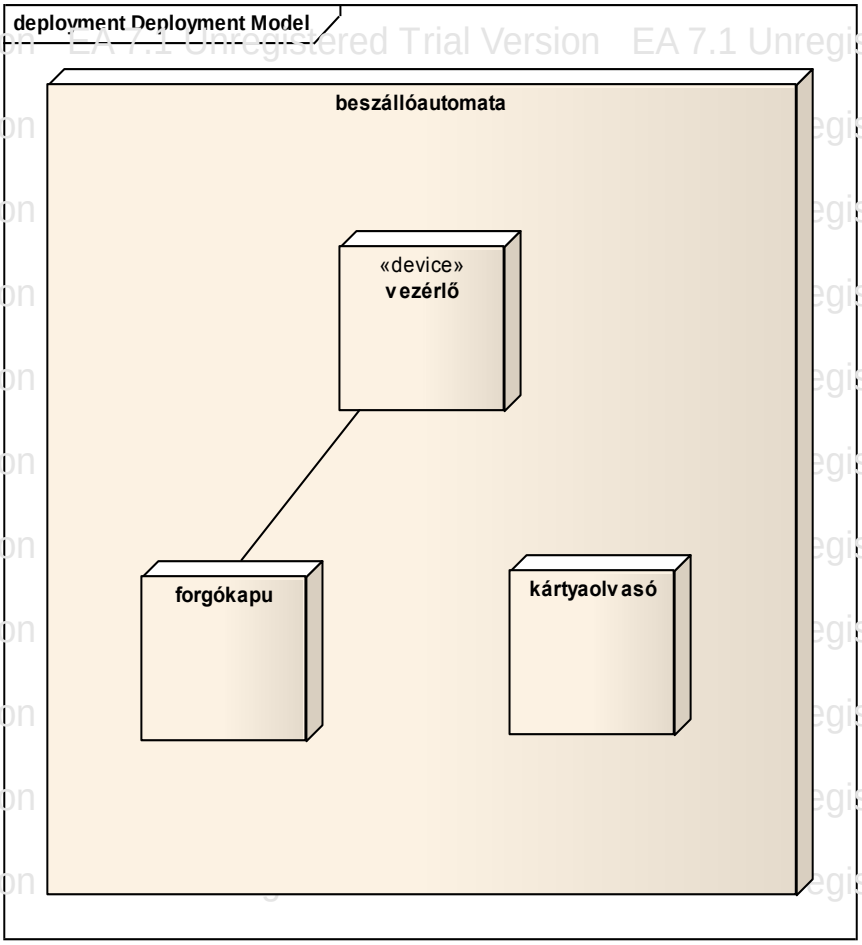


Rendszernél (egy légitársaság):
A portokra nevet is írhatunk, itt pl. GUI- Graphical User Interface.



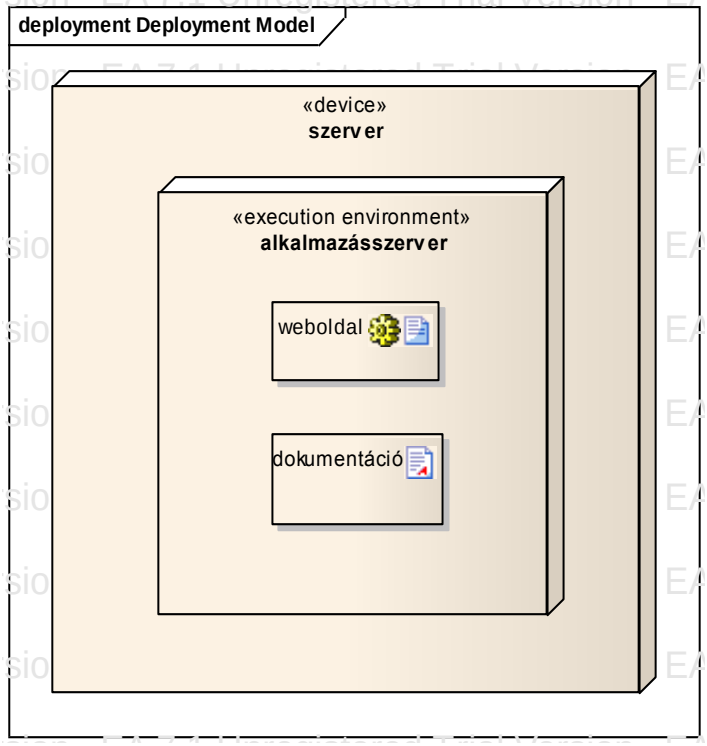
V.8. Kialakítási diagram

Minden modellezett szoftverrendszer végül egy számítógép vagy számítógép-hálózat által lesz végrehajtva. Szoftver alatt itt a kódot, a hozzá tartozó adatokat és a beállítási, konfigurációs fájlokat is értjük. A kész fizikai rendszer (szoftver- és hardverkomponensek) felépítését mutatja be ez a diagramfajta. Kétféle megközelítés létezik, az első szerint a rendszerstruktúrát hangsúlyozzunk:



Itt csomópontok (Node) vannak (a téglatest a rajzjele), amik között asszociációs kapcsolatok lehetnek. A <<device>> sztereotípiával ellátott csomópont egy fizikai eszközt jelent. A csomópontok neveit konkrétabban is megadhatjuk, pl.: bsza5 : beszállóautomata. Ebben az esetben az aláhúzás a csomópont példányosítását jelenti, ahogy az osztályokból konkrét objektumokat hoztunk létre, itt is hasonló dologról van szó.

A második megközelítés szerint a szoftverösszetevők rendszerre való kihelyezését hangsúlyozzunk, ilyenkor a telepítés részletei a lényegesek.
Ehhez a fajta kialakítási diagramhoz szükség van a műtermék (Artifact) fogalmára.
Műtermék (Artifact): az információ egy fizikai darabja, pl.: állományok, a futási idejű adatszerkezetek a memóriában, az adatbázistáblák, e-mailek, dokumentumok, stb.
Ezek a műtermékek helyezhetők ki a csomópontokra. Az ábrán az alkalmazásszerver csomópontra két műterméket helyeztünk ki, egy weboldal típusút és egy dokumentum típusút (természetesen más sztereotípiákat is meg lehet adni):

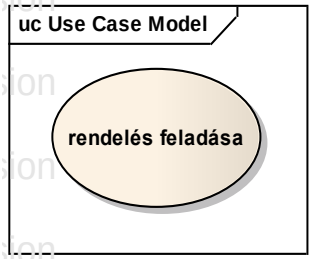


Ezzel megismerkedtünk az összes **szerkezeti diagrammal**. A továbbiakban **a viselkedési** vagy más néven **dinamikus diagramokkal** fogunk foglalkozni.

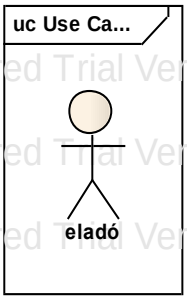
V.9. Használati eset (use-case) diagram

Ez a diagram a rendszer viselkedését írja le, ahogyan az egy külső szemlélő szemszögéből látszik. A rendszer belső szerkezetéről vagy viselkedéséről nem tesz említést, ezzel nem foglalkozik. Általában a fejlesztési ciklus elején készítjük. Használati eset diagram készülhet egy már meglévő rendszerről (hogy a működését jobban megértsük), vagy egy tervezett rendszerről (hogy összegyűjthessük a rendszerkövetelményeket, hogy megmutathassuk a megrendelőknek, felhasználóknak, és hogy a kész rendszer teszteléséhez is használhassuk majd).

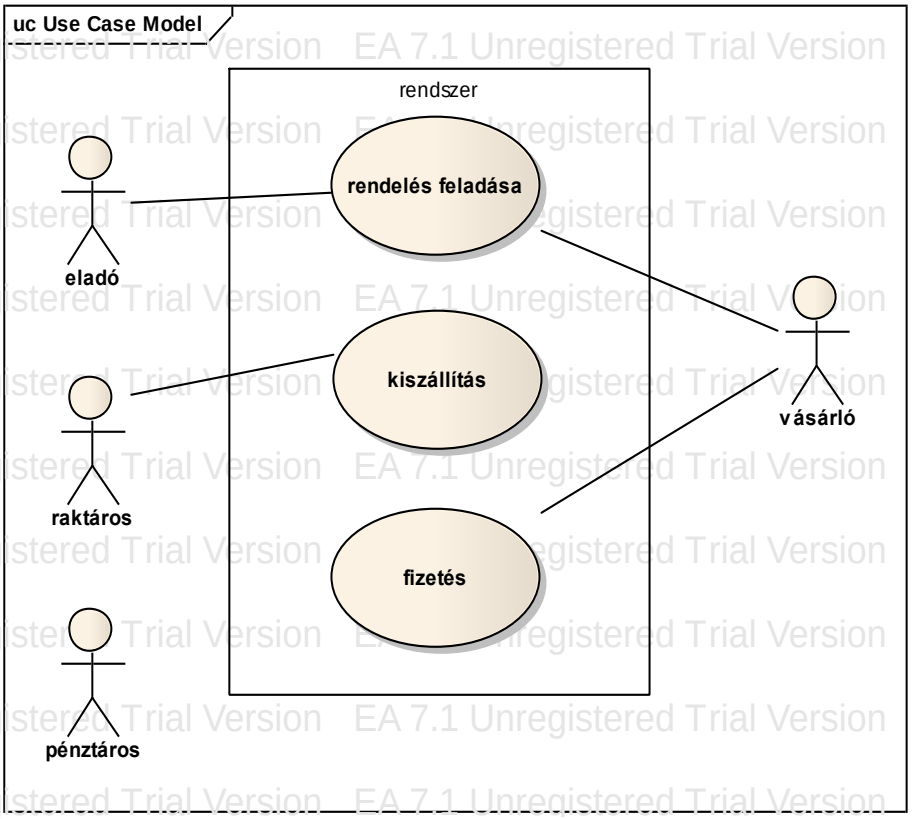
A használati eset diagram a következő összetevőkből áll:
Használati eset: tevékenységek sorozata, amelyet a rendszer végre tud hajtani a szereplőkkel kommunikálva. Rajzjele az ellipszis, amibe vagy alá odaírjuk a nevét.



Szereplő (Actor): személy, csoport, szervezeti egység vagy fizikai eszköz, aki vagy ami kapcsolatba lép a rendszerrel. Rajzjele egy pálcikaemberke.

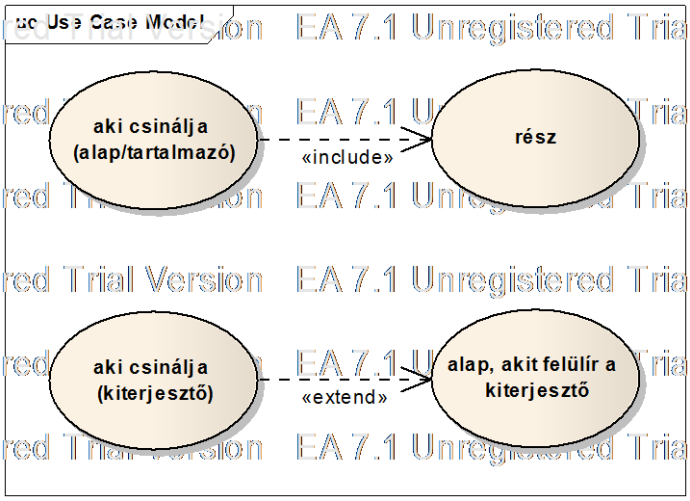


Rendszerhatár (Boundary): a megvalósítandó rendszer és a szereplők közötti határ.



Természetesen a szereplők, a use-case-ek és szereplő-use-case között kapcsolatok is lehetnek. A szereplők és a use-case-ek között általában asszociációs kapcsolat van, ahogy az a fenti ábrán is látszik.
Use-case-ek között **<<include>> és <<extend>> kapcsolat** szokott leggyakrabban előfordulni. Az <<include>> kapcsolat azt jelenti, hogy egy résztevékenységet kiemelünk az alap use-case tevékenységsorozatából és azt külön use-case-ben tüntetjük fel. Ezt a résztevékenységet aztán más use-case-ek is használhatják. Az <<extend>> kapcsolatnál a kiterjesztő megszakíthat egy másik use-case-t a működésében. A megszakított nem is tud a megszakító létezéséről (mint a patch-programoknál). Ezek választható feladatok, ellentétben az <<include>> feladatokkal, amik

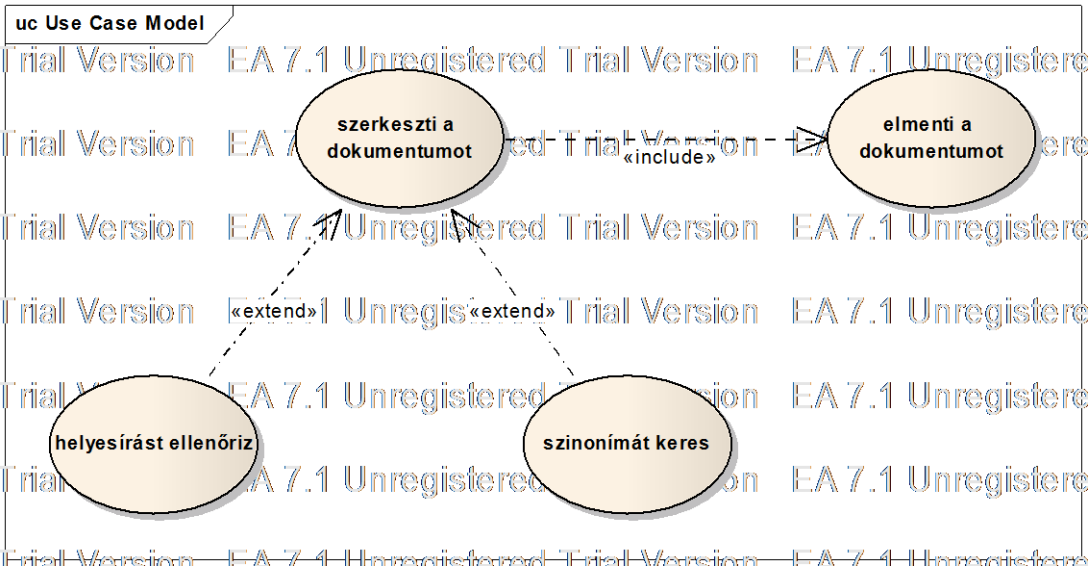
kötelezőek.
Az ábrán látható, hogy a nyíl mindig attól indul, aki csinálja azt a bizonyos kapcsolatfajtát (aki include-olja a másikat vagy aki kiterjeszti, megszakítja a másik működését):



Az <<include>> és az <<extend>> kapcsolat megkülönböztetésére jó módszer a következő négy kérdés megválaszolása:

1. Szabadon választható-e a feladat (a use-case által ábrázolt tevékenységsorozat)?
2. Az alapfeladat teljes-e a feladat nélkül is?
3. Feltételhez kötött-e a feladat végrehajtása?
4. A feladat megváltoztatja-e az alapfeladat viselkedését?

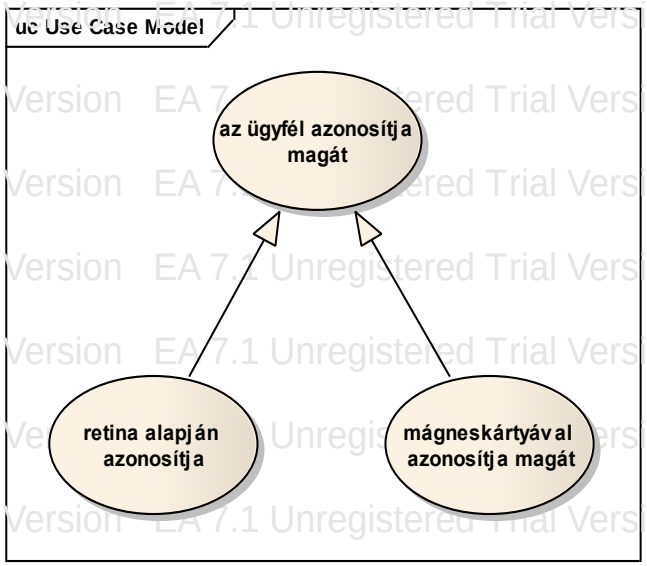
Azért jók ezek a kérdések, mert <<include>> kapcsolat esetén mindre NEM-mel lehet válaszolni, <<extend>> kapcsolat esetén viszont mindre IGEN-nel.
Nézzünk erre egy példát:



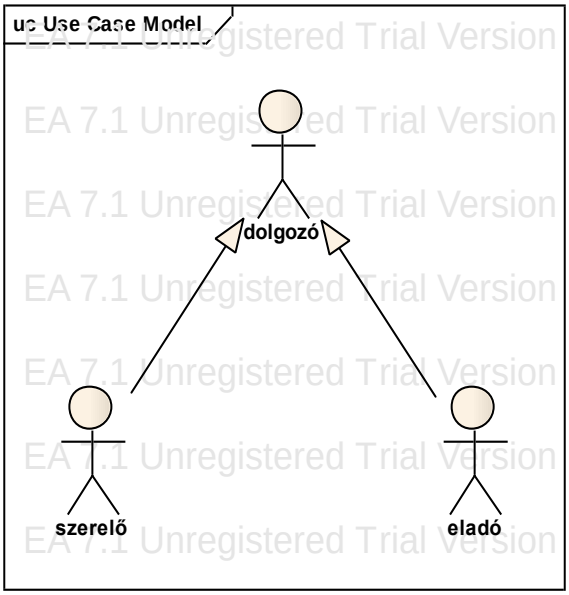
A példa egy szövegszerkesztő alkalmazás (mondjuk a Word) működésének egy részletét mutatja be. A szerkeszti a dokumentumot a fő tevékenység. Ebből kiemeltük az elmenti a dokumentumot. Tegyük fel, hogy még csak most készítjük a diagramot és még csak a use-case-ek vannak meg, a

kapcsolatok nem. Tegyük fel az első kérdést: szabadon választható-e a feladat? Gondoljunk bele: a dokumentum elmentése funkció nélkül mire lenne jó egy szövegszerkesztő? Ebből adódik a második kérdésre is a válasz: ha ilyen fontos a mentés, akkor nem teljes nélküle az alapfeladat. A harmadik kérdésre a válasz kevésbé egyértelmű, hiszen minden szövegszerkesztő rákérdez, hogy akarja-e menteni a dokumentumot. A negyedik kérdésre viszont ismét egyértelmű választ adhatunk, hiszen a mentés nem változtatja meg a dokumentum szerkesztés lépéseit, mert szervesen közéjük tartozik ő is. Tehát a négy kérdésből háromra egyértelmű nem-mel tudtunk válaszolni, ezért ez biztosan <<include>> kapcsolat lesz.
A helyesírást ellenőriz és a szinonímát keres feladatok közül csak az elsőt nézzük meg, gondolom senki számára nem meglepő, hogy ez a két tevékenység eléggé hasonló logikával működik a modellezés szempontjából.
A helyesírás ellenőrzés szabadon választható-e? Persze, hiszen ha én azt gondolom, hogy tudok helyesen írni, vagy nincs is az adott nyelvhez helyesírás ellenőrző modul, akkor is tudok szöveget szerkeszteni. Az alapfeladat teljes-e a feladat nélkül. Az előbbi válasz alapján: igen. Feltételhez kötött-e a feladat végrehajtása? Itt megint érvelhetünk az igen és a nem válasz mellett is.(Igen, mert be kell kapcsolni a helyesírás-ellenőrző funkciót.) A feladat megváltozatja-e az alapfeladat viselkedését? Igen, hiszen ha bekapcsoljuk ezt a funkciót, az általunk beírt szöveget meg fogja változtatni, vagy pirossal aláhúz egyes szavakat, vagy ha az automatikus javítás is be van kapcsolva, akkor át is írhatja az általunk begépelteket. Tehát mivel itt is három kérdésre határozottan igen-nel lehet válaszolni, ezért ez a kapcsolat <<extend>> típusú lesz.

A use-case-ek között előfordulhat még általánosítás kapcsolat is, pl. egy beléptető-rendszer tervéből egy részlet:



A személyek között is megadhatunk általánosítás kapcsolatot, pl. egy számítástechnikai bolt és szerviz dolgozói lehetnek eladók vagy szerelők:



Végül nézzük meg, hogyan ellenőrizhetjük, hogy a use-case diagramunk jó-e! Erre használhatjuk a **MiSzHaT-próbát.**

Mi – A feladat azt írja le, hogy Mit kell csinálni és nem azt, hogy hogyan?
Sz – A feladatot a Szereplő nézőpontjából írtuk le és nem a rendszeréből?
Ha – A feladat végrehajtása Hasznos a szereplő számára?
T – A use-case diagram Teljes forgatókönyvet ír le, nem maradt-e ki valami?

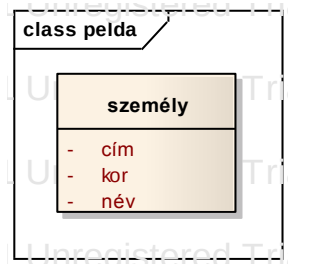
Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: R. A. Maksimchuk – E. J. Naiburg: UML földi halandóknak. Kiskapu Kiadó, Budapest 2006. és Harald Störrle: UML 2. Panem Kiadó, Budapest 2007.)

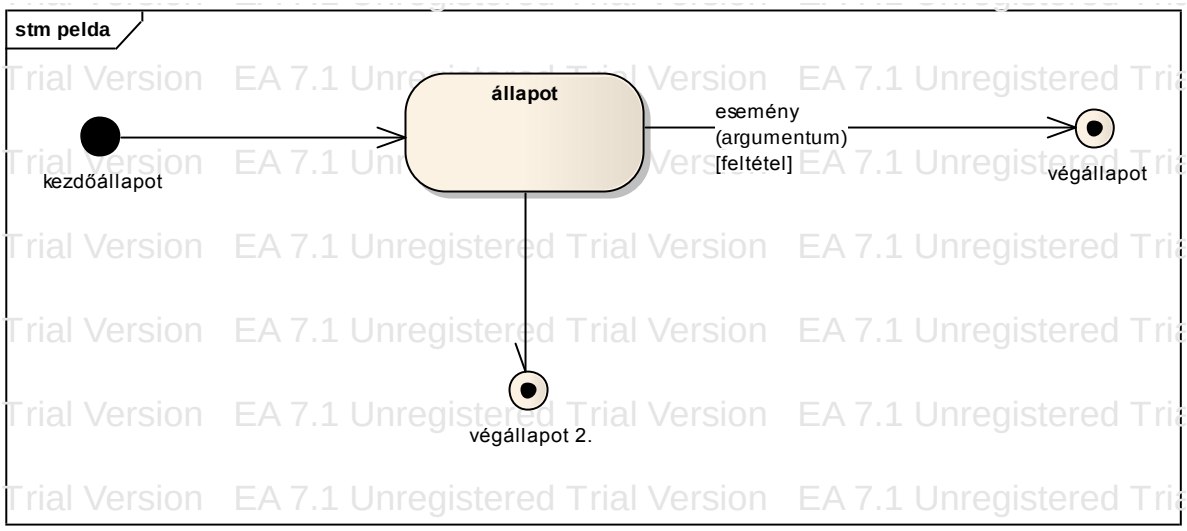
V.10. Állapotautomata

Az állapotautomata (state machine) vagy állapotgép az osztályok objektumainak, a használati eseteknek és a protokolloknak a dinamikus viselkedését mutatja, vagy a dialógusok lefutásának leírására is alkalmas.
Ezek közül mi csak az objektumok állapotaival fogunk foglalkozni.
Állapot: az objektum állapotát az attribútumai konkrét értékeinek n-esével jellemezzük.
Esemény: tevékenység, történés, ami valamely objektum állapotát megváltoztatja. Ha az esemény végrehajtása időben elhúzódik, megkülönböztethetjük tőle a pillanatszerű akciót.
Nézzünk egy példát a két fogalomra:

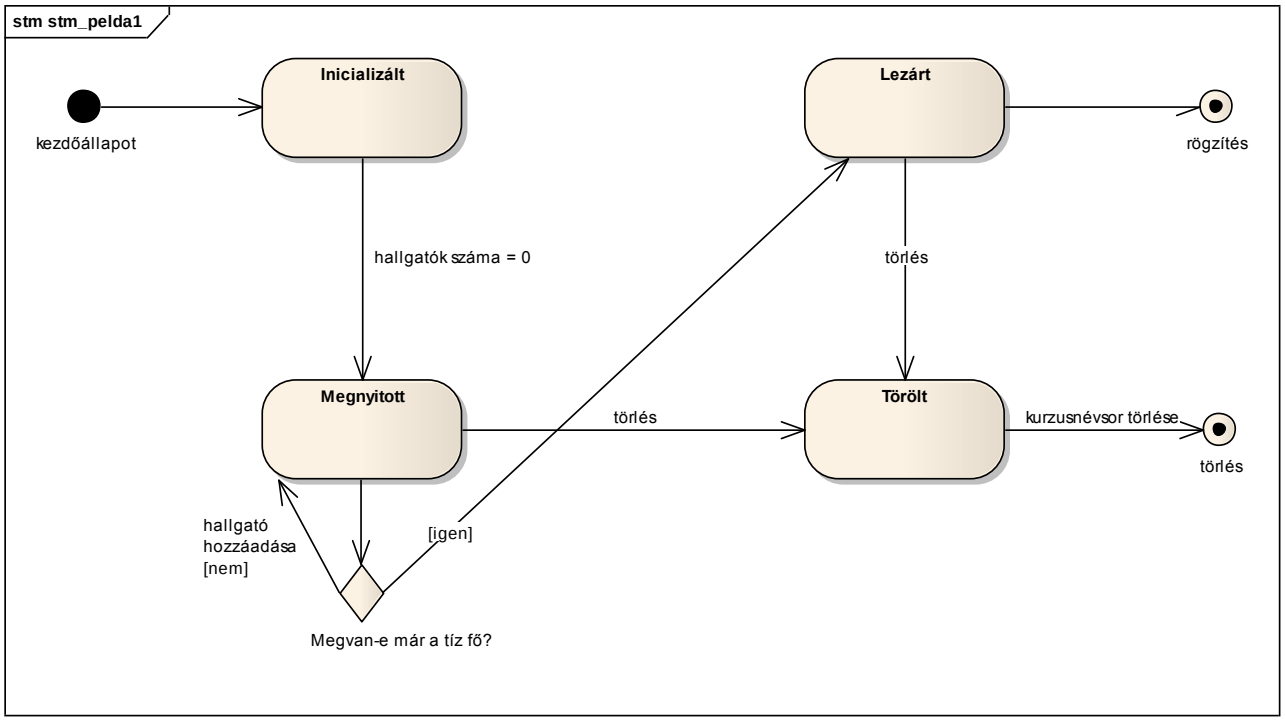


Szabó Zsolt
Budapest, Fő utca 2.
25

A személy nevű osztály három tulajdonsággal (attribútummal) rendelkezik: van neve, címe és életkora. A belőle létrehozott objektumokat ezen attribútumok értékeinek a hármasával jellemezhetjük. Az osztálydiagram alatt látható egy konkrét személy típusú objektum egy lehetséges állapota.
Milyen események változtathatják meg ezen objektum állapotát? Pl. a névváltoztatás esemény, a költözés vagy a születésnap.
Az állapotautomata diagram állapotokon kívül állapotámeneteket tartalmazhat. **Állapotátmenet: két állapot közötti kapcsolat, amely kifejezi, hogy egy adott állapotban lévő objektum egy esemény vagy valamely feltétel bekövetkezésének hatására milyen másik állapotba kerül.**
Az állapotautomata egy kezdőállapotot és egy vagy több végállapotot is tartalmazhat. Az állapotátmeneteket szimbolizáló nyilakra ráírhatjuk az állapotváltást okozó esemény vagy akció nevét, illetve a feltételt:

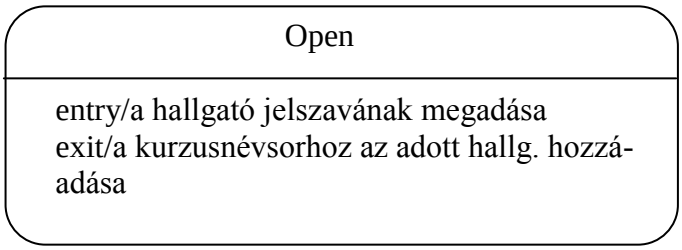


Nézzünk egy ETR-szerű rendszert példaként, ahol a hallgató jelszóval tud bejelentkezni és felvenni egy tárgyat. A példában egy csoportban maximum 10-en lehetnek:

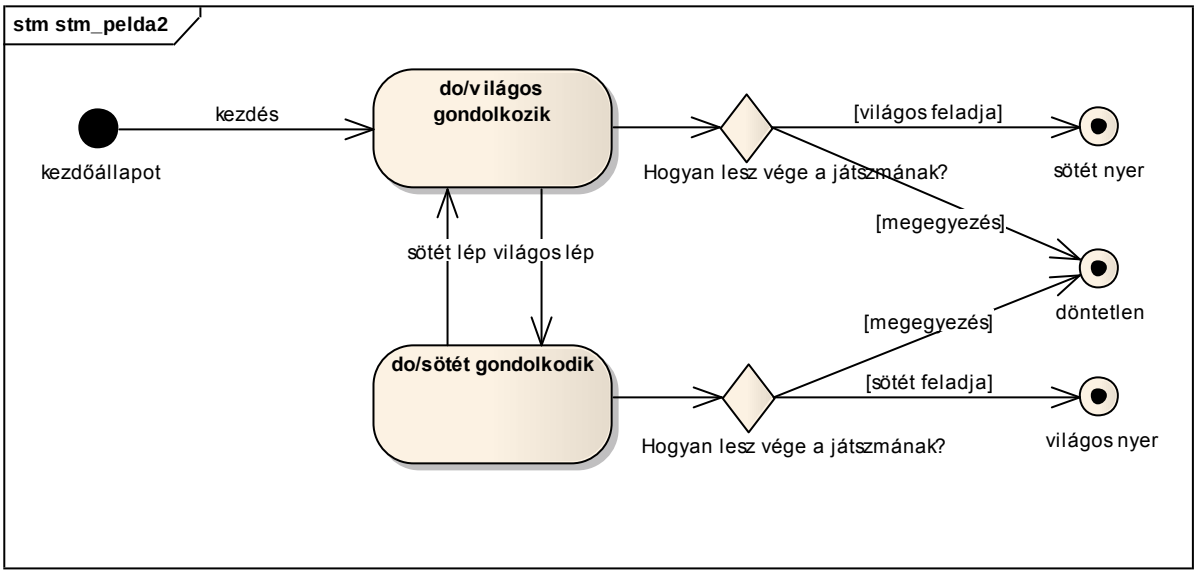


A rombusz az esetválasztó csúcs. Azokra az átmenetekre, amelyek az esetválasztó csúcsból indulnak, szögletes zárójelbe helyezett feltételt kell írni. Ha az esetválasztó csúcsból kiinduló átmenetek feltételei kölcsönösen kizárják egymást és együttesen lefednek minden lehetséges esetet, akkor a feltételt írhatjuk az esetválasztó csúcshoz.
Az objektumok az adott állapotban tevékenységeket is végezhetnek. Három fázist különböztetünk meg: **entry** – amikor az objektum belép az adott állapotba, akkor milyen tevékenységet végez, **do** – amikor az adott állapotban van és **exit** – amikor kilép az adott állapotból.

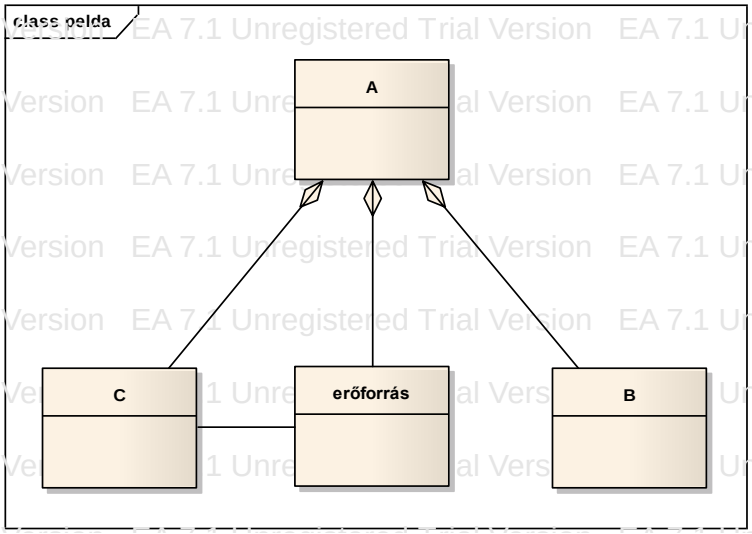
Nézzük az előző példa Open állapotát:



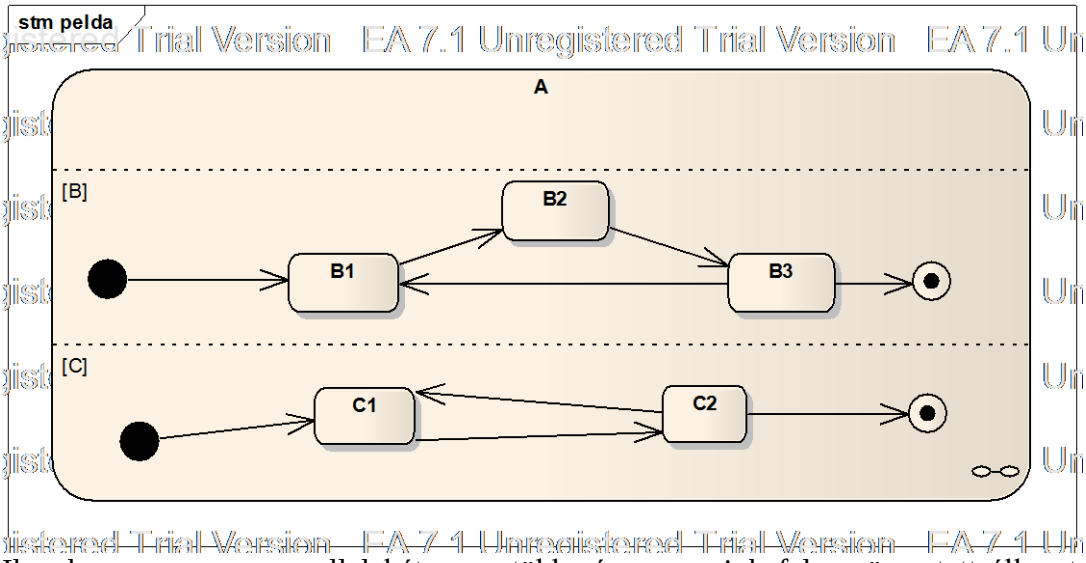
Előfordulhat, hogy egy állapotot azzal tudunk legjobban jellemezni, hogy milyen tevékenységet folytat abban az állapotban az objektum. Ilyenkor nem adunk külön nevet az állapotnak, hanem a do fázist használjuk megnevezésként. Nézzük például a sakkjátszmát:



A néhány állapotot és állapotátmenetet tartalmazó diagramok még áttekinthetőek, de hamar eljutunk addig, amikor már túl bonyolult lenne az ábra. Ennek megoldására összetett állapotokat lehet létrehozni, kétféle módon. Az összetett állapot olyan állapot, amely alállapotokat tartalmaz.
Az első módszer az **állapotok aggregációja** nevet viseli. Nézzük meg ezt egy példán keresztül! Tegyük fel, hogy a következő osztálydiagramhoz készítünk állapotautomatát:

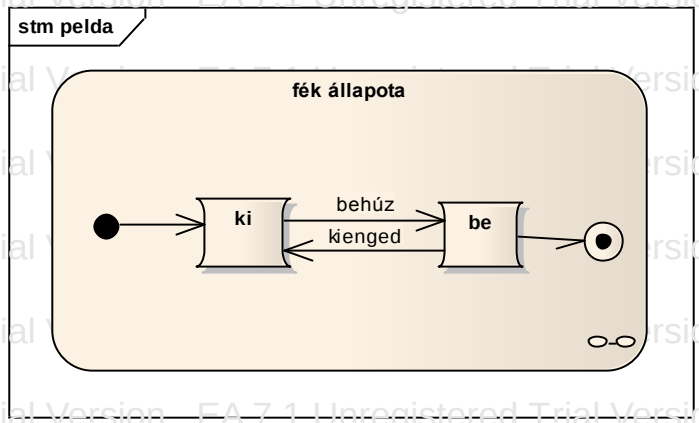


Az A osztály által szimbolizált rendszer B és C alrendszerekből áll, amik egy erőforrást akarnak elérni. Az erőforrást egyszerre csak az egyik használhatja, így konkurálnak egymással. Az A objektumának az állapotát nyilván a B és a C rész állapota is befolyásolja, mégpedig ezek **egymással párhuzamosan** kerülhetnek különféle állapotokba. Ezt így ábrázolhatjuk:

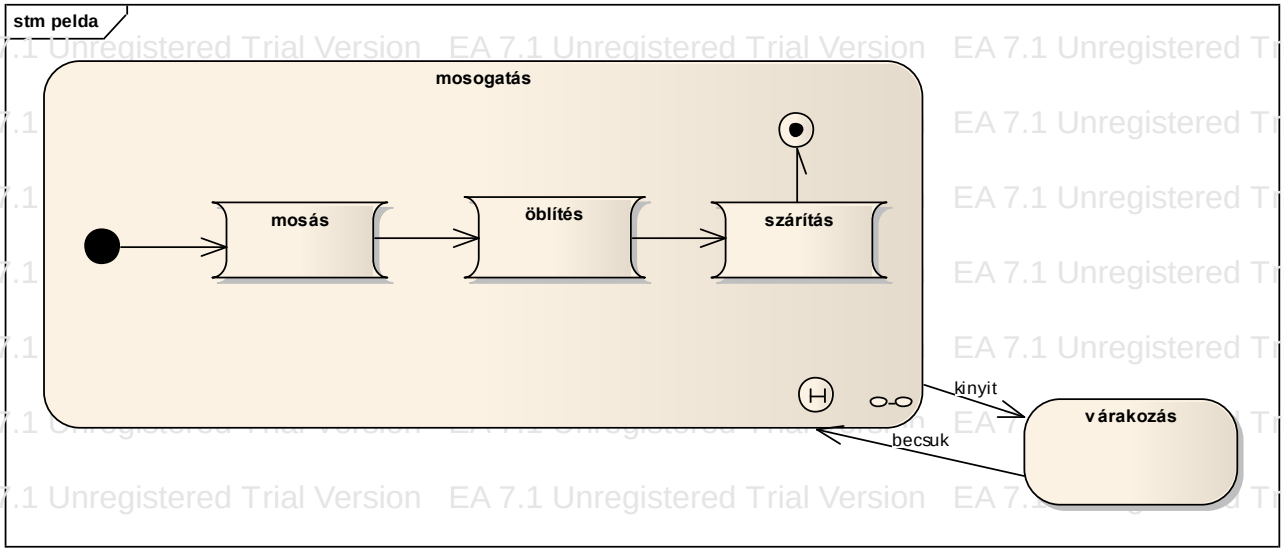


Ilyenkor szaggatott vonallal két vagy több részre osztjuk fel az összetett állapotot és a sávokba rajzoljuk bele az egymással párhuzamosan működő állapotautomatákat. Az A objektum állapotát a B és a C objektum állapota együtt határozza meg.

A második módszer a **diszjunkt szub- vagy alállapotok** nevet viseli. Itt az alállapotok sorban, szekvenciálisan követik egymást, az összetett állapotot mindig csak egy határozza meg közülük. Pl.:



Még egy speciális állapottal kell megismerkednünk ennél a diagramfajtánál, az **emlékező vagy történeti állapottal (history state).** Ha egy összetett állapotból kilépünk és szeretnénk majd ugyanott folytatni a tevékenységet, ahol abbahagytuk, akkor van szükségünk erre az állapotra. Létezik **egyszerű történeti állapot**, amely csak az állapotkonfiguráció legfelső szintjét őrzi meg (az alállapotoknak is lehetnek további alállapotai, ezek több szintűen is egymásba ágyazhatók), és létezik a **mély történeti állapot**, amely teljes mélységében megőrzi az állapotkonfigurációt. Az egyszerű történeti állapotot egy bekarikázott H betű jelzi, a mélyet pedig egy bekarikázott H*. Nézzünk egy példát a történeti állapot használatára is, a mosogatógépét:

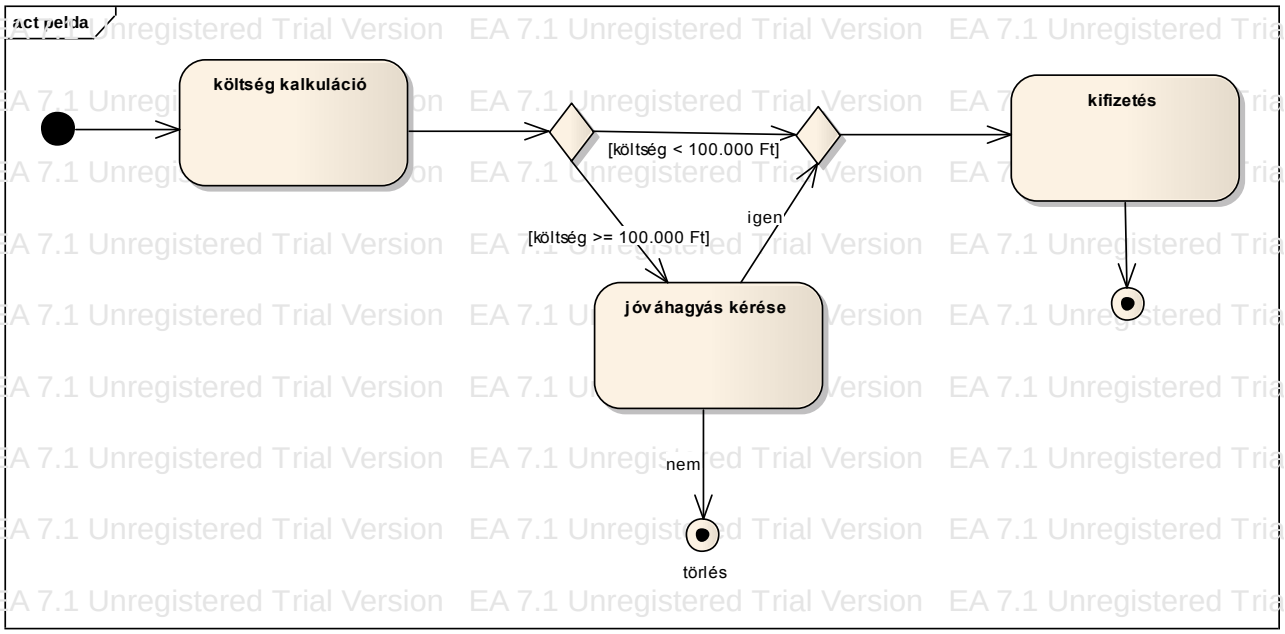


Ha a mosogató-program futása közben kinyitjuk a gép ajtaját, akkor várakozás állapotba kerül, de amikor újra becsukjuk az ajtaját, ott tudja folytatni (abban az alállapotban), ahol abbahagyta.

V. 11. Tevékenységdiagram

A probléma megoldásának a lépéseit szemlélteti, a párhuzamosan zajló vezérlési folyamatokkal együtt. Az állapotautomata egy változatának is tekinthető, ahol az állapotok helyére a végrehajtandó tevékenységeket tesszük, az állapotátmenetek pedig a tevékenységek befejezésének eredményeként valósulnak meg. Hasznos az üzleti vagy munkafolyamatok modellezésére, használati esetek vagy konkrét algoritmusok lefutásának leírására.

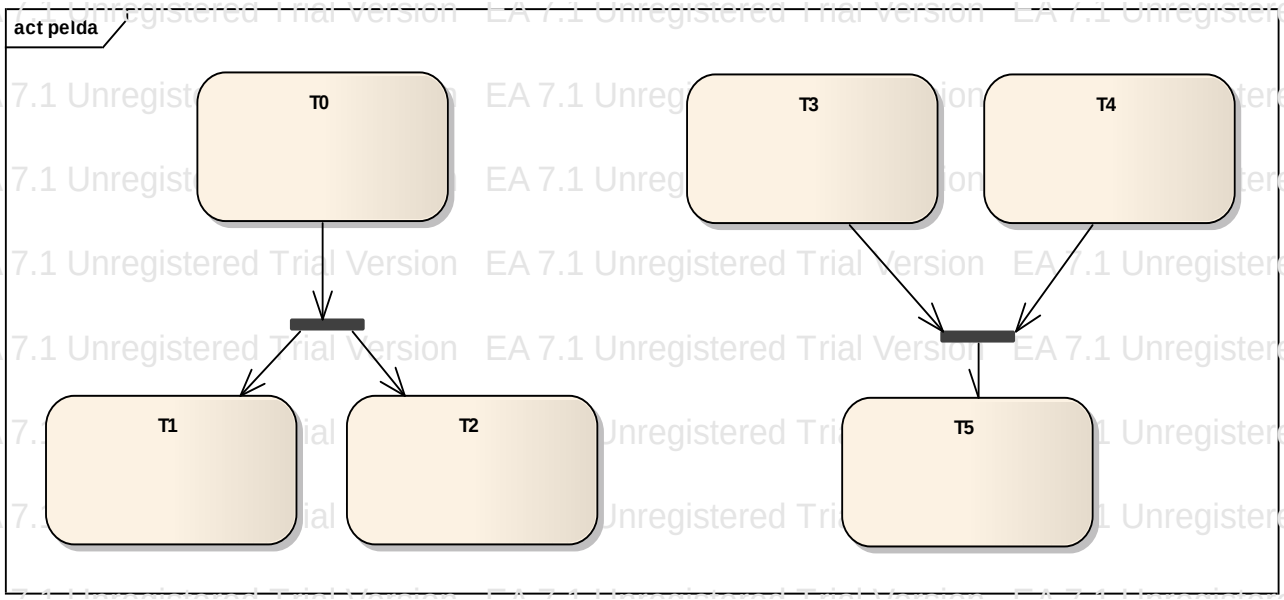
Nézzünk egy konkrét példát tevékenységdiagramra, amely egy cég beszerzési szabályait mutatja: ha a 100.000 Ft-ot nem haladja meg a beszerzendő anyag ára, akkor azt kifizetik, de ha az összeg eléri a 100.000 Ft-ot, akkor ahhoz előbb igazgatói jóváhagyást kell hozzá kérni.



Láthatóan valóban nagyon hasonlít az állapotautomatára. Elágazást az állapotautomatában is

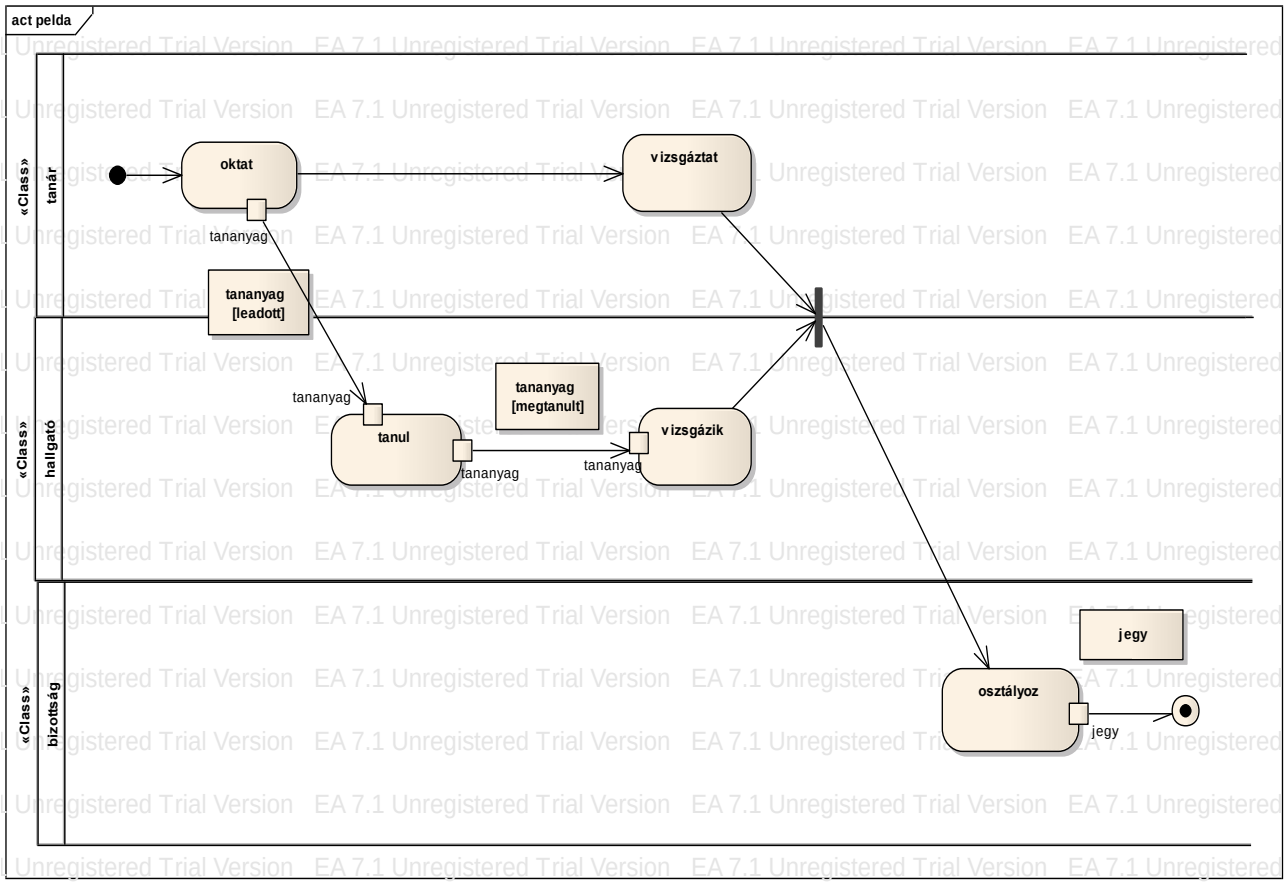
ugyanígy kell csinálni (sarkára állított négyzet)!

A tevékenységeknél vannak olyan feladatok, amelyek egyidejűleg, egymással párhuzamosan végezhetők, és néha bizonyos műveletek végrehajtása előtt más feladatokat be kell fejezni (pl. a házépítésnél nem kezdhetik el addig a tetőt, amíg a falak nem állnak). Az ilyen feladatok ábrázolására szolgál a **szinkronizáció:**



Az ábra szerint a T0 tevékenység befejeződése után egyszerre, párhuzamosan indulhatnak a T1 és a T2 tevékenységek (ezt **elágazásnak** (fork) nevezik); a T3 és a T4 tevékenységek is egymással párhuzamosan mennek, és amikor mindkettő befejeződik, akkor indulhat a T5 (ezt **csatlakozásnak** (join) nevezik).

A következő ábrán két új dolgot is szeretnék bemutatni. Az egyik újdonság a sávok vagy **partíciók** használata, a másik pedig az, hogy a tevékenységek közé a tevékenység által létrehozott vagy az általuk megváltozatott állapotú objektumokat is be lehet rajzolni. Ezt objektum-folyamnak (object-flow) nevezzük. Szögletes zárójelben megadhatjuk az objektum állapotát is, amennyiben azt a tevékenységek megváltoztatják és a modell szempontjából ezt fontosnak tartjuk jelezni.
Az ábra a szigorlat közbeni tevékenységeket mutatja be:

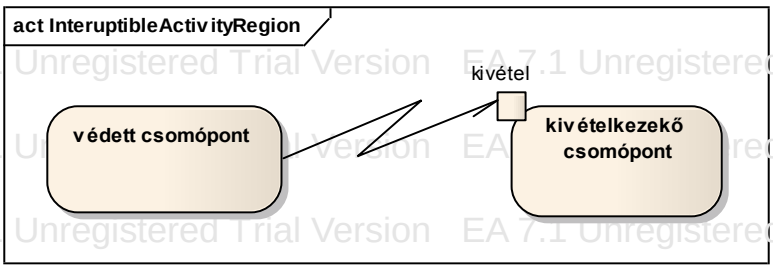


Kivételek

Eddig jól, szabályosan lefutó tevékenységeket modelleztünk, de előfordulhat, hogy feldolgozási hiba miatt egy tevékenységet meg kell szakítani, hogy a hiba kezelése a tevékenységen kívül végbemehessen.

A kivétel tulajdonképpen egy jól definiált, nemlokális vezérlési ág. A kivételkezelésnek két része van: egyrészt a kivételt ki kell váltani, másrészt el kell kapni és kezelni kell.

A dolog logikáját a következő ábra mutatja:



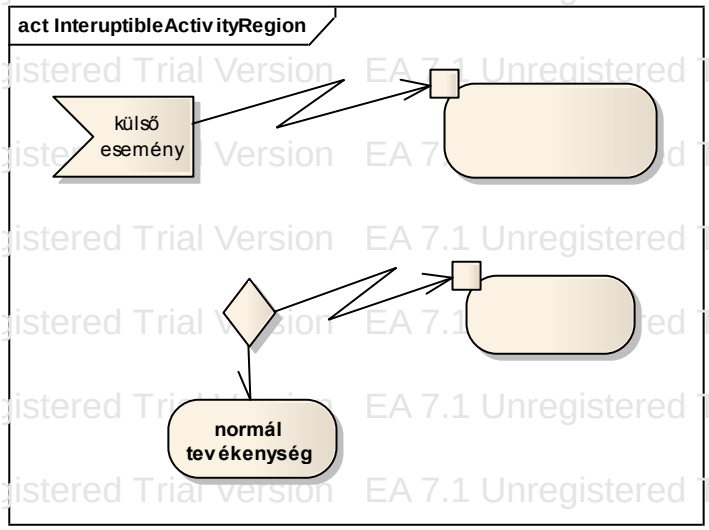
A védett csomóponton belül kiváltódhat egy kivétel (pl. az almenü kezelése során a felhasználó megszakítja a tevékenységet) és ezt a kivételt egy másik csomópontban, a kivételkezelőben tudjuk feldolgozni (a példa esetén a főmenübe ugrunk és ott folytatódik a program futása).

Kivételek kiváltásának négyféle módja van:

Külső esemény	Valamilyen esemény lép fel a feldolgozás alatt lévő tartományban és ez kihat a feldolgozás lefutására. Pl.: ez a helyzet az operációs rendszeren belüli folyamatváltáskor.

Időpont	Egy időpontot érünk el, és ezért speciális feldolgozás válik esedékessé. Pl.: az internetes jegyfoglalási folyamat bizonyos idő után inaktivitás miatt megszakad.
Esetválasztás	Egy kivétel kiváltódhat még célzottan, esetválasztás eredményeképpen is, pl. ha olyan hibát fedezünk fel, amelyet nincs lehetőség helyben (lokálisan) kezelni.
Tevékenység (közvetlen)	Végül pedig egy kivételt egy normál tevékenység is kiválthat, pl. ha a fenti három ok valamelyike miatt kiváltott kivétel után kivétel objektum előállítása szükséges.

A tevékenység által közvetlenül kiváltott kivételt már bemutatta az előző ábra. Az időpont által kiváltottnak a rajzjele egy homokóra (a szabvány szerint), de ezt az Enterprise Architect nem támogatja. A külső esemény és az esetszétválasztás okozta kivételkiváltás így nézhet ki:



V.12. Kölcsönhatási diagramok

A sorrenddiagram, a kommunikációs, az időzítés és a kölcsönhatás áttekintő diagram mind a **kölcsönhatási diagramok** csoportjába tartoznak. Az első három fajta tulajdonképpen ugyanazt ábrázolja, csak kicsit másra vannak kihegyezve.

A sorrenddiagram üzenetváltásokat ábrázol, amelyek több, kölcsönhatásban lévő partner között zajlanak le. A partnerek lehetnek osztályok, aktorok, komponensek, csatlakozók és csomópontok. Ezt akkor érdemes használni, ha kevés résztvevő (partner) van, de azok sok üzenetet küldenek.

A **kommunikációs diagram** szintén erre szolgál, de őt akkor érdemes választani, ha sok résztvevő van és azok viszonylag kevés üzenetet küldenek egymásnak.

Az időzítésdiagram akkor megfelelő, ha a résztvevők állapotainak komplex időbeli egymásra hatását akarjuk szemléltetni. Ez is csak kevés résztvevő esetén praktikus.

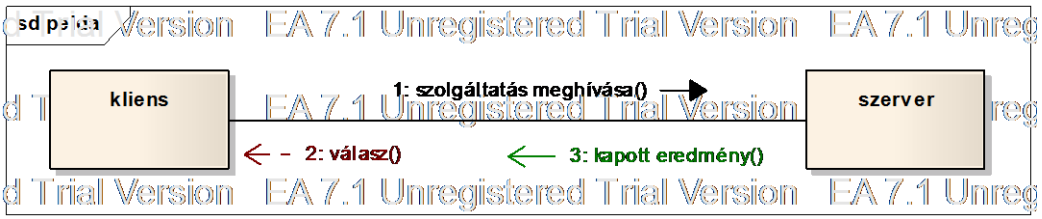
A kölcsönhatás áttekintő diagram kicsit kilóg a sorból, mivel ő a fenti három fajta módon megrajzolt (de ugyanarra a rendszerre vonatkozó) diagramok között fennálló összefüggéseket a tevékenységdiagramok vizuális eszközeivel fejezi ki. Ő valóban egy áttekintést nyújt.

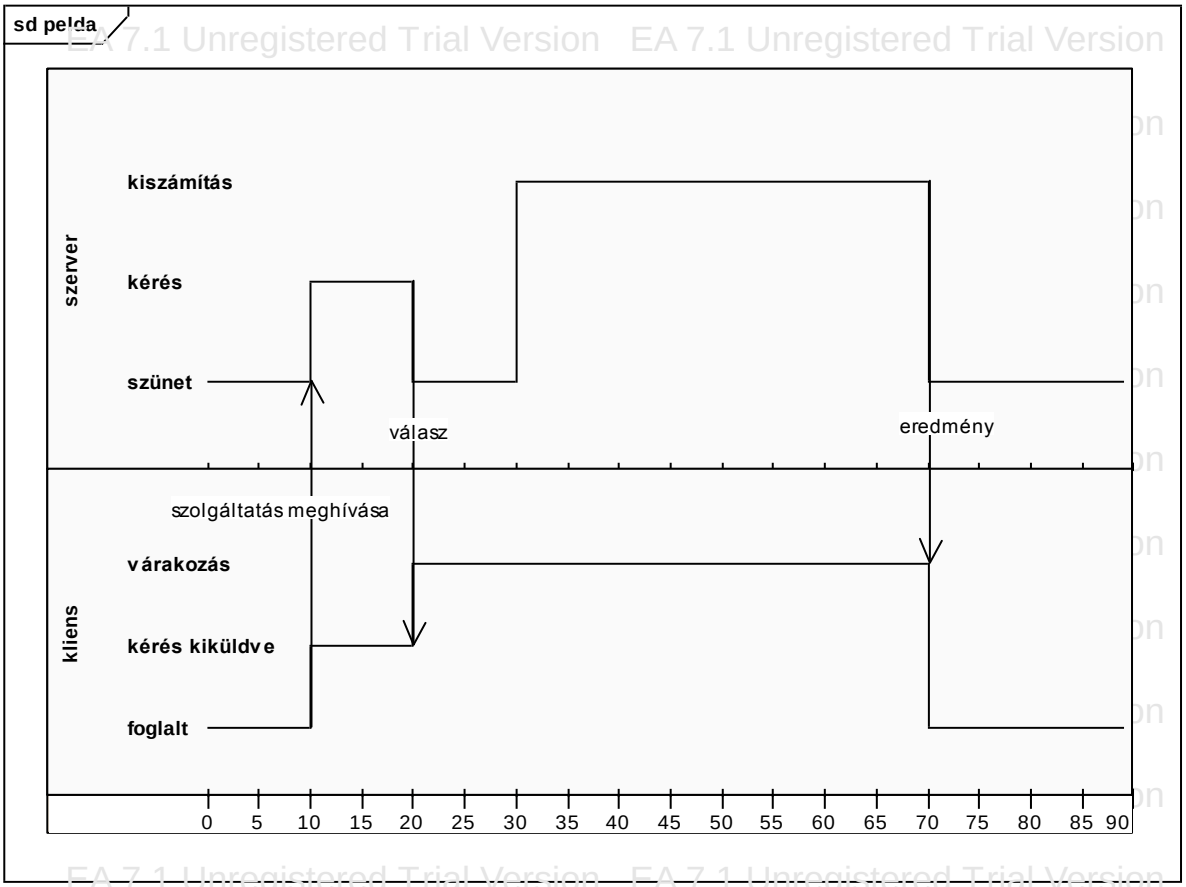
Mivel ez a négy diagramtípus ilyen erősen összefügg, ezért együtt tárgyalom őket.

Üzenettípusok:
Az első háromfajta kölcsönhatási diagramban egyaránt a következő háromfajta üzenettípust használjuk.
Kérés: Szinkron üzenet. Kérés alkalmával a küldő szünetelteti aktivitását (vár) és a fogadó aktiválása következik be.
Válasz: Szinkron üzenet. A kérést küldő a válaszüzenet hatására újra aktiválódik.
Szignál: Aszinkron üzenet, vagyis a küldő nem szünetelteti az aktivitását az üzenet elküldése után, hanem tovább dolgozik.

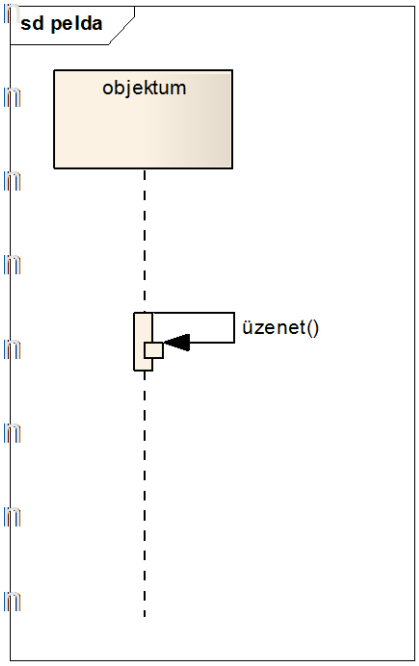
A szinkron és az aszinkron üzenet közötti különbséget egy példával szemléltetném: a levélküldés aszinkron, a telefonálás pedig szinkron típusú. Vagyis a levél elküldése után a feladó foglalkozhat mással, viszont egy telefonbeszélgetés mind a „küldőt” (hívó), mind a „fogadót” (hívott) lefoglalja a beszélgetés végéig.

A kérés rajzjele egy kifestett hegyű nyíl, a válaszé egy szaggatott vonalú nyíl és szignálé egy folyamatos vonalú, normál végű nyíl.
Nézzük meg ugyanazt az üzenetváltást a három diagramtípussal (első a sorrend, aztán a kommunikációs és az időzítés). Először a kliens egy kérést küld a szervernek. Aztán a szerver válaszol, majd később egy szignál üzenettel elküldi a kért számítás eredményét.

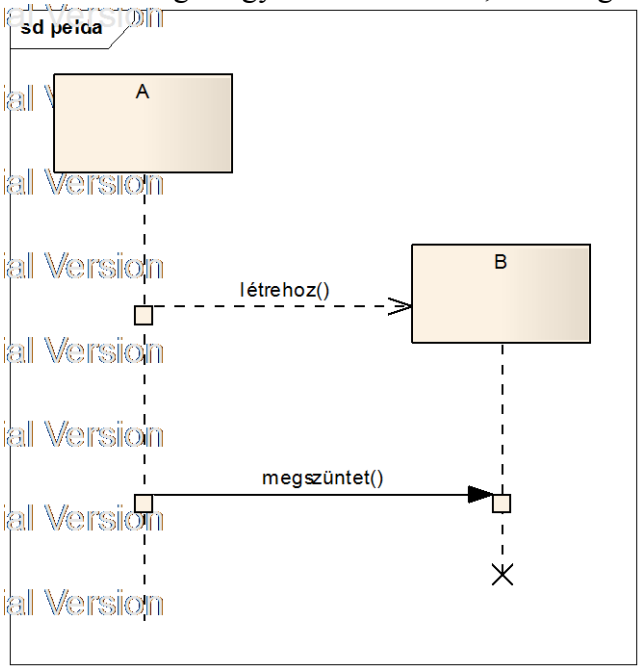




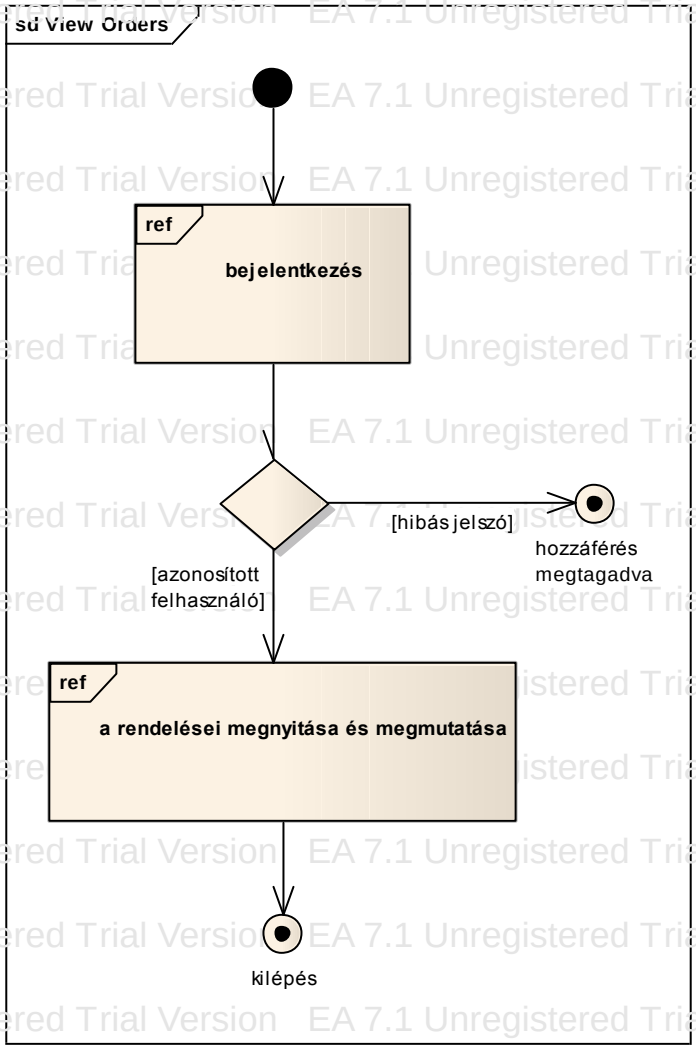
A sorrenddiagramban látható függőleges szaggatott vonalat **életvonalnak** nevezzük. Minden objektum vagy aktor, aki vagy ami részt vesz az üzenetküldésben rendelkezik egy ilyen életvonallal. Az életvonalon látható üres téglalapot **aktivációs résznek** nevezzük, ilyenkor csinálhat valamit az adott szereplő.
Egy objektum saját magát is aktiválhatja:



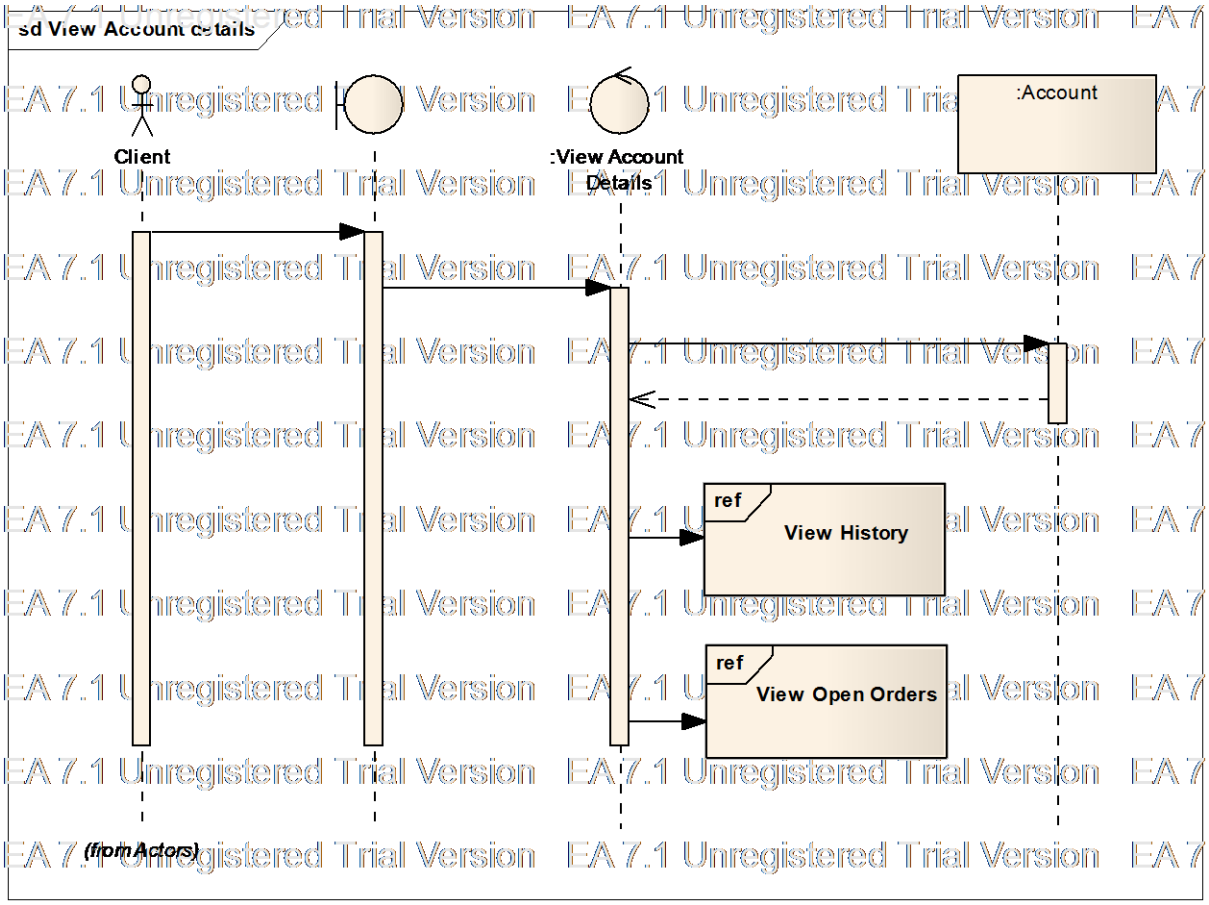
Egyik objektum **létrehozhatja vagy megszüntetheti** a másikat: (A megszűnő B objektum életvonala végén egy keresztet látunk, ez a megszűnés rajzjele.)



A sorrenddiagram, a kommunikációs és az időzítésdiagram áttekintése után nézzük meg a **kölcsönhatás áttekintő diagramot** is. Ez olyan, mintha egy tevékenységdiagramot rajzolnánk (kezdőpont, végpontok, elágazás, stb.), de a tevékenységek helyett ref-ek állnak (**referenciák**). A referenciák általában egy sorrenddiagramra vonatkoznak, amit már részletesen megrajzoltunk. A példában mindkét ref mögött létezik egy-egy ugyanilyen nevű sorrenddiagram, ami megmutatja a bejelentkezés és a felhasználó megrendeléseinek a részleteit.



Előfordulhat olyan kölcsönhatás áttekintő diagram is, ahol a sorrenddiagramba ágyazzuk be ezeket a ref-eket, mert ott nem akarjuk részletesen kifejteni a ref-ek által szimbolizált interakciókat. (Ez a példa az Enterprise Architect példája.)
(A ref-en kívül további interakciós operátorok is léteznek, pl. loop - ciklus, opt – opcionális, alt – alternatívák, brk – megszakítás, break, strict – szigorú sorrend, stb. Ezeket arra használhatjuk, hogy az interakciók egy részét megismételtethessük, vagy választhatóak legyenek, stb. Ezek azonban nem részei a számon kérendő tananyagnak. Akit érdekel, elolvashatja Harald Störrle: UML 2 című könyvének 12.4.-es fejezetében, ahol példákat is talál a használatukra.)


sd View Account Details
Client
:View Account Details
:Account
ref
View History
ref
View Open Orders
(from Actors)

)

Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: : Ian Somerville: Szoftverrendszerek fejlesztése. Második, bővített, átdolgozott kiadás, Panem Kiadó, Budapest 2007.*)*

VI. A jó szoftver tulajdonságai

VI.1. Nem funkcionális tulajdonságok

A szoftverterméket az általa nyújtott szolgáltatásokon kívül számos egyéb tulajdonság is jellemez, amelyek tükrözik a szoftver minőségét. Ezek a tulajdonságok nincsenek konkrét kapcsolatban a szoftver által végzett tevékenységgel, hanem inkább a program működés alatti viselkedését, szerkezetét, a forráskód szervezését és a kapcsolódó dokumentációt jellemzik. Ezek a **nem-funkcionális** tulajdonságok. Ide tartoznak pl. a szoftver válaszideje egy felhasználói kérdésre, vagy a programkód érthetősége.

A szoftvertől elvárható tulajdonságok jellegzetes halmaza nyilvánvalóan függ az alkalmazás típusától. Pl. egy banki rendszernek elsősorban biztonságosnak kell lennie, egy interaktív játéknak könnyen irányíthatónak, egy telefonhívás-kapcsoló rendszernek megbízhatónak, stb. Ezek a tulajdonságok négy fő csoportba gyűjthetők:

Terméktulajdonság	Leírás
Karbantarthatóság	A szoftvert oly módon kell megírni, hogy fokozatosan követni tudja az ügyfél igényeinek változásait. Ez elég kritikus tulajdonság, mert az üzleti környezet változásának elkerülhetetlen következménye a szoftver változása.
Üzembiztonság	Magában foglalja a megbízhatóságot, a biztonságosságot és a védelmet. Az üzembiztos szoftver rendszerösszeomlás esetén sem okozhat fizikai vagy gazdasági kárt.
Hatékonyság	A szoftver nem pazarolhatja a rendszer-erőforrásokat (memória, processzoridő).
Használhatóság	A szoftvernek használhatónak kell lennie anélkül, hogy a megcélzott felhasználói rétegnek indokolatlan erőfeszítéseket kellene tennie ennek érdekében. Vagyis megfelelő felhasználói interfésszel és kielégítő dokumentációval kell rendelkeznie.

A szoftvertervezési módszerek, eszközök és technikák általában a karbantartható és üzembiztos szoftverek előállítására fókuszálnak. A hatékonyság nagymértékben függ a szakterület-specifikus ismeretektől, míg a használhatóságról majd később, részletesebben beszélek.

VI. 2. A folyamatmenedzsment alaptétele

Folyamat: Az ember azon cselekedete, amely során a rendelkezésre álló módszerekkel, eszközökkel és felszereléssel a nyersanyagból (input) olyan terméket állít elő (output), amely értéket képvisel a vevőszámára.
A folyamatmenedzsment alaptétele: A termék minőségét jórészt az határozza meg, hogy milyen érettségű az a folyamat, amelynek során kifejlesztése és fenntartásra kerül.

Éppen ezért, ha a szoftver előállítási folyamatot tudjuk javítani, akkor annak egy csomó pozitív vonzata lesz:

- Jobb minőségű lesz a termék

- Nő a hatásfok
- Nő az eredményesség
- Csökkennek a költségek
- Flexibilisebbek lesznek a folyamatok
- Elégedettebbek lesznek az alkalmazottak

A folyamat minőségét a **CMM (Capability Maturity Model)** írja le. Ez egy USÁ-ban elterjedt minőségbiztosítási tanúsítvány, Európában inkább az ISO-t használják. Ennek ellenére Magyarországon is vannak CMM minősítéssel rendelkező cégek.

A CMM 5 szintet ír le:

1. Kezdetleges. A szoftverfolyamat esetleges, kaotikus. Kevés részfolyamat van definiálva, a siker az egyéni erőfeszítésen és a hősiességen múlik. A szoftverfolyamat egy fekete doboz. Nem láthatóak a projekt belső folyamatai.

2. Megismételhető. Léteznek az alapvető menedzsment folyamatok, a határidők, a költségek és a funkciók követése. Ez a szint már lehetővé teszi, hogy egy hasonló témájú projekt sikeres legyen.

3. Jól meghatározott. A szoftverfolyamat lépései dokumentáltak, szabványosítottak és integrálódtak a folyamatba.

4. Szervezett. Ezen a szinten kezdünk el mérhető adatokat gyűjteni a folyamatról és a termékről.

5. Optimalizált. A folyamatokról érkező visszajelzések és az innováció segítségével folyamatosan javítjuk a folyamatot.

VI. 3. A szoftvertervezés főbb kihívásai

A bizalom kihívása: A távoli szoftverrendszereknél, amelyeket weblapokon vagy webszolgáltatási interfészeken keresztül érünk el (és esetleg neten keresztül fizetünk is), fontos, hogy megbízhassunk bennük.
A heterogenitás kihívása: A rendszerek egyre inkább igénylik a hálózaton keresztüli, több különböző számítógépen és különböző operációs rendszereken történő osztott működés lehetőségét. Itt a feladat az, hogy olyan technikákat fejlesszenek ki, amelyek üzembiztosan, rugalmasan képesek összekötni a különböző architektúrájú komponenseket.
A szállítás kihívása: A hagyományos szoftvertervezési technikák többsége igen időigényes, hosszú idő szükséges a kívánt szoftverminőség eléréséhez. Az üzleti élet azonban gyorsan változik, így az üzleti folyamatokat támogató szoftvereknek is gyorsan kell változniuk. Itt a feladat a nagy rendszerek gyors, de megfelelő minőségű előállítása.
A fenti három dolog nem független egymástól, pl. egy meglévő rendszer gyors megváltoztatásához szükség van a hálózaton keresztüli hozzáférésre.

VII. Szoftverkövetelmények

VII.1. A követelmények fajtái

A szoftvertervezők által megoldandó problémák gyakran összetettek, így nehéz pontosan leírni, hogyan kellene működnie a rendszernek. A szolgáltatások és megszorítások leírásai a **rendszer követelményei,** ezen szolgáltatások és megszorítások kitalálásának, elemzésének, dokumentálásának és ellenőrzésének a folyamatát **pedig a követelmények tervezésének** nevezzük.

A követelmény szó elég általános, ezért érdemes két szintre bontani a követelményeket: a felhasználói és a rendszerkövetelményekre. **A felhasználói követelmények** diagramokkal kiegészített természetes nyelvű leírások arról, hogy milyen szolgáltatásokat várunk el a rendszertől és annak milyen megszorítások mellett kell működnie. Ezek magas szintű, absztrakt követelmények. Ez a leírás az ügyfelek és a fejlesztők képviselői (menedzserek) számára készülnek, akik nem rendelkeznek részletes technikai ismerettel a rendszerről. **A rendszerkövetelmény-specifikáció** a rendszer által végzendő tevékenység részletes, precíz leírása, amely a vezető technikai személyzetnek és a projektvezetőknek szól, és a rendszer vásárlója és a fejlesztő közötti szerződés része lehet.

A szoftvertervezés számos problémája a követelményspecifikáció pontatlanságaiból ered. A rendszerfejlesztő számára természetes, hogy úgy értelmezzen egy kétértelmű követelményt, hogy közben annak megvalósítását egyszerűsítse. (Viszont nem biztos, hogy az ügyfél ezt akarta.)

Elvben a rendszer funkcionális követelményeit leíró specifikációjának **teljesnek és ellentmondásmentesnek** kell lennie, a gyakorlatban nagyméretű, összetett rendszereknél ez lehetetlen.

A követelményeket gyakran felosztják funkcionális és nemfunkcionális, illetve szakterületi követelményekre.

A funkcionális követelmények. A rendszer által nyújtandó szolgáltatások ismertetései, hogy hogyan kell reagálnia a rendszernek bizonyos bemenetekre.

Nemfunkcionális követelmények. A funkciókra és szolgáltatásokra tett megszorítások. Gyakran a rendszer egészére vonatkoznak.

A nemfunkcionális követelményeket a következőképpen csoportosíthatjuk:
Nemfunkcionális követelmények:
- termékkövetelmények
    - használhatósági
    - hatékonysági (teljesítmény, tárterület)
    - megbízhatósági
    - hordozhatósági
- szervezeti (a fejlesztő szervezet, vállalat)
    - telepítési
    - implementációs
    - szabvány
- külső követelmények
    - együttműködési
    - etikai
    - törvényi (adatvédelmi, biztonsági)

A nemfunkcionális követelményekkel kapcsolatos általános probléma, hogy nehéz a meglétüket ellenőrizni.
Pl.: „Rendszercél: A rendszernek a tapasztalt ellenőrök számára könnyen használhatónak kell lennie, és a felhasználói hibák száma a lehető legkisebb legyen.
Verifikálható formában:
A tapasztalt ellenőrök képesek legyenek az összes rendszerfunkció használatára egy kétórás képzés után. A képzés után a tapasztalt felhasználók által elkövetett hibák száma átlagosan ne haladja meg a napi kettőt."

A **szakterületi követelmények** A szakterületének jellegzetességeiből származnak, nem a rendszer felhasználójának egyéni igényeiből. Itt a legfőbb problémát az jelenti, hogy ezeket a követelményeket az alkalmazás szakterületén használt terminológiával fogalmazzák meg, amihez a szoftvertervezők általában nem értenek. A szakterület szakértői kihagyhatnak információkat a követelményből, mivel az számukra teljesen nyilvánvaló, de a szoftver fejlesztőinek nem.

VII.2. Felhasználói követelmények

A rendszernek csak a külső viselkedését írják le, és kerülni kell benne a rendszer tervezésének jellemzőit. A felhasználói követelményeket természetes nyelven írják, így a következő **problémák** merülnek fel: 1. egyértelműség hiánya; 2. követelmények keveredése; 3. követelmények ötvöződése.

Ha a felhasználói követelmények túl sok információt tartalmaznak, az korlátozza a rendszerfejlesztő szabadságát, hogy újító megoldást adjon, másrészt nehezen érthetővé teszi a leírást. A felhasználói követelményeknek egyszerűen a kulcsfontosságú igényekre kell összpontosítaniuk.

Egy példa:
2.6. Rács eszközök
Az egyedek diagramon történő pozícionálását segítendő a felhasználó bekapcsolhat egy rácsot akár centiméteres, akár hüvelykes beosztással, a vezérlőpanelen lévő opció segítségével. Kezdetben a rács kikapcsolt állapotban van. A rács a szerkesztés menete alatt tetszőlegesen ki- bekapcsolható, és bármikor válthatunk a centiméteres és hüvelykes beosztás között. A rács opció az ablakméretre illeszthető nézetben is elérhető lesz, de a rácsvonalak száma csökkeni fog annak elkerülése végett, hogy a kisebb diagramokat kitöltsék a rácsvonalak.

2.6. Rács eszközök

2.6.1. A szerkesztő biztosítson egy rács eszközt, ahol a vízszintes és a függőleges vonalak mátrixa hátteret nyújt a szerkesztőablakhoz. Ez a rács legyen passzív rács, ahol az elemek igazítása a felhasználóra tartozik.
Magyarázat: Egy rács segíti a felhasználót egy rendezett diagram elkészítésében, az elemek jó elhelyezésében. Bár egy aktív rács, ahol az elemek a rácsvonalakhoz „tapadnak" hasznos lehet, de a pozícionálást pontatlanná teszi. Annak eldöntésében, hogy hova kerüljenek az elemek a felhasználó a legilletékesebb személy.

Az első változat első mondata három követelményt mos össze: fogalmi, funkcionális (a rács); nemfunkcionális (a mértékegységek); nemfunkcionális felhasználói követelményt (a rács ki- és be kapcsolásának a módja). Ezzel ellentétben a második változat a lényegre összpontosít. A követelményekhez fűzött magyarázat fontos, hogy a rendszerfejlesztők és karbantartók megértsék,

miért került be a követelmény, és hogy meg tudják becsülni a követelmények megváltoztatásának hatását.

A felhasználói követelmények leírásakor tartsunk be néhány egyszerű irányelvet:

- Találjunk ki és használjunk egy szabványos formátumot.
- Használjuk következetesen a nyelvet, pl. tegyünk különbséget szükséges és kívánatos között (kell és javasolt).
- Használjunk szövegkiemelést (félkövér és dőlt betűk) a kulcsfontosságú részek hangsúlyozására.
- Kerüljük el a számítógépes zsargon használatát.

VII.3. Rendszerkövetelmények

A rendszerkövetelmények a felhasználói követelmények részletesebb leírásai. Alapjául szolgálnak a rendszer megvalósítási szerződéséhez, így az egész rendszer teljes és ellentmondásmentes meghatározását kell tartalmazniuk. A rendszerkövetelmények specifikációja tartalmazhatja a rendszer különböző modelljeit, pl. objektummodellt, vagy adatfolyam-modellt.
Elvben a rendszerkövetelmények feladata annak leírása, hogy mit kell csinálnia a rendszernek, nem pedig az, hogy hogyan kellene megvalósítani, de a részletességnek ezen a szintjén jóformán lehetetlen kizárni minden tervezési információt.

A természetes nyelvet gyakran használják a rendszerkövetelmények specifikációja megírásához. A fentebb már említett három probléma mellett itt továbbiak merülhetnek fel:

4. A természetes nyelv megértése azon alapszik, hogy az író és az olvasó ugyanazokat a szavakat használja ugyanazokhoz a fogalmakhoz. Ez viszont nincs feltétlenül így, főleg a természetes nyelv többértelműsége miatt.
5. Túl rugalmas. Ugyanazt a dolgot teljesen eltérő formában is elmondhatjuk, az olvasó dolga, hogy kitalálja, a követelmények mikor egyeznek meg és mikor különböznek.
6. A természetes nyelvű követelmények modularizálására nincs könnyű módszer. Lehet, hogy bonyolult az összes kapcsolódó követelményt megtalálni. Ahhoz, hogy felfedezzük egy változtatást következményeit, lehet, hogy a teljes specifikációt át kell néznünk.

A fenti okok miatt a természetes nyelvnek több alternatíváját is kipróbálták:

- strukturált természetes nyelv
- tervleíró nyelvek
- grafikus jelölések (use case-ek)
- matematikai specifikációk (véges állapotú automaták, halmazok)

A strukturált természetes nyelv a természetes nyelv egyfajta leszűkítése a rendszerkövetelmények leírásához. Ennek az az előnye, hogy a természetes nyelv kifejezőképességét és érthetőségét jórészt megtartja, de egységességet is nyújt. Erre egy példa az űrlap alapú megközelítés, ahol egy vagy több szabványos űrlapot kell definiálnunk, és végig ezeket használjuk a követelmények kifejtéséhez. Pl.:

Funkció Csomópont hozzáadása

Leírás Hozzáad egy csomópontot a meglévő tervhez. A felhasználó kiválasztja a csomópont típusát és helyét. Hozzáadás után a csomópont lesz aktuális. A felhasználó úgy választja ki a csomópont helyét, hogy a kurzort oda viszi, ahová a csomópontot helyezni akarja.

Bemenet Csomópont típus, Csomópont pozíció, Terv azonosító

Forrás A Csomópont típus és a Csomópont pozíció a felhasználótól eredő bemenetek, a Terv azonosító pedig az adatbázisból.

Kimenet Terv azonosító

Cél A terv adatbázis. A terv az adatbázisba kerül a művelet befejezésekor.

Igény Tervezési gráf, melynek gyökere a bemeneti azonosító.

Előfeltétel A terv nyílt és megjeleníthető a felhasználó képernyőjén.

Utófeltétel A terv változatlan, eltekintve egy megadott típusú csomópont hozzáadásától egy megadott helyen.

Mellékhatás Nincs.

Tervleíró nyelv: PDL A természetes nyelvű specifikáció többértelműségének kivédésére találták ki a programleíró nyelveket PDL (Program Description Language). A PDL olyan programozási nyelvből származó nyelv, mint a Java vagy az Ada. Tartalmazhat új, absztrakt konstrukciókat a kifejezőerő növelésére. A PDL–ek szoftveres eszközökkel szintaktikailag és szemantikailag is ellenőrizhetők. Két esetben javasolt a használatuk:

- Ha egy művelet egyszerűbb tevékenységek sorozataként definiált és a végrehajtás sorrendje fontos.
- Ha hardver- és szoftverinterfészeket kell megadni.

A PDL használatának hatékony módja, ha összekapcsoljuk a strukturált természetes nyelvvel. A teljes rendszer specifikálásához űrlap alapú megközelítést használunk, a vezérlési sorozatok és az interfészek részletesebb leírásához pedig PDL-t.

Egy ATM működésének PDL-leírásából egy részlet:

```
class ATM
{
 //deklarációk helye
 public static void main (String args[]) throw InvalidCard {
 try
 {
              thisCard.read(); // InvalidCard kivételt dobhat
              pin = KeyPad.readPin(); attempts = 1;
              while (!thisCard.pin.equals(pin) & attempts <4)
              {
           pin = KeyPad.readPin(); attempts = attempts + 1;
         }
         if (!thisCard.pin.equals(pin) throw new InvalidCard (”A kód téves”);
```

```
        thisBalance = thisCard.getBalance();
        do
        {
        Screen.prompt(”Válasszon szolgáltatást”);
        service = Screen.touchKey();
        switch (service)
                {
                 case Services.withdrawalWithReceipt:
                receiptRequired = true;
                 case Services.withdrawalNoReceipt:
                amount = KeyPad.readAmount();
                if (amount > thisBalance)
                        {
                        Screen.printmsg(”Kevés az egyenlege”);
                        break;
                        }
                        Dispenser.deliver (amount);
                        newBalance = thisBalance – amount;
                        if (receiptRequired)
                         Receipt.print(amount, newBalance);
                        break;
                 //egyéb szolgáltatások helye
                default: break;
                }
        }
        while (service != Service.quit);
        thisCard.returnToUser(”Vegye ki a kártyát.”);
   }
   catch (InvalideCard e)
   {
     Screen.printmsg(”A kártya vagy a kód nem érvényes”);
   }
  // egyéb kivételek kezelésének helye
 } //main vége
} //ATM vége
```

VII.4. A szoftverkövetelmények dokumentuma

A rendszerfejlesztőkkel szemben támasztott elvárások hivatalos leírása.
A követelménydokumentum lehetséges használói és hogy mire használják:

Használók	**Mire?**
megrendelők	Meghatározzák a követelményeket és ellenőrzik, hogy azok megfelelnek-e az igényeiknek. Változtatásokat adnak meg a követelményekhez.
menedzserek	Az árajánlat elkészítéséhez és a rendszerfejlesztési folyamat megtervezéséhez használják.

rendszertervezők	Annak megértéséhez használják, hogy milyen rendszert kell fejleszteni.
rendszerteszt-tervezők	Validációs tesztek készítésére.
rendszerkarbantartás-tervezők	Segít megérteni a rendszer és a rendszer részei közötti összefüggéseket.

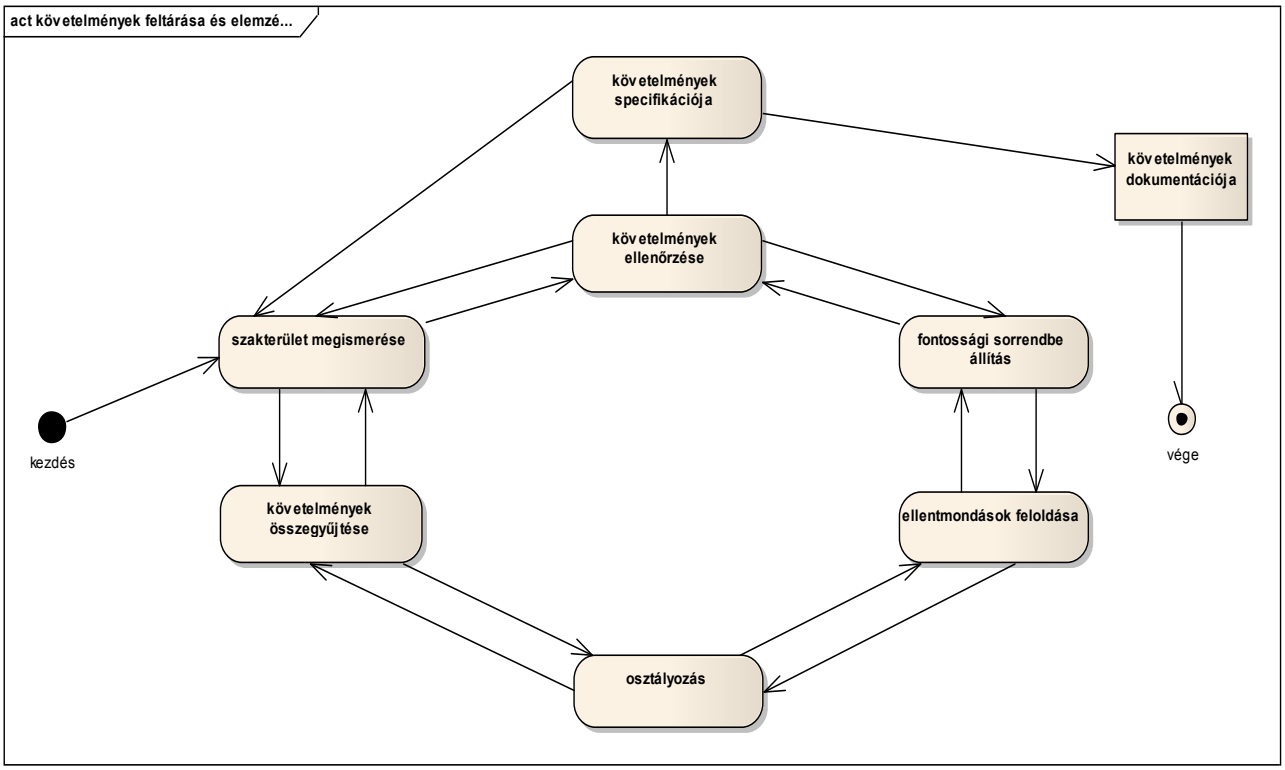
VIII. A követelmények feltárása és elemzése

A követelménytervezés (szoftverspecifikációs folyamat) második nagy tevékenysége (lásd II.1. rész). E tevékenység során együtt kell működni a szoftverfejlesztőknek a megrendelőkkel és a végfelhasználókkal. Itt ki kell választani az úgynevezett **kulcsfigurákat, akiknek közvetett vagy közvetlen befolyása lehet a rendszerkövetelményekre.**

A feltárás és elemzés nehézségeinek okai:

- A kulcsfigurák gyakran nem tudják pontosan, mit várnak el a számítógépes rendszertől. Bonyolult lehet számukra annak kifejezése, mit akarnak a rendszertől; valóságtól elrugaszkodott kívánságaik lehetnek, mivel nincsenek tisztában a költségekkel.
- A rendszer kulcsfigurái a saját szakterületi fogalmaikkal fejtik ki a követelményeket.
- Az egyes kulcsfiguráknak különböző követelményeik vannak, ebben a követelménytervezőknek fel kell fedezniük a közös dolgokat és ellentmondásokat.
- A rendszerkövetelményeket politikai tényezők is befolyásolhatják. Lehetnek olyan vezetők, akik azért igényelnek specifikus rendszerkövetelményeket, hogy ezzel növeljék a szervezeten belüli befolyásukat.
- A gazdasági és az üzleti környezet eközben dinamikusan változik, új kulcsfigurák jelenhetnek meg, újabb követelményekkel.

A feltárási és elemzési folyamat általános modellje:



1. *A szakterület megismerése.* Az elemzőknek fejleszteniük kell az alkalmazás szakterületére vonatkozó ismereteiket.
2. *Követelmények összegyűjtése.* A rendszer kulcsfiguráival való együttműködés. Eközben javul a szakterület megértése.
3. *Osztályozás.* A követelmények strukturálatlan gyűjteményét összefüggő csoportokba szervezi.
4. *Ellentmondások feloldása.* Ahol több kulcsfigura is érintett, elkerülhetetlen, hogy a követelmények ellentmondása ne kerüljenek.
5. *Fontossági sorrend felállítása.* A kulcsfigurákkal együttműködve, kiválasztjuk a legfontosabb követelményeket.
6. *Követelményellenőrzés.* Teljes-e, ellentmondásmentes-e és összhangban van-e azzal, amit a kulcsfigurák a rendszertől valójában várnak.

A továbbiakban a követelmények feltárására és elemzésére három technikát nézünk meg.

VIII.1. Nézőpont-orientált feltárás

A közepes vagy nagy rendszerek esetén a végfelhasználók általában különböző típusokba sorolhatók. Ismét a banki pénzkiadó rendszert (ATM) véve példának:

- *a bank jelenlegi ügyfelei:* akik számára a rendszer szolgáltatást nyújt
- *más bankok képviselői:* akiknek kölcsönös megállapodásuk van egymás ATM-jeinek használatáról
- *a bankfiókok vezetői:* akik vezetői információkat nyernek a rendszerből
- *a bankfiók alkalmazottai:* akiknek feladata a rendszer napi működtetése, az ügyfelek panaszainak kezelése
- *a bank marketing osztálya*: amely a rendszert marketingcélokra használja
- *hardver- és szoftverkarbantartó mérnökök:* akik a hardver- és szoftverelemek

karbantartásáért és fejlesztéséért felelnek
- stb.

Ebből is látszik, hogy még egy viszonylag egyszerű rendszer esetén is sok különböző nézőpontra kell tekintettel lennünk. Ezek a perspektívák általában átfedik egymást.

A nézőpont-orientált szemlélet legfőbb erőssége az, hogy felismeri a többféle perspektíva létezését, és eszközöket ad a különböző kulcsfigurák által javasolt követelmények ellentmondásainak felderítésére. A különböző módszerek mást és mást értenek a „nézőpont" szó alatt. Nézőpont lehet:
- adatforrás vagy adatnyelő
- reprezentációs eszközkészlet
- a szolgáltatások fogadója

Az interaktív rendszereknél a 3. szemléletmód a legcélravezetőbb. Ebből a csoportból egy konkrét módszer a VORD (Viewpoint-Oriented Requirements Definition). A VORD-ben a nézőpont- és szolgáltatás információk szabványos űrlapok segítségével kerülnek összegyűjtésre. Emellett használ még nézőpont-hierarchia diagramokat és esemény forgatókönyveket is.

Pl.: ATM Első lépés a lehetséges nézőpontok azonosítása. Ez ötletgyűjtéssel történhet, amikor a kulcsfigurák összejönnek és lehetséges nézőpontokat javasolnak. Következő lépés a nézőpontok és a szolgáltatások azonosítása. A szolgáltatásokat ajánlott a nézőpontokhoz elhelyezni. Egy szolgáltatás több nézőponthoz is tartozhat. A szolgáltatások felsorolása mellett a nézőpontok bemenetet is biztosítanak ezekhez a szolgáltatásokhoz. Pl.: Az ATM használóknak pénzfelvételkor meg kell adniuk a kívánt pénzösszeget. A nézőpontok vezérlő információ is nyújtanak annak megállapítására, hogy a szolgáltatások felkínálásra kerülnek-e és ha igen, mikor. A vezérlő információt az automatán lévő gombok biztosítják, az adatokat pedig a felhasználó kártyája és a billentyűzet. Ezután elkészíthetjük a nézőpont-hierarchia diagramot.

A következő lépésben kitöltjük a szolgáltatás- és nézőpont-sablonokat. Végül elkészítjük az esemény-forgatókönyveket, és keresztellenőrzést végzünk a hibák és az ellentmondások felderítésére. A VORD-módszerhez CASE-eszközt érdemes használni.

VIII.2. Forgatókönyvek

Az emberek általában könnyebben kezelik a valós életbeli problémákat, mint az absztrakt leírásokat. **A forgatókönyvek interakciósorozatok leírásai.** A következő részekből állnak:
- a rendszer kezdeti állapotának leírása
- a forgatókönyvbeli események normális menetének leírása
- egy leírás arról. hogy mi romolhat el és ezt hogyan kezeli a rendszer
- egyéb tevékenységek leírása, amelyek ugyanabban az időben mehetnek végbe
- a rendszer végállapotát a forgatókönyv befejezésekor

A forgatókönyveknek két fő típusa van: esemény-forgatókönyvek és a használati esetek.

A VORD módszer esemény forgatókönyvet használ.

A használati esetek (use-case-ek) az UML jelölésrendszer részei. A use-case diagramokat szekvenciadiagrammal szokás kiegészíteni, ezek mutatják a további információkat.

VIII.3. Etnográfia

Megfigyelésen alapuló technika, amely felhasználható a társadalmi és szervezeti követelmények megértéséhez. Az elemző elmélyed abban a munkakörnyezetben, ahol a rendszert majd használni fogják, megfigyeli a napi munkát és jegyzeteket készít.

Az emberek gyakran nehéznek találják kifejezni munkájuk részleteit. A saját munkájukat megértik, de azt valószínűleg nem, hogy az milyen összefüggésben áll a szervezet többi részével. Azok a társadalmi és szervezeti tényezők, amik a munkát érintik, de az egyének számára nem nyilvánvalóak, csak akkor válhatnak világossá, ha egy tárgyilagos külső szemlélő észreveszi őket.

Az etnográfia különösen hatékony a követelmények két típusának felderítésénél:

- Azoknál a követelményeknél, amelyek az emberek tényleges munkavégzési módjából erednek, nem pedig abból, ahogyan a folyamatdefiníciók szerint kellene dolgozniuk. Pl.: a légiforgalom-irányítók kikapcsolhatják azt az összeütközési riasztórendszert, amely a légi járművek egymást keresztező útvonalait érzékeli, annak ellenére, hogy a használatát a szabványok előírják. Ha úgy tervezik meg a pályákat, hogy a repülők eltávolodnak egymástól mielőtt még ez problémát okozna, akkor a riasztórendszer csak akadályozná őket a munkában.
- Azoknál a követelményeknél, amelyek az együttműködésből és a más emberek tevékenységének számon tartásából erednek. Pl.: a légiforgalom-irányítók felhasználhatják a többi irányító adatait annak megjósolására, hogy hány repülő lép majd be az ő szektorukba. Ezért egy önműködő légi irányítási rendszernek ajánlott lehetővé tenni a szektorbeli irányítók számára, hogy valamelyest belelássanak a szomszédos szektorok munkájába.

Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: : Ian Somerville: Szoftverrendszerek fejlesztése. Második, bővített, átdolgozott kiadás, Panem Kiadó, Budapest 2007.*)*

IX. Szoftverprototípus készítése

A prototípus a szoftverrendszer kezdeti verziója, amelyet arra használnak, hogy bemutassák a koncepciókat, kipróbálják a tervezési opciókat, és hogy jobban megismerjék a problémát és annak lehetséges megoldásait.
A szoftverprototípus két tevékenységet támogat a követelménytervezési (szoftverspecifikációs) folyamatban: követelmények feltárását és a követelmények validálását.
A rendszerprototípus fejlesztésének előnyei:

1. A funkciók bemutatásakor azonosítani lehet a szoftverfejlesztők és a felhasználók közötti félreértéseket.
2. A szoftverfejlesztésen dolgozók hiányos és/vagy ellentmondásos követelményekre akadhatnak.
3. Hamar a rendelkezésünkre áll egy működő rendszer, így demonstrálhatjuk a vezetőségnek az alkalmazás megvalósíthatóságát és hasznosságát.
4. A prototípus felhasználható a valódi rendszer specifikációjának megírásakor.
5. A rendszer jobban használható lesz.
6. A rendszer jobban illeszkedik a felhasználói igényekhez.
7. Jobb a tervezés minősége.
8. Jobb a rendszer karbantarthatósága.
9. Kevesebb erőfeszítés szükséges a fejlesztéshez.

A prototípus felhasználható még:

1. Felhasználók képzésére.
2. Rendszer tesztelésére. „Megerősítő tesztek” - a prototípus és a kész rendszer ugyanazt a tesztesetet futtatja. Ha mindkettő ugyanazt az eredményt adja, akkor jó, ha nem, a különbség okait meg kell vizsgálni.

A prototípuskészítés rendszerint a szoftverfolyamat korai szakaszában növeli a költségeket, később viszont jelentősen csökkenti.
A prototípuskészítés céljait érdemes az elején írásban megadni, mert e nélkül a vezetés vagy a végfelhasználók félreérthetik a rendeltetését. Ilyen cél lehet: az alkalmazás megvalósíthatóságának demonstrálása vagy a felhasználói felületek bemutatása, stb. Ugyanaz a prototípus nem szolgálhatja az összes célt.
A folyamat következő szakasza annak eldöntése, hogy mit tegyünk bele a prototípusba. Az utolsó szakasz a kiértékelés. Ennek folyamán gondoskodni kell a felhasználók képzéséről, mivel időbe telik, amíg megszokják az új rendszert és csak ezután fedezhetik fel a követelménybeli hibákat és hiányosságokat.

IX.1. Prototípus fajták és gyors prototípuskészítési technikák

Két fő típusa van a prototípusoknak: az evolúciós és az eldobható.

Az **evolúciós prototípus** készítésének célja egy működő rendszer átadása a végfelhasználóknak. Ezért a legjobban megértett és leginkább előtérbe helyezett követelményekkel javallott kezdeni. A kevésbé fontos és körvonalazatlanabb követelmények akkor kerülnek megvalósításra, amikor a felhasználók kérik. Ez a módszer a weblapfejlesztés és az e-kereskedelmi alkalmazások szokásos technikája.

Az **eldobható prototípus** készítésének célja a rendszerkövetelmények validálása vagy származtatása. A nem jól megértett követelményekkel érdemes kezdeni, mivel azokról szeretnénk többet megtudni.

A **gyors prototípuskészítési technikák** olyan fejlesztési technikák, amik az átadás gyorsaságára helyezik a hangsúlyt, nem a teljesítményre, a karbantarthatóságra vagy a megbízhatóságra. Három ilyen technika létezik:

1. fejlesztés dinamikus magas szintű nyelven
2. adatbázis-programozás
3. komponensek és alkalmazások összeépítése

A dinamikus magas szintű nyelvek olyan programozási nyelvek, amik erőteljes futási idejű adatkezelő eszközöket tartalmaznak. A nyelv olyan eszközöket tartalmaz, amiket a C-hez hasonló nyelveken több primitív konstrukcióból kellene felépíteni. Ilyen nyelvekre példa: a Lisp (lista struktúrán alapul), a Prolog (a logikán alapul) és a SmallTalk (objektumokon alapul).
Nem túl rég a nagyon magas szintű nyelveket nem használták széles körben a nagy rendszerek fejlesztésénél, mert nagy teljesítményű futtató rendszert igényelnek. Az ilyen nyelven írt programoknak nagy a tárigénye és lassabbak a hagyományos nyelven írtaknál. Viszont a növekvő teljesítmény és a hardverek csökkenő költsége ezeket a szempontokat kezdi háttérbe szorítani.

A Java napjaink alapvető fejlesztési nyelve. A gyökerei a C++-ban vannak, és magában foglalja a SmallTalk számos jellemzőjét, pl. a platformfüggetlenséget és az automatikus tárkezelést. A Java a nagyon magas szintű nyelvek számos előnyét nyújtja, a hagyományos harmadik generációs nyelvek kínálta szigorúság és teljesítményoptimalizálási lehetőségek mellett. Sok újrafelhasználható komponens érhető el Java-ban, így az igen alkalmaz evolúciós prototípus készítésére.

A SmallTalk-ot és a Java-t leginkább interaktív rendszerek, a Prolog-ot és a Lisp-et szimbolikus feldolgozást végző rendszerek prototípusához használják. A prototípuskészítéshez használt nyelv kiválasztásánál a következő kérdéseket érdemes feltenni:

1. Mi a probléma alkalmazási szakterülete? Pl.: természetes nyelvű feldolgozáshoz – Prolog és Lisp
2. Milyen felhasználói interakciókra van szükség? webböngészőhöz integrálható – Java és a Smalltalk; szöveges – Prolog
3. Milyen támogatási környezetet kapunk a nyelvvel?

Ezeket a nyelveket kombinálva is használhatjuk a prototípuskészítéshez.

Adatbázis-programozás

Az üzleti alkalmazások többsége adatbázisból származó adatokat manipulál és az adatok átszervezésével és formázásával állítja elő a kimenetet. Az ilyen alkalmazások fejlesztésének a támogatására ma minden kereskedelmi adatbázis-kezelő rendszer lehetővé teszi az adatbázis-programozást. Az adatbázis-programozási nyelvet és annak támogatási környezetét együtt **negyedik generációs nyelv**nek (4GL) nevezik. A legtöbb 4GL támogatja a webes adatbázis-kezelést.
A negyedik generációs nyelvek azért sikeresek, mert speciális interaktív alkalmazások előállítására optimalizálták őket. Ezek az alkalmazások egy szervezeti adatbázisból információt nyernek ki, ezt bemutatják a felhasználók terminálján és a felhasználók által végzett változtatásokkal frissítik az adatbázist.
Egy 4GL környezet a következő eszközökből áll:

- Egy adatbázis-lekérdező nyelv, ami általában az SQL.

- Egy interfészgenerátor, amit az adatbevitelre és megjelenítésre szolgáló űrlapok létrehozására használnak.
- Egy táblázatkezelő a numerikus információ elemzéséhez és módosításához.
- Egy jelentésgenerátor, melyet az adatbázisból kinyert információk bemutatására szolgáló jelentések létrehozására használnak.

A 4GL-ek igen alkalmasak prototípuskészítéshez, de a kész termék velük való előállításakor vannak hátrányok is. Általában lassabbak és sokkal több memóriát fogyasztanak. Egy kísérletben egy 4GL nyelvű program C++-ba való átírása 50%-kal csökkentette a memóriahasználatot. A C-be való átírás után pedig 10-szer gyorsabban futott, mint az eredeti.

Komponensek és alkalmazások összeépítése

Gyorsan létre tudunk hozni prototípusokat, ha rendelkezésünkre állnak újrafelhasználható komponensek. Az összeépítési mechanizmusnak tartalmaznia kell vezérlő eszközöket és valamilyen megoldást a komponensek kommunikációjára.

Az újrafelhasználáson alapuló prototípusfejlesztést két szinten lehet támogatni:

- **Alkalmazási szinten,** ahol teljes alkalmazásrendszereket integrálnak a prototípussal azért, hogy azok funkcióit meg lehessen osztani. Pl.: ha a prototípusban szövegszerkesztésre van szükség, egy teljes szövegszerkesztő programot is integrálunk.
- **Komponens szinten**, ahol a rendszer implementálásához az egyedi komponenseket egy szabványos keretrendszeren belül integrálják. A komponens független, végrehajtható entitás. Forráskódja nem hozzáférhető, így nem a rendszer többi részével együtt fordítjuk. A komponensek közzéteszik interfészeiket, amiken keresztül elérhetjük a műveleteiket. Egy komponensnek lehet biztosított és elvárt interfésze. Az elvárt interfész a komponens által a rendszertől várt szolgáltatásokat definiálja. A komponensek integrálásának szabványos keretrendszere lehet egy evolúciós fejlesztéshez tervezett scriptnyelv, mint a Visual Basic, a TCL/TK, Python vagy a Perl. A scriptnyelvek típus nélküli magas szintű nyelvek, melyeket arra terveztek, hogy komponensek integrálásával rendszereket hozzanak velük létre. Másik lehetőségként ez a keretrendszer lehet egy általánosabb komponensintegrálási keretrendszer, ami a CORBA-n, DCOM-on vagy JavaBean-en alapul.

TERVEZÉS

X. Objektum-orientált tervezés

Olyan tervezési stratégia, amelyben a rendszertervezők műveletek és funkciók helyett „dolgokban” gondolkodnak. Az objektumok saját állapotukat karbantartó és erről információs műveleteket biztosító egységek, amik egymással együttműködnek. Az objektumok elrejtik az állapotuk reprezentációját, korlátozzák a kívülről történő hozzáférést. Az objektumok potenciálisan újrafelhasználható komponensek, de sokszor érdemesebb nagyobb egységet választani az újrafelhasználáshoz. Egy objektumorientált tervezési folyamat az osztályoknak és azok közötti kapcsolatoknak a megtervezéséből áll.

X.1. Vezérlés, ütemezés és az objektumok élettartama

Egy objektum „szolgáltatáskérés” üzenetet küldhet egy másiknak, akitől a szolgáltatást kéri. A **soros végrehajtás**, amikor az első objektum megvárja a kért szolgáltatás befejeződését nem követelmény. Ezért az objektumok közötti kölcsönhatás általános modellje lehetővé teszi az

objektumok számára, hogy azok konkurens módon, párhuzamos folyamatokként legyenek végrehajtva. A gyakorlatban a legtöbb objektum-orientált nyelvben a soros végrehajtási modell az alapértelmezés (**szinkron kommunikáció**), ahol az objektumok szolgáltatási kérelmei ugyanúgy vannak implementálva, mint a függvényhívások.

Az újabb OOP nyelvekben azonban, mint pl. a JAVA-ban vagy a C#-ban, léteznek a szálak, amelyek megengedik a konkurens módon végrehajtódó objektumok létrehozását és az **aszinkron kommunikációt** (a hívóobjektum folytatja a működését az általa igényelt szolgáltatás futása alatt is).

A konkurens objektumok kétféleképpen implementálhatók:

1. *Aktív objektumok:* önmaguk képesek belső állapotukat megváltoztatni és üzenetet küldeni, anélkül, hogy más objektumtól vezérlőüzenetet kaptak volna. (Ellentétük a passzív objektum.) Az aktív objektumot reprezentáló folyamat ezeket a műveleteket folyamatosan végrehajtja, így soha nem függesztődik fel.
2. *Szerverek:* az objektum a megadott műveleteknek megfelelő eljárásokkal rendelkező párhuzamos folyamat. Az eljárások egy külső üzenetre válaszolva indulnak el és más objektumok eljárásaival párhuzamosan futhatnak. Mikor befejezték a tevékenységüket, az objektum várakozó állapotba kerül és további kéréseket vár.

A szervereket leginkább osztott környezetben érdemes használni, ahol a hívó és a hívott objektum különböző számítógépeken hajtódik végre. Az igényelt szolgáltatásra adott válaszidő megjósolhatatlan, ezért úgy kell megtervezni a rendszert, hogy a szolgáltatást igénylő objektumnak ne kelljen megvárni a szolgáltatás befejeződését. A szerverek persze egyedi gépen is használhatók, ahol a szolgáltatás befejeződéséhez némi időre van szükség, pl. nyomtatás és a szolgáltatást több különböző objektum is igényelheti.

Aktív objektumokat akkor célszerű használni, ha egy objektumnak saját állapotát megadott időközönként frissíteni kell. Ez valós idejű rendszerekben gyakori, ahol az objektumok a rendszer környezetéről információt gyűjtő hardvereszközökkel állnak kapcsolatban.

Ütemezés

Amennyiben a programot kevesebb processzoron futtatjuk, mint a vezérlési szálak száma, akkor a processzorok idejét meg kell osztani az egyes szálak között. Ehhez szükség van egy **ütemezőre.** Az ütemező lehet programon kívüli eszköz, pl. az operációs rendszer része, vagy a programon belüli ütemező objektum. Az első megoldás a párhuzamos task-ok közötti kommunikációhoz, a második pedig az ütemezés belső megvalósításához igényel új objektumokat.

Az ütemezés kialakításakor két stratégia közül választhatunk:

- **Nem preemptív**: az egyes szálakat indító aktív objektumokra bízzuk, hogy lemondjanak a processzor használatáról
- **Preemptív**: az ütemező az engedélyezett időszelet lejártakor erővel elveheti a processzort az aktív objektumtól.

A két stratégia eltérő objektumokat tételez fel és a közös erőforrások megosztását is más módon kezeli. A nem preemptív ütemezőknél az erőforrások egymást kölcsönösen kizáró használatát a processzor használatáról való lemondás időtartamának a megfelelő megválasztásával érhetjük el. A preemptív ütemezőknél pedig szemaforokkal védjük az erőforrásokat.

Objektumok élettartama

A megvalósítandó objektum belső állapotát az attribútumainak pillanatnyi értéke határozza meg. Ezeket az adatokat az objektum élete során tárolni kell. A háttértárolón azon objektumok adatait kell tárolni, amelyek élettartama hosszabb, mint a program futási ideje. Ezeket **perzisztens**

objektumoknak nevezzük. (Azokat az objektumokat pedig, amelyek élettartama a program futási idejénél nem hosszabb, **tranziens**-nek nevezzük.) Ha a program nagyszámú perzisztens objektummal dolgozik, érdemes egy adatbázis-kezelő rendszerrel kiegészíteni. A relációs adatbázisokat nagyszámú, de viszonylag kevés osztályhoz tartozó objektum tárolására érdemes igénybe venni, egyébként érdemesebb objektum-orientált adatbázis-kezelőt használni.

Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: Ian Somerville: Szoftverrendszerek fejlesztése. Második, bővített, átdolgozott kiadás, Panem Kiadó, Budapest 2007.*)*

XI. Valós idejű (real-time) szoftverek tervezése

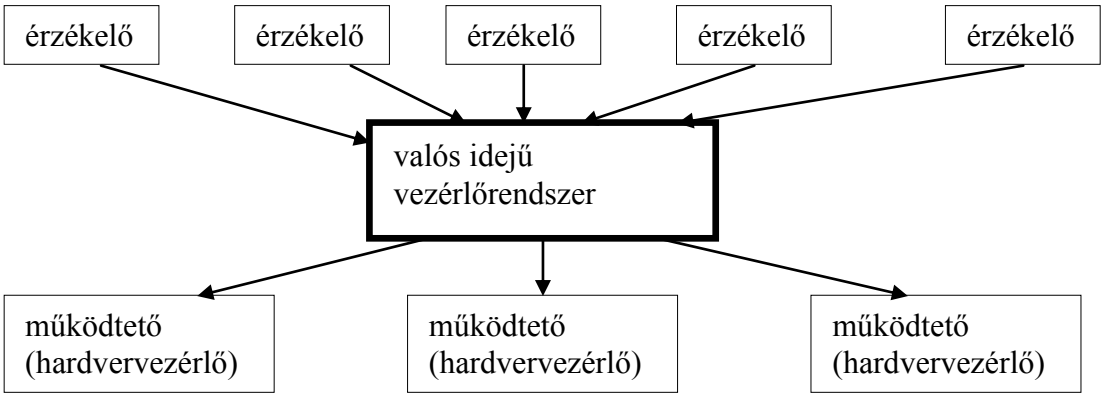
XI.1. Alapfogalmak

A valós idejű rendszer definíciója: **a rendszer helyes működése egyrészt a rendszer által adott eredményektől, másrészt attól az időtől függ, amennyi idő alatt ez az eredmény előáll.** Egy „gyengén" valós idejű rendszer (soft real-time) műveletei korlátozottak, ha a meghatározott időn belül az eredmények nem jönnek létre. Egy „erősen" valós idejű rendszer (hard real-time) műveletei érvénytelenek, ha az eredmények nem jönnek létre a megadott időn belül. (Vagyis a specifikáció rendszerhibának tekinti az időkövetelmények be nem tartását.)

Valósidejű egy rendszer, ha a specifikációjában valamilyen időbeli viselkedésre vonatkozó előírás szerepel a külső, valós időhöz kötötten.

A valós idejű rendszerek a gyakorlatban egyben beágyazott rendszerek is. **Beágyazott rendszer:** a célfeladatot (vezérlést) ellátó számítástechnikai rendszer beépül egy nem számítástechnikai rendszerbe. Pl.: automata mosógép, ipari gyártósor, mobiltelefon, stb. Emiatt a valós idejű és a beágyazott megnevezést a továbbiakban szinonimaként fogom használni.

A valós idejű rendszer általános modellje:



A fenti felépítés miatt hívják őket **inger/válasz rendszereknek is.** Kaphat periodikus (rendszeresen ismétlődő) és aperiodikus (rendszertelen) ingereket. A periodikus ingereket általában az érzékelők váltják ki, a rendszertelen ingereket pedig akár az érzékelő, akár a működtető kiválthatja. Ez utóbbiak gyakran rendkívüli eseményt jelölnek.

Egy valós idejű rendszernek tudnia kell válaszolnia a különböző gyakorisággal és különböző időpontokban beérkező ingerekre. Ezért a szerkezetét úgy kell megadni, hogy a vezérlést az inger beérkezésétől számítva a lehető legrövidebb idő alatt át kell adni a kezelőnek. Ez soros végrehajtású (szekvenciális) programokban nem oldható meg, ezért **konkurens, együttműködő folyamatok** zajlanak a valós idejű rendszerekben.

A valós idejű rendszerek két fő típusa:

- **figyelő- és vezérlőrendszerek**: időszakosan lekérdezik a rendszer környezetéről információt szerző érzékelőket. Az érzékelőkről elolvasott adatok alapján a működtetőknek utasításokat adnak. (pl. épület behatolás elleni riasztórendszere)

- **adatgyűjtő rendszerek**: további feldolgozás és elemzés céljából gyűjtenek adatokat az érzékelőktől. Általában a termelő-fogyasztó modellnek megfelelően szervezik őket. A termelőfolyamat egy körkörös pufferbe helyezi az adatokat, ahonnan a fogyasztófolyamat kiveszi és felhasználja őket. (pl. nukleáris reaktor neutronfluxusát figyelő rendszer)

XI.2. Rendszertervezés

A rendszertervezés folyamatának része annak eldöntése is, hogy a rendszer mely képességeit valósítsuk meg szoftveresen és melyeket hardveresen. A hardverelemmel jobb teljesítményt lehet elérni, a szoftver pedig rugalmasabb. A tervezés során a hardver/szoftver elosztásról való döntést a lehető legkésőbb kell meghozni. Egy jó szoftvertervezési folyamat olyan rendszert eredményez, amit akár hardveresen, akár szoftveresen meg lehet valósítani.

A valós idejű rendszerek tervezésének lépései:

- Meg kell határozni a rendszer által feldolgozandó ingereket és az azokra adott válaszokat.
- Minden inger-válasz párhoz meg kell határozni az időzítési megszorításokat.
- Választani kell a rendszer számára egy végrehajtási platformot, vagyis hardvert és egy valós idejű operációs rendszert (futtatórendszert) kell választani.
- Az inger és a válasz feldolgozását konkurens folyamatokba kell gyűjteni. Az ingerek és válaszok minden osztályához egy folyamatot rendelünk.
- Minden ingerre és válaszra meg kell tervezni a szükséges számításokat végző algoritmusokat. Ezt gyakran a tervezés korai szakaszában kell megtenni, hogy kiderüljön a feldolgozási műveletek időigénye.
- Ki kell alakítani az ütemezési rendszert, amely biztosítja, hogy a folyamatok időben elindulnak és határidőre befejeződnek.
- A rendszert integrálni kell egy valós idejű futtatórendszer vezérlése alatt.

Egy valós idejű rendszer időzítésének elemzése bonyolult. Erre két fő megközelítés létezik:

- **legrosszabb eset (worst case):** Itt a legkedvezőtlenebb végrehajtási időt determinisztikus modell alapján mindig pontosan ki kell számolni.
- **valószínűségi modell:** A végrehajtási időket a valószínűségi modell alapján tervezzük, vagyis a határidő-mulasztást ugyanolyan kockázati tényezőnek tekintjük, mint a hibák bekövetkezését.

Valós idejű rendszerek modellezésére az UML állapotautomata és időzítés diagramjait használják, illetve pl. a Rhapsody (ILogix) programban, ún. Live Sequence Chart-okat (életvonal-diagramokat), amik az állapotautomata diagram kiterjesztésének tekinthetők.

A rendszer megvalósításához használt **programozási nyelv** is befolyásolhatja a tervet.

- Az erősen valós idejű rendszereket néha még ma is **assemblyben vagy C-ben** írják. Ezekben hatékony programot lehet fejleszteni, de nincsenek nyelvi elemek a párhuzamosság és a megosztott erőforrások kezelésére.
- Az **Adát** eredetileg beágyazott rendszerek megvalósítására tervezték, így olyan tulajdonságokkal rendelkezik, mint a párhuzamos programozás taszkok segítségével, a randevú mechanizmus a taszkok szinkronizációjára, a kivételkezelés és a reprezentációs előírás. Az 1983-as verzió még nem volt alkalmas az erősen valós idejű rendszerek megvalósítására, mivel nem lehetett a taszkokhoz határidőket megadni és nem volt beépített kivétel arra az esetre, ha a határidő nem teljesül. Ezért 1998-ban átdolgozták a nyelvet, és így olyan védett típusokat biztosítottak, amelyek megkönnyítették a védett, megosztott adatszerkezetek implementációját és javították a taszkok ütemezését és időzítését. Ezzel az Ada valós idejű programozási nyelv

lett, de még mindig nem biztosít elegendő vezérlési lehetőséget az erősen valós idejű rendszerekhez.

- A **Java** kezdeti változatait kisebb beágyazott rendszerek vezérlőinek írására tervezték. Ezért a nyelv konkurens objektumok (szálak) és szinkronizációs módszerek formájában támogatja a konkurens folyamatok megvalósítását. Azonban a Java nem rendelkezik szigorú időkorlátokkal és nincs lehetőség a szálak ütemezésére sem. A Javával mint valós idejű programozási nyelvvel a következő problémák merültek fel:
  - Nem lehet megadni, hogy a szál mikor kezdjen el futni.
  - A szemétgyűjtés mechanizmusa teljesen ellenőrizhetetlen, ami megjósolhatatlanná teszi a szálak időbeli viselkedését.
  - Nincs lehetőség a megosztott erőforrásokhoz rendelt sorok méretének meghatározására.
  - A JVM implementációja gépről-gépre változik, így ugyanaz a program különböző időbeli viselkedéssel is rendelkezhet.
  - A nyelv nem teszi lehetővé a részletes futási idejű tár- és processzorelemzést.

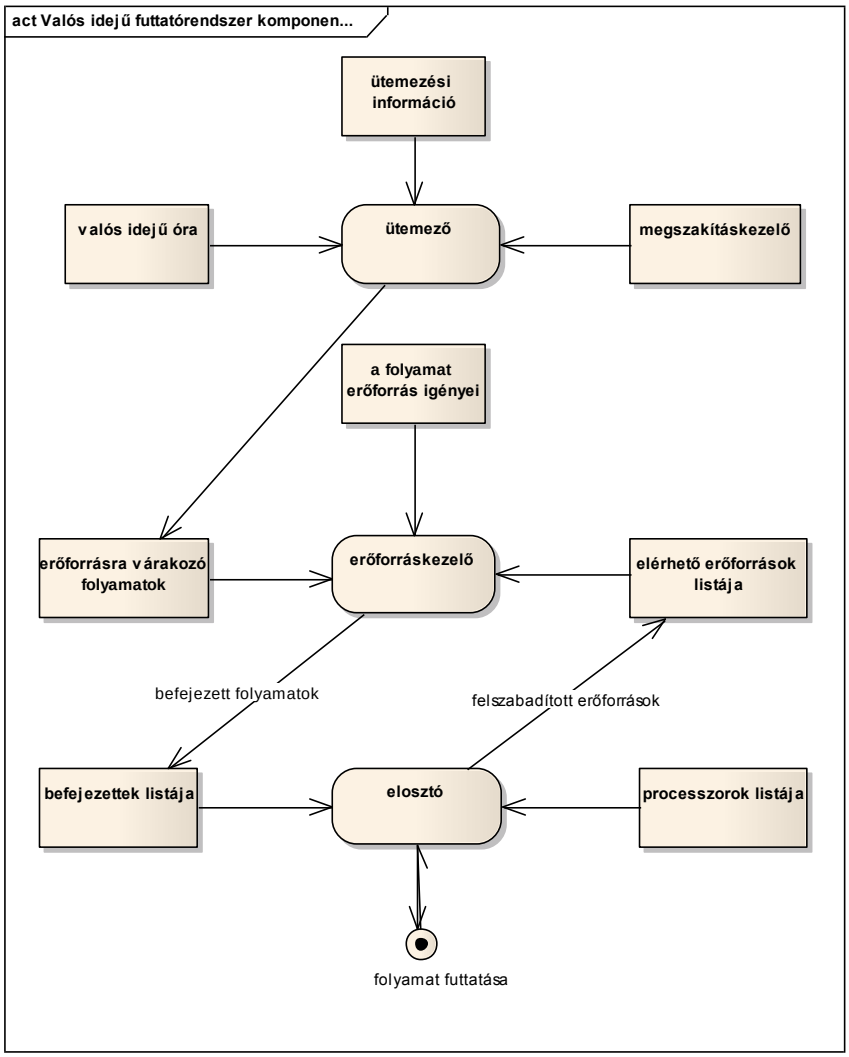
Ezen gondok megoldására dolgoznak egy valós idejű Java-változaton (pl. Sun J2ME, Java 2 Micro Edition). A nyelv hordozhatósága és valós idejűsége mindenképpen ellentétes tulajdonságok.

XI.3. Valós idejű futtatórendszerek

Egy valós idejű futtatórendszer analóg az általános célú számítógépek operációs rendszer fogalmával. Valós idejű futtatórendszerek (operációs rendszerek) például: QNX, RTLinux, RTAI, stb.)

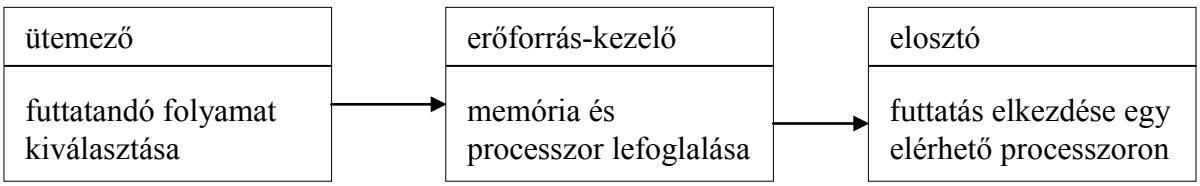
Egy valós idejű rendszerben ő kezeli a folyamatokat és végzi az erőforrás-kiosztást. Általában nem tartalmaz olyan bonyolult operációs-rendszer funkciókat, mint az állománykezelés. **A futtatórendszer komponensei: valós idejű óra; megszakításkezelő; ütemező; erőforráskezelő; elosztó.**

A folyamatos szolgáltatást nyújtó rendszerek futtatórendszerének két további komponenssel is kell rendelkeznie: **konfigurációkezelővel** - a rendszer hardverének dinamikus újrakonfigurálásáért; **hibakezelővel** - a hardver és szoftverhibák felderítésére és elhárításukra szolgál.



A valós idejű rendszerek által feldolgozott ingerek általában különböző **prioritással** (fontossággal) bírnak. A futtatórendszernek legalább két prioritási szintet kell tudnia kezelnie:
megszakítás szintű – a legmagasabb prioritási szint, pl. a valós idejű óráé.
órajel szintű – ilyen prioritást kapnak a periodikus folyamatok. Újabb prioritási szintet adhatunk a háttérben futó folyamatoknak, amelyeknek nincs valós idejű határideje.

A valós idejű futtató rendszer folyamatkezelése: a periodikus folyamatokat előre megadott időközönként végre kell hajtani, hogy adatgyűjtést végezzenek és vezéreljék a működtetőt. A periodikus folyamatkezelés esetén a futtatórendszer műveletei:



Ha a futtatórendszer megszakítást érzékel, akkor az valamilyen szolgáltatásra vonatkozó igényt jelent. A megszakítási mechanizmus a vezérlést egy előre megadott memóriacímre adja, amely egy

rutinra ugró utasítást tartalmaz. A megszakítást kiszolgáló rutinnak egyszerűnek, gyorsnak, rövidnek kell lennie. A megszakítás futása alatt a többi megszakítás le van tiltva, ezért ezt az időt minimalizálni kell.

FEJLESZTÉS

XII. Gyors szoftverfejlesztés

XII.1. Iteratív, inkrementális szoftverfejlesztés

A szoftverrendszereknél ma a legkritikusabb követelmény a gyors fejlesztés és üzembe helyezés. (Sok vállalat hajlandó engedni a minőségből, és kompromisszumot köt a gyors üzembe helyezés érdekében.) Ezek a vállalatok folyamatosan változó környezetben tevékenykednek, ezért a szoftverrel szemben támasztott elvárásaik elkerülhetetlenül változnak. A valós elvárások csak akkor tisztázódnak, amikor a rendszert már átadták és a felhasználók tapasztalatot szereztek a használata során.
A hagyományos szoftverfejlesztési folyamatok túl lassúak a gyorsan változó üzleti környezethez. **A gyors szoftverfejlesztési folyamatokat arra tervezték, hogy segítségükkel gyorsan készíthessük használható szoftvereket.** Ez általában iteratív folyamat, ahol **a specifikáció, a tervezés, a fejlesztés és a tesztelés átfedi egymást.** Ilyenkor a szoftvert nem egy teljes egységként fejlesztik, hanem lépésenként, és minden lépés magában foglalja a rendszer egy további funkcionalitását.
Több gyors szoftverfejlesztési technika is létezik, de van 3 közös vonásuk:

- A specifikáció, a tervezés és az implementálás párhuzamosan zajlik. Nincs részletes rendszerspecifikáció, a felhasználói elvárások leírása csak a rendszer legfontosabb jellemzőire terjed ki.
- A rendszert lépésről lépésre fejlesztik (inkrementális). A végfelhasználók is részt vesznek minden lépés specifikációjában és az eredmény kiértékelésében. Indítványozhatnak változtatásokat és új követelményeket.
- A rendszer felhasználói felülete gyakran egy beépített fejlesztői környezettel készül el, amiben vizuálisan tervezhetjük meg a kinézetét és aztán automatikusan legenerálja webböngészők számára vagy pl. Windows alá.

Az inkrementális (lépésenkénti) elkészítés és átadás **előnyei:**

- Az ügyfélszolgáltatások hamar használhatók, a fejlesztés korai szakaszában már hasznot lehet belőle húzni.
- Mivel a felhasználót bevonják a fejlesztésbe, a rendszer sokkal jobban meg fog felelni az igényeiknek és ráadásul jobban elkötelezik magukat a szoftver mellett. Az inkrementális szoftverfejlesztés közel áll az emberek probléma-megoldási módjához: mivel ritkán dolgozunk ki teljes megoldásokat, hanem a megoldásainkat lépésenként továbbfejlesztjük, és visszalépünk, ha hibáztunk.

Nyilván ennek a megközelítésnek az alkalmazása is okozhat **nehézséget:**

- kezelési problémák, mert kevés a rendszerdokumentáció
- szerződésbeli problémák, mert nincs rendszer-specifikáció, amin a hagyományos szerződés alapul. A fejlesztők idő alapon szeretnének szerződni, a megrendelők pedig fix összegre.
- validációs problémák, a független validáció nehéz (validáció – A megfelelő terméket készítettük-e el?)

- karbantartási problémák, mert a folyamatos változtatás elronthatja a rendszer struktúráját

XII.2. Agilis módszerek, extrém programozás (XP)

A nagy, hosszú élettartamú és kritikus rendszereknél elengedhetetlen a hagyományos, formális fejlesztés, ahol sok idő kell a tervezésre, formalizálásra, dokumentálásra.
A kis- és középvállalkozások rendszereinél és a PC-re fejlesztett programoknál azonban érdemesebb magára a szoftverre koncentrálni és nem a tervezésre és dokumentálásra.
Az 1990-es években ezért új, agilis módszerek jelentek meg. Az agilis módszerek mind az inkrementális fejlesztésen alapulnak, de különböző folyamatokat ajánlanak ennek elérésére.

Az agilis módszerek alapelvei:

- az ügyfél bevonása: minél jobban be kell vonni a fejlesztési folyamatba. Ők adják és rangsorolják a rendszerkövetelményeket, és kiértékelik a rendszer minden egyes iterációját.
- inkrementális átadás: lépésenként fejlődik a rendszer.
- az emberek nem folyamatok: a fejlesztőcsapat képességeit ismerni kell és ki is kell használni. Hagyjuk, hogy a csapattagok a feladatokat a saját módszerükkel végezzék el, nem pedig előírt folyamatok szerint.
- változtatási lehetőség: a rendszerkövetelmények gyakran változnak, erre fel kell készülni
- kezelési egyszerűség: az egyszerűségre kell koncentrálni, mind a fejlesztendő szoftver, mind a fejlesztési folyamat tekintetében. Amikor csak lehet, aktívan kell dolgozni a rendszer komplexitásának (összetettségének) csökkentésén.

Ezeket az alapelveket a gyakorlatban néha **nehéz** megvalósítani, mivel:

- az ügyfél fejlesztési folyamatba való bevonása vonzó, de ez csak akkor lesz sikeres, ha tud és akar is időt tölteni a fejlesztőcsapattal.
- előfordulhat, hogy egyes csapattagok nem rendelkeznek megfelelő személyi adottságokkal az intenzív részvételhez, ami viszont jellemző az agilis módszerekre.
- a változtatások fontossági sorrendbe való rendezése nehéz, ha sok az érintett, mivel ők általában eltérő prioritásúnak ítélik a változtatásokat.
- az egyszerűség fenntartása külön munkát igényel.

Az agilis módszerek közül a legismertebb és legszélesebb körben alkalmazott az **extrém programozás.**
Ennek lényege: minden követelményt forgatókönyvként állítanak össze (ennek neve: felhasználói történet), amely feladatokra bontható és egy-egy feladat önállóan implementálható. A programozók változó párokban dolgoznak és minden feladatra teszteket készítenek, mielőtt még a kódot megírnák. Minden tesztnek sikeresen le kell futnia, mielőtt az új kódot elhelyeznék a rendszerben. A rendszer kiadásai között csak kis idő telik el. Alapelv a folyamatos tökéletesítés is, a kódot át kell írni, átszervezni, hatékonyabbá tenni, amikor csak lehet. (A párokban való programozás érdekes módon majdnem olyan hatékony, mintha külön-külön dolgoznának, pedig egy feladattal foglalkoznak mind a ketten.)

Az extrém programozás kiadási ciklusa (az XP folyamata):

1. A kiadáshoz történő felhasználói történet kiválasztása
2. A történet feladatokra bontása
3. A kiadás tervezése
4. A szoftver fejlesztése/integrálása/tesztelése

5. A szoftver kiadása
6. A rendszer értékelése
(És újra az 1-es pont.)

Egy példa (LIBSYS-rendszer, könyvtári rendszer):

Felhasználói történet:
„Egy cikk letöltése és kinyomtatása
Először válasszuk ki egy megjelenített listából a kívánt cikket. Aztán adjuk meg a rendszernek a fizetési módot – ez lehet előfizetéses vagy vállalati számláról történő vagy hitelkártyával. Ezután kapunk egy kitöltendő szerzői jogi űrlapot. Ha ezt elküldtük, a cikk letöltődik a számítógépünkre. Ezután választunk egy nyomtatót és kinyomtatjuk a cikk egy másolatát. Közöljük a rendszerrel a sikeres nyomtatás tényét.
Ha a cikk csak nyomtatható, akkor nem őrizhetjük meg a PDF-verziót, így az automatikusan törlődik a számítógépünkről.”

Feladatokra bontása:
„3. feladat: A fizetési lehetőségek implementálása
A fizetés három különböző úton történhet. A felhasználó kiválasztja, hogy milyen módon szeretne fizetni. Ha a felhasználónak van könyvtári olvasójegye, akkor beírhatja ide annak azonosítóját, amit a rendszernek ellenőriznie kell. Másik lehetőség, hogy megad egy szervezeti számlaszámot. Ha az érvényes, akkor a cikknek megfelelő költséggel megterhelik a számlát. Végül beírható a 16 számjegyből álló hitelkártyaszám és lejárati dátum. Ellenőrizni kell ennek az érvényességét, és amennyiben érvényes, akkor terhelést kell küldeni a hitelkártyaszámlára.”

Teszteset-leírás:
„4. teszt: Hitelkártya érvényességének ellenőrzése
Bemenet:
A hitelkártyaszámot egy karaktersorozat, míg a kártya lejárati hónapját és évét két egész szám reprezentálja.
Tesztek:
Ellenőrizni kell, hogy a karaktersorozat minden bájtja számjegy-e. Ellenőrizni kell, hogy a hónap egy és 12 között van-e és az év nagyobb vagy egyenlő-e az aktuális évvel.
A hitelkártya számának első négy számjegye alapján ellenőrizni kell, a kibocsátó érvényességét a kártyakibocsátók táblázata alapján. A hitelkártya érvényességét ellenőrizzük úgy, hogy elküldjük a kártyaszámot és a lejárati dátuminformációt a kártya kibocsátójához.
Kimenet:
OK vagy hibaüzenet, ami a kártya érvénytelenségét jelzi.”

Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Programtervezési minták. Kiskapu Kiadó, Budapest 2004.*)*

XIII. Programtervezési minták (Design Pattern)

XIII.1. Mi is az a tervezési minta?

Objektumközpontú programot tervezni nehéz, újrahasznosíthatót még nehezebb. Meg kell találni a megfelelő osztályokat és az osztályok közötti kapcsolatokat. A tervnek igazodnia kell a megoldandó problémához, de eléggé általánosnak kell lennie ahhoz, hogy később könnyen módosítható legyen. A tapasztalt tervezők jó terveket készítenek, a kezdők viszont elvesznek a lehetőségek dzsungelében. Éppen ezért jó lenne, ha a tapasztalt tervezők-fejlesztők valamilyen módon át tudnák adni a kezdőknek a jól működő megoldásokat.
Ha egy problémára ismerünk egy megoldási mintát, ami már bevált, akkor azt legközelebb, hasonló problémánál is alkalmazhatjuk. Ezt felismerve, **a GoF (Gang of Four – „Négyek bandája”**: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) összegyűjtötte a tapasztalt objektum-orientált tervezők mintáit (23 darabot) és egységes leírást, magyarázatot készítettek hozzájuk. Így egy katalógust kaptunk, amelyben programtervezési minták találhatók.
Az ő definíciójuk szerint **a minta: „Egymással együttműködő objektumok és osztályok leírása, amely testreszabott formában valamilyen általános tervezési problémát old meg egy bizonyos összefüggésben.”** A tervezési minta azonosítja a részt vevő osztályokat és objektumokat, szerepüket és kapcsolataikat.

XIII.2. A minták részei és leírásuk

A mintáknak négy lényeges elemük van:
1. a minta neve
2. a probléma
3. a megoldás
4. a következmények.
A minta neve egy hivatkozási eszköz, hogy ne kelljen körülírni a tervezési problémát. A probléma rész írja le, hogy mikor alkalmazzuk a mintát. Néha feltételek is szerepelnek itt, amelyek teljesülése esetén érdemes csak ezt a mintát alkalmazni. A megoldás azokat az elemeket írja le viszonyaikkal, hatáskörükkel és együttműködési lehetőségeikkel, amelyek felépítik a tervet. A következmények részben szerepelnek a minta alkalmazásának előnyei és hátrányai.

A tervezési minták leírására a következő űrlap-alapú megközelítést használjuk (ezt az űrlapot kell kitölteni minden mintáról):
A minta neve és besorolása
Cél
Egyéb nevek
Feladat
Alkalmazhatóság
Szerkezet
Résztvevők
Együttműködés
Következmények

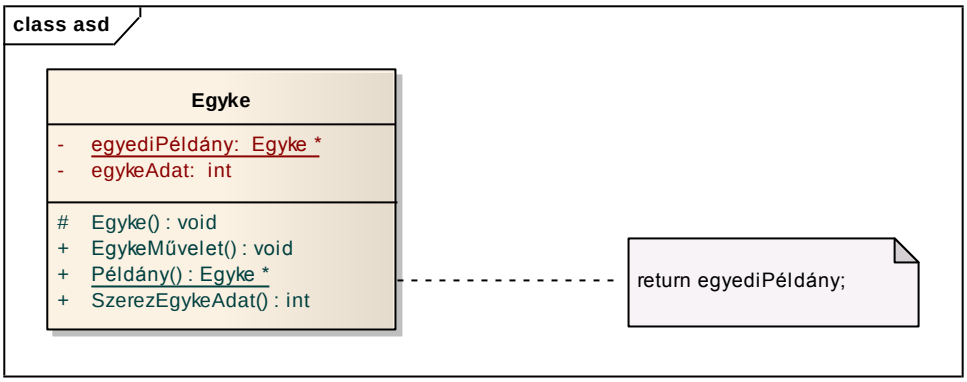
Megvalósítás
Példakód
Ismert felhasználások
Kapcsolódó minták

XIII.3. Egy konkrét minta bemutatása

A minta neve és besorolása: Egyke, objektum-létrehozási minta
Cél: Egy osztályból csak egy példányt engedélyezni, és ehhez globális hozzáférési pontot megadni.
Egyéb nevek: Singleton
Feladat: Egyes osztályok esetén fontos, hogy pontosan egy példány legyen belőlük. Egy rendszerben több nyomtató is lehet, de nyomtatási sorból csak egyet lehet használni. Vagy csak egy fájlrendszer és csak egy ablakkezelő futhat.
Hogyan biztosíthatjuk, hogy egy osztályból csak egyetlen példány legyen, de azt könnyen el lehessen érni? Ha globális változót hozunk létre, akkor az objektum könnyen elérhető lesz, de akkor akárhány példányt létrehozhatunk belőle.
Jobb megoldás, ha maga az osztály a felelős a példányok nyilvántartásáért.
Alkalmazhatóság:

- Pontosan egy példányra van szükség egy osztályból és annak elérhetőnek kell lennie az ügyfelek számára.
- Ennek az osztálynak leszármazott osztályokkal bővíthetőnek kell lennie, és az ügyfeleknek képeseknek kell lenniük a saját kódjuk módosítása nélkül használni a bővített osztály példányát.
-

Szerkezet:



Résztvevők:
Egyke:

- Meghatároz egy olyan Példány műveletet, amely lehetővé teszi, hogy az ügyfelek hozzáférjenek az osztály egyedi példányához. A Példány statikus tagfüggvény C++-ban (vagy osztálymetódus más nyelvekben).
- Felelős lehet saját egyedi példányának (objektumának) létrehozásáért.

Együttműködés: Az ügyfelek az Egyke példányt kizárólag az Egyke Példány műveletén keresztül érik el.
Következmények:
Az Egyke minta előnyei:

- Szabályozott hozzáférés az egyetlen példányhoz.

- Nincs globális változó.
- A leszármazással bővíthető az Egyke osztály funkcionalitása.
- Nem csak pontosan egy, hanem akárhány példány ellenőrzött létrehozására is alkalmas.

Megvalósítás:
C++-ban:

```
class Egyke
{
  public: static Egyke * Példány();
  protected: Egyke();
  private: static Egyke * egyediPéldány;
};
Egyke * Egyke::egyediPéldány = 0;
Egyke * Egyke::Példány()
{
 if (egyediPéldány == 0)
 egyediPéldány = new Egyke;
 return egyediPéldány;
}
```

Az ügyfelek az Egykéket kizárólag a Példány tagfüggvényen át érhetik el. Az egyediPéldány kezdőértéke nulla, és a statikus Példány függvény hozza létre az objektumot és állítja rá az egyediPéldány mutatót. Ez a tagfüggvény lusta előkészítést használ, az általa visszaadott értéket a program nem hozza létre és nem tárolja addig, amíg először hozzá nem férnek az objektumhoz.
A konstruktor védett (protected), tehát ezzel az ügyfél nem hozhat létre objektumokat.

Példakód:
Képzeljünk el egy labirintust generáló játékot. A labirintus szobákból áll, minden szobának 4 fala van, a falakon lehet ajtó, az ajtók lehetnek zárva vagy nyitva, stb. A labirintus alkalmazásnak a labirintust felépítő „Elvont gyár” (ez is egy minta) osztályát csak 1 pldányban kell létrehozni, erre lehet használni az Egyke mintát:

```
class LabirintusGyar
{
  public: static LabirintusGyar * Példány();
  protected: LabirintusGyar ();
  private: static LabirintusGyar * egyediPéldány;
};
LabirintusGyar * LabirintusGyar::egyediPéldány = 0;
LabirintusGyar * LabirintusGyar::Példány()
{
 if (egyediPéldány == 0)
 egyediPéldány = new LabirintusGyar;
 return egyediPéldány;
}
```

Ismert felhasználások: Az osztályok és metaosztályaik közötti kapcsolat. A metaosztály az osztályok osztálya és mindegyik metaosztálynak egy példánya van.
Kapcsolódó minták: Az Egyke mintával számos minta megvalósítható, például az Elvont gyár, az Építő és a Prototípus.

(Ajánlott irodalom: Ian Somerville: Szoftverrendszerek fejlesztése. Második, bővített, átdolgozott kiadás, Panem Kiadó, Budapest 2007.*)*

VERIFIKÁCIÓ ÉS VALIDÁCIÓ

XIV. Verifikáció és validáció

Ellenőrző és elemző folyamatok, amelyek biztosítják, hogy a szoftver megfelel a specifikációnak és kielégíti a megrendelő igényeit.
Validáció: „A megfelelő terméket készítettük-e el?"
Verifikáció: „A terméket jól készítettük-e el?"
A validáció az általánosabb folyamat, itt nem csak a specifikációnak való megfelelést vizsgáljuk, hanem a vásárló elvárásainak teljesülését is, amibe beleértendők a nem funkcionális tulajdonságok is: hatékonyság, hibatűrés, erőforrásigény.
Verifikálni és validálni nem csak a kész terméket kell, hanem minden fejlesztési fázisban, vagyis a követelményeket és a terveket is.

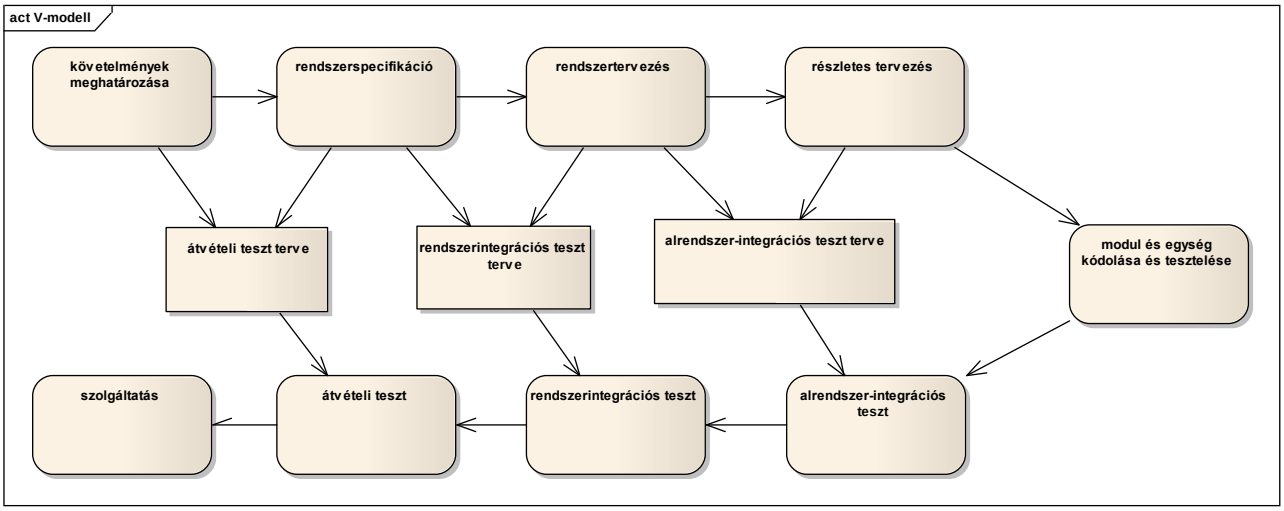
A V & V folyamaton belül a rendszer ellenőrzésére két technika használható:
- **Statikus:** szoftverátvizsgálások (szoftver alatt értve a követelményspecifikációt, tervet és a programszöveget is) és automatikus elemzések.
- **Dinamikus**: szoftvertesztelés. (Csak az implementációval.)

A tesztelésnek két fajtája ismert:
- **hiányosságtesztelés:** a program és a specifikációja között meglévő ellentmondások felderítése. Az ilyen teszteket a rendszer hiányosságainak feltárására tervezik, nem a valós működés szimulálására.
- **statisztikai (vagy validációs) tesztelés:** a program teljesítményének és megbízhatóságának tesztelése valós körülményeket szimulálva, vagyis a teszteseteknek a valós felhasználói bemeneteket és azok gyakoriságát kell mutatnia.

Másik megközelítésben **komponens- és rendszertesztelésről** beszélhetünk. A komponenstesztelésnél függvényeket, objektumokat vagy újrafelhasználható komponenseket tesztelünk. Ezeket utána alrendszerbe vagy a teljes rendszerbe integráljuk és azt nézzük meg, hogy az így kapott rendszer megfelelő funkcionális és nemfunkcionális tulajdonságokkal rendelkezik-e.

Bonyolultabb rendszereknél a rendszerteszt **integrációs és kiadástesztre** bomlik.

Verifikáció és validáció tervezés
A V & V költséges folyamat, bonyolult valós idejű rendszereknél akár a költségek 50%-át is elérheti, ezért gondosan meg kell tervezni.
A teszttervek a fejlesztés és a tervezés közötti kapcsolatot jelentik, azért nevezzük V-modellnek, mert az ábra alakja egy fekvő V-betű:



A szoftver átvizsgálása

A szisztematikus programtesztelés időigényes és drága folyamat. Minden tesztfuttatás egyetlen vagy legfeljebb néhány hibát derít fel. Ezzel ellentétben a programkód átvizsgálása hatékonyabb és olcsóbb. A program hibáinak több, mint 60%-a felderíthető programátvizsgálással.

A programkód átvizsgálásának hatékonyságát két ok magyarázza:

- Egy menetben több, független hiba is kiderülhet.
- Az átvizsgálók felhasználják a szakterületre és a programozási nyelvre vonatkozó ismereteiket. Valószínűleg találkoztak már régebben is ilyen hibafajtákkal, így tudják, mire kell figyelni.

A programkód átvizsgálását egy legalább 4 emberből álló csapatnak érdemes végeznie. A szerző, az olvasó, a tesztelő és a moderátor. Maga az átvizsgálás max. 2 óra, aztán a szerző módosítja a programot. Ezután vagy újra átvizsgálják a kódot, vagy a moderátor úgy dönt, hogy ez nem szükséges. Az átvizsgálás folyamatát a szokásos programozási hibák – nyelvtől függő - listájára kell alapozni. Egy lehetséges hibalista:

- Adathibák
- Vezérlési hibák
- Input/output hibák
- Interfészhibák
- Tárkezelési hibák
- Kivételkezelési hibák

Automatizált statikus elemzés

A statikus programelemzők olyan szoftvereszközök, amelyek a program forrásszövegének vizsgálatával derítik fel a lehetséges hibákat és anomáliákat. Ehhez nincs szükség a program futtatására. Észlelhetik a nem használt kódrészleteket, inicializálatlan változókat, stb. Néhány statikus elemzéssel felismerhető hibafajta és anomália (az anomális nem feltétlenül programhiba, lehet következmény nélküli is):

- Adathibák
- Vezérlési hibák
- Input/output hibák
- Interfészhibák
- Tárkezelési hibák

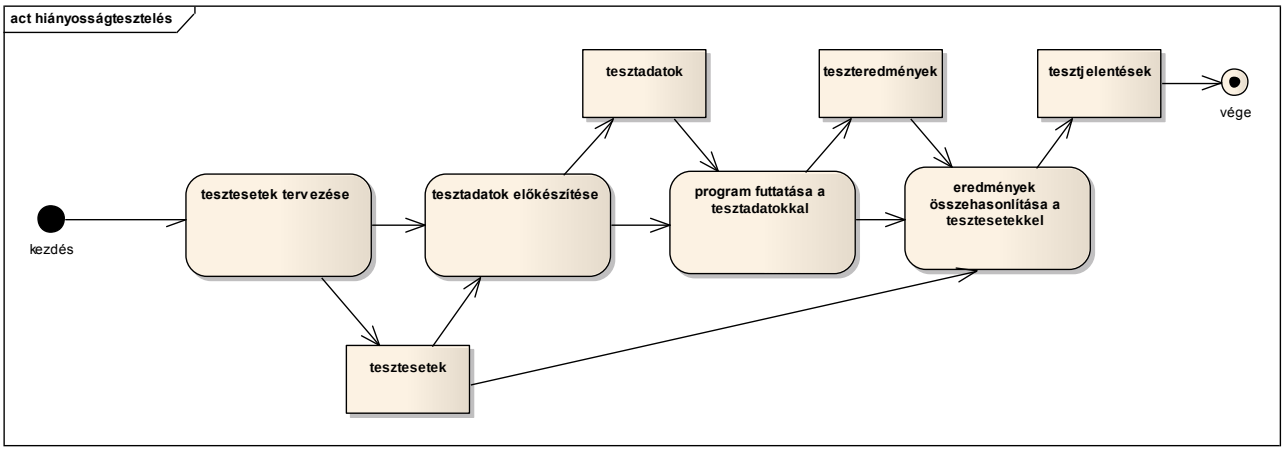
A statikus elemzés szakaszai:
- a vezérlési folyamat elemzése (ciklusok, elérhetetlen kódrészek)
- az adathasználat elemzése
- interfészelemzés
- információáramlás elemzése (a bemenő és kimenő változók közötti függőségek felderítése)
- útvonalelemzés (a program összes lehetséges végrehajtási útvonalának meghatározása)

A statikus elemzők különösen hasznosak ha C-t, vagy hozzá hasonló gyengén típusos nyelvet használunk. A Unix és Linux rendszerek tartalmaznak egy LINT nevű, C-programokhoz készült statikus elemzőt.
Az automatizált statikus elemzés Java-nál és a modern C++ fordítók használata esetén nem hatékony.
Az automatizált statikus elemzés nem helyettesítheti az átvizsgálásokat, mert bizonyos hibafajtákat nem tud felismerni. (Pl. a helytelen kezdeti értékadást)

XV. Szoftvertesztelés

XV.1. Hiányosságok tesztelése

A cél, hogy a rendszer átadása előtt feltárjuk a rejtett hibákat. A sikeres hiányosságtesztelés tehát olyan tesztelés, ami *a rendszer helytelen működését okozza*, így feltár egy hibát. Sajnos a teszteléssel csak a hibák jelenlétét lehet felderíteni, a hibátlanságot nem lehet vele megmutatni!
A hiányosságtesztelés folyamatának általános modellje:



Teszteset: az inputok és a rendszertől várt outputok specifikációja, és egy ismertető, hogy mit tesztelünk.
Tesztadatok: kifejezetten a rendszer tesztelésére létrehozott inputok. Ezek néha automatikusan is generálhatók.
A tesztelési tapasztalatok alapján néhány irányelv adható a hiányosságtesztelés eredményességének javítására:
1. Válasszunk olyan bemeneteket, amelyek rákényszerítik a rendszert arra, hogy az összes hibaüzenetet legenerálja.
2. Alkalmazzunk olyan inputokat, amelyek a bemeneti pufferek túlcsordulását eredményezik.
3. Ugyanazon inputot vagy inputsorozatot több alkalommal ismételjük meg.
4. Kényszerítsük ki az érvénytelen outputok generálását.

5. Kényszerítsük ki azt, hogy a számítási eredmények túl kicsik vagy túl nagyok legyenek.

Fekete doboz tesztelés (funkcionális tesztelés)
A rendszer belseje nem ismert, viselkedésmódja csak a bemenetekre adott válaszok tanulmányozásával érthető meg. Itt a tesztek a program vagy komponensspecifikációból származnak. A tesztelőnek olyan bemeneteket kell választania, amelyek rendellenes viselkedést okoznak. Az ilyen tesztesetek kiválasztása a tapasztalatokon és a szakterületi ismereteken alapul.

Ekvivalencia-osztályozás
Egy program inputjai általában különböző osztályokba esnek. Ezek rendelkeznek valamilyen közös jellemzővel, pl. pozitív vagy negatív számok, szóköz nélküli sztringek, stb. A programok általában az osztály minden tagjára hasonló módon viselkednek. Ezért nevezik ezeket ekvivalencia – osztályoknak vagy tartományoknak. Ekvivalencia-osztály például az érvénytelen és az érvényes inputok halmaza is. A hiányosságtesztelés szisztematikus megközelítése az ekvivalencia-osztályok azonosításán alapul. Először meg kell határozni az osztályokat, majd minden osztályból teszteseteket kell választani. Az osztály határáról és közepéről is érdemes teszteseteket választani. Az ekvivalencia-osztályok a programspecifikáció vagy a felhasználói dokumentáció alapján azonosíthatók.
Pl.: a program legalább 4 és legfeljebb 10 darab inputot vár, amik 10.000-nél nagyobb, ötjegyű számok. Itt az ekvivalencia-osztályok:
input értékek száma: kevesebb, mint 4; 4 és 10 között; 10-nél több;
input értékek: kisebb, mint 10.000; 10.000 és 99.999 között; nagyobb, mint 99.999
lehetséges tesztadatok: 3,4,7,10,11 darab; 9.999, 10.000, 50.000, 99.999, 100.000 érték

másik példa: egy keresőrutin, amelyet nem üres sorozatokon való keresésre terveztek és vagy visszaadja a megtalált elem indexét és a Found logikai változót igazra állítja, vagy ha nem találta, akkor hamisra és az index érték definiálatlan lesz.
A keresőrutin ekvivalenciaosztályai:

Sorozat	Elem
Egy érték	A sorozatban van
Egy érték	Nincs a sorozatban
Több, mint egy érték	Első elem a sorozatban
Több, mint egy érték	Utolsó elem a sorozatban
Több, mint egy érték	Köztes elem a sorozatban
Több, mint egy érték	Nincs a sorozatban

És az ez alapján készült tesztesetek:

Sorozat	Amit keresünk	A kimenet (Found, index)
25	25	true, 1
25	6	false, ???
14,25,36,4,8	14	true, 1
45,74,36,1,85,2	2	true, 6
71,85,69,23,15	23	true, 4
71,85,69,23,15	40	false, ???

Fehér doboz vagy struktúrateszt
Ezt a szoftver implementációjának ismeretében készítjük. Többnyire kis programegységekre,

objektumokra, alprogramokra alkalmazzák. Az algoritmusról szerzett tudás alapján pontosabban tudjuk azonosítani az ekvivalencia-osztályokat. Pl.: a keresőeljárás legyen a bináris keresés, ahol a sorozatot egy rendezett tömbként adjuk át. A kód vizsgálatával láthatjuk, hogy a keresési teret három részre lehet osztani: középső elem, nála kisebbek és nála nagyobbak.

Útvonal tesztelés

Fehér-doboz tesztelési stratégia, amelynek az a célja, hogy a programban minden független végrehajtási útvonalat kipróbáljunk. Ebben az esetben minden utasítás legalább egyszer lefutott és minden feltétel tesztelt igaz és hamis esetre is.

A programban található független utak száma általában arányos a program méretével, ezért az útvonal tesztelést az egység- és modultesztnél használják.

Az útvonal tesztelés kezdete egy programfolyamat-gráf. Ha nincs goto a programban, akkor ezt könnyű előállítani. A szekvenciális utasítások (értékadás, eljáráshívás, I/O utasítás) kihagyhatók a gráfból. A feltételes utasítások (if, case) minden ága külön út lesz a gráfban, a ciklusok pedig a ciklusfeltételt reprezentáló pontba visszamutató nyilak.

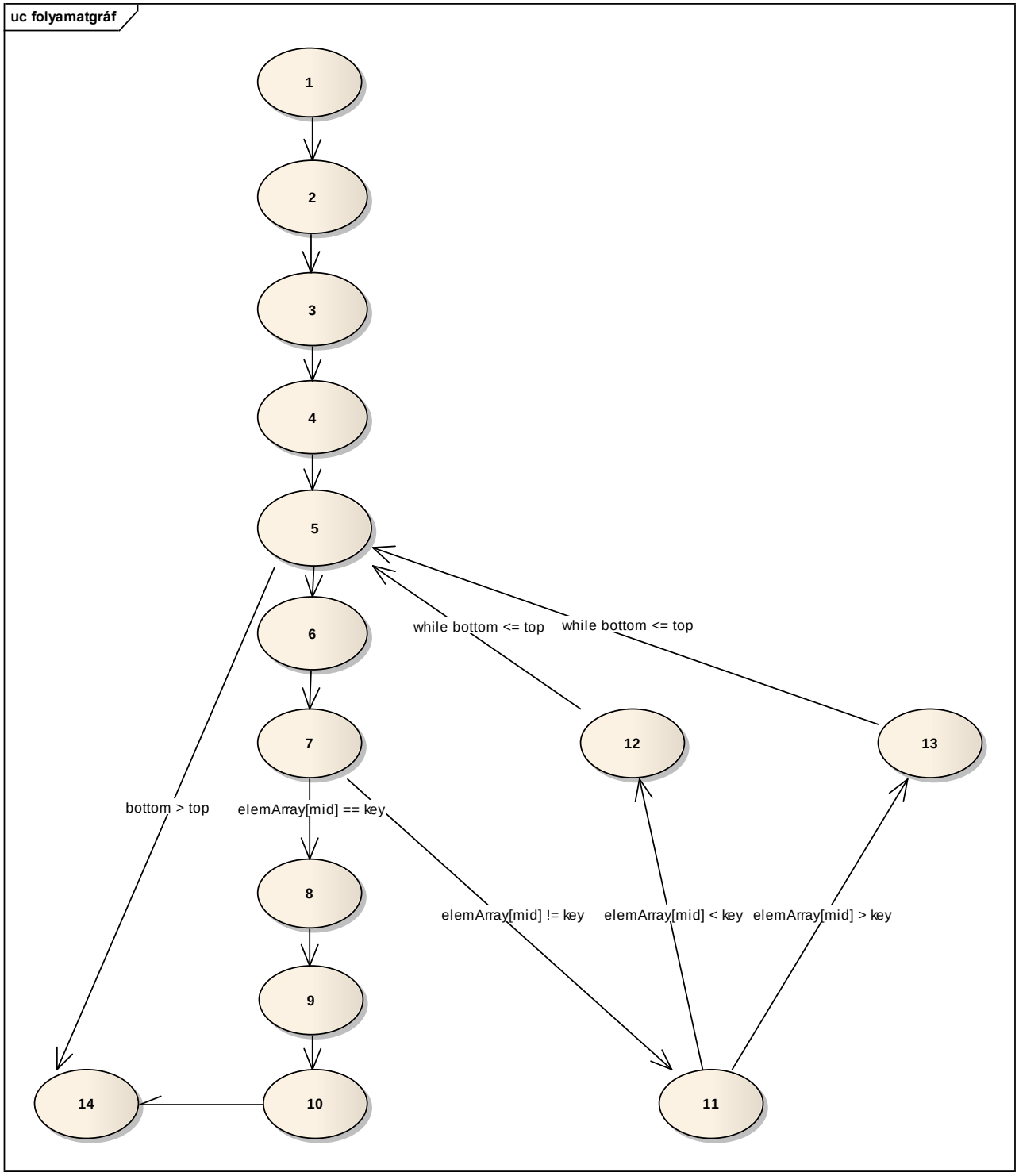
Nézzük a következő Java-nyelvű bináris keresőrutint:

Első paramétere a keresendő elem, a második egész számok rendezett tömbje, amiben keresünk és a harmadik egy objektum, ami a keresés eredményét fogja tartalmazni. Ez utóbbi egy logikai részből áll, hogy megtalálta-e a tömbben a keresett elemet, és ha igen, akkor a másik része tartalmazza a megtalálás helyét, vagyis az adott elem tömbindexét.

```
class BinSearch
{
public static void search(int key, int [] elemArray, Result r)
{
     int bottom = 0; //1.
     int top = ellemArray.length – 1; //2.
     int mid; //3.
     r.found =false; r.index = -1; //4.

     while(bottom <= top) //5.
     {
          mid = (top + bottom) / 2;  //6.
          if (elemArray [mid] == key) //7.
          {
               r.index = mid;  r.found = ture;  return; //8-9-10.
          }
          else
          {
               if (elemArray[mid] < key) bottom = mid + 1; //11-12.
               else top = mid – 1; //13.
          }
     }
} //14.
}
```

Ennek a folyamatgráfja a következő lesz:



A példában a független utak a következők:
1,2,3,4,5,6,7,8,9,10,14
1,2,3,4,5,14
1,2,3,4,5,6,7,11,12,5,...
1,2,3,4,5,6,7,11,13,5,....
Egy programban a független utak száma meghatározható a folyamatgráf ciklomatikus

komplexitásával is. CG(G) = élek száma – csomópontok száma + 2. (A példában: 16-14+2 = 4.)
Azon programoknak, amelyek nem tartalmaznak goto-t, a ciklomatikus komplexitása eggyel több, mint a programban található feltételek száma. (Feltétel alatt itt egyszerű logikai feltételt értünk. Ha egy logikai kifejezés tartalmaz logikai operátort, pl. or vagy and-et, akkor az már összetett feltételnek számít, ami annyi darab egyszerű feltételnek felel meg, ahány ilyen feltételt kapcsoltunk össze benne or-ral vagy and-del. pl. if (A || B && C) ez három egyszerű feltételnek felel meg.)
Ennek a strukturális tesztelési módszernek az a célja, hogy bizonyítsa, hogy minden független programútvonalon végigmentünk. Egy független programútvonal olyan út, amely a folyamatgráfban legalább egy új csomóponton keresztülmegy.
A valamennyi független programútvonal teszteléséhez minimum annyi teszteset szükséges, mint amennyi a gráf ciklomatikus komplexitása.
Egyszerű programoknál nem nehéz a folyamatgráf felrajzolása, de amikor egy program nagyon összetett szerkezettel rendelkezik, akkor érdemes dinamikus programelemzőt használni a program futási profiljának felderítésére.
A **dinamikus programelemzők** olyan, a fordítóprogrammal együttműködő tesztelő eszközök, amelyek számolják, hogy az egyes utasítások hányszor kerülnek végrehajtásra. Miután a program lefutott, a futási profil megmutatja, hogy a program mely részei futottak, illetve maradtak ki a tesztesetek végrehajtása során.

Szoftvertechnológia

© Szabolcsi Judit
2012

(Ajánlott irodalom: Ian Somerville: Szoftverrendszerek fejlesztése. Második, bővített, átdolgozott kiadás, Panem Kiadó, Budapest 2007.*)*

MENEDZSMENT

XVI. Emberek menedzselése

A szoftverekkel foglalkozó cégek **legnagyobb vagyonát az ott dolgozó emberek** alkotják. Ők adják a szervezet szellemi tőkéjét. A szoftvermenedzserek feladata pedig az, hogy az emberekbe fektetett tőke minél jobban kamatozzon. Ennek feltétele, hogy **a vállalatoknak meg kell becsülniük a dolgozóikat. Az embereknek megfelelő szintű felelősséget kell adni, és képességeikhez mérten kell jutalmazni őket.**
A projektek bukásának egyik legfőbb oka a rossz humánmenedzsment. A vezetők sokszor nem veszik figyelembe az egyének korlátait, teljesíthetetlen határidőket szabnak és hajcsárként viselkednek. Úgy támasztanak új követelményeket, hogy nem elemezték ennek hatását a projektcsapatra és a termékre.
Az emberek menedzselésének négy kritikus tényezője van:

1. **Következetesség.** Az embereket a csapatban azonos módon kell kezelni.
2. **Elismerés.** A különböző emberek különböző szaktudással rendelkeznek. A csapat minden tagjának meg kell adni a lehetőséget a közreműködésre.
3. **Befogadás.** Olyan munkakörnyezet kialakítása, ahol mindenki véleményét (még az újakét, vagy a legfiatalabbakét is) figyelembe veszik.
4. **Őszinteség.** A vezetőnek mindig őszintén el kell mondania, hogy a csapatban mi megy jól és mi megy rosszul. Őszintén kell nyilatkoznia a saját tudásáról és el kell fogadnia a beosztottak véleményét, ha a saját ismeretei hiányosak.

XVI.1. A személyzet kiválasztása

A projektvezető egyik feladata a projekten dolgozó emberek kiválasztása. Ehhez általában a következő információk szükségesek:

7. **A pályázó szakmai önéletrajza.**
8. **A személyes interjúból nyert információk.** Tud-e kommunikálni, milyenek a szociális képességei.
9. **A pályázót személyesen ismerők (volt kollégái, főnökei) ajánlásai**. Ezt csak akkor érdemes figyelembe venni, ha ismerjük az ajánlót, mert egyébként az ajánlás igazságtartalma nem ítélhető meg.
10. **Tesztek.** Szakmai tudást mérő tesztek és pszichometriai tesztek, amelyekkel a jelentkező pszichológiai profilját állítják elő, megadva az illető bizonyos típusú feladatokra való alkalmasságát és hozzáállását.

A kiválogatást befolyásoló tényezők:

11. gyakorlat az alkalmazási területen
12. tapasztalat a platform terén (rendszerint nem kritikus tényező, mert aki jó problémamegoldó képességgel rendelkezik és vannak szakterületei tapasztalatai, az hamar megismeri az új platformot)
13. gyakorlat a programozási nyelvben (ez is csak rövid ideig tartó projekteknél

lényeges, ahol nincs idő egy új nyelv megtanulására)
14. problémamegoldó képesség
15. iskolázottság (ahogy az illető egyre több szakmai tapasztalatot szerez, úgy csökken ennek a jelentősége)
16. kommunikációs képesség
17. alkalmazkodó képesség
18. hozzáállás (pozitívan kell hozzáállnia a munkájához és hajlandóknak kell lennie új dolgokat tanulni)
19. személyiség (ki kell jönnie a csapat többi tagjával)

XVI.2. Az emberek motiválása

A projektvezető egyik feladata, hogy motiválja a vele dolgozó embereket. A motiváció a munka és a munkakörnyezet olyan megszervezését jelenti, amelyben az ember a leghatékonyabban tud dolgozni. Ha az ember nem motivált, akkor nem érdekli az éppen végzett munka, lassan dolgozik, sokat hibázik.
Maslow szerint az embereket a szükségleteik kielégítés motiválja. A Maslow-féle szükségletpiramis alján a **fiziológiai szükségletek** kielégítése áll (evés, ivás, alvás, megfelelő hőmérséklet, stb.). Ezután jönnek a **biztonsági szükségletek** (fizikai, az életet, az egészséget veszélyeztető tényezők kiiktatása). Utána a **szociális szükségletek** (csoporthoz tartozás szükséglete: család, barátok, munkahelyi közösség). A **megbecsülés iránti igény** (elismerjék az ember munkáját, képességei, tudását), végül az **önmegvalósítás igénye** (alkotásra való lehetőség, saját képességek kifejlesztése, csiszolása) zárja a sort.
Maslow szerint ezek a szintek egymásra épülnek, és csak az alacsonyabb szintek biztosítása után lehet feljebb lépni.

A szoftverfejlesztőknél nyilván a felső három szint kielégítése a legfontosabb.

- Az alkalmazottak szociális szükséglete kielégíthető, ha engedélyezett a munkatársakkal való találkozás, informális keretek között is.
- A megbecsülés iránti igény kielégíthető, ha a szervezet érezteti velük, hogy megbecsüli őket, pl. nyilvánosan elismerik a képességeiket és a képességeiknek és a gyakorlatuknak megfelelően fizeti őket.
- Az önmegvalósítási igényt a képességeknek megfelelő (nehéz, de nem lehetetlen) feladatokkal lehet kielégíteni. Emellett a továbbképzések szervezése is jó módszer.

Bass és Dunteman szerint az embereket három típusba lehet sorolni:

- **Feladatorientált** az, akit a munkája, az elvégzendő feladat intellektuális kihívása motivál.
- **Önorientált** az, akit elsősorban a személyes sikerek és az elismerés motivál. Ők a munkájukat eszközként használják saját céljaik elérésére. Gyakran hosszú távú céljaik vannak, pl. karrierépítés, saját vállalkozás felépítése, stb.
- **Kapcsolatorientált** az, akit a munkatársai jelenléte, támogatása, elismerése motivál. Ők akkor dolgoznak hatékonyan, ha jó a munkahelyi légkör, barátságosak a kollégák.

A kapcsolatorientált emberek általában szeretik a csoportmunkát, a feladat- és önorientáltak viszont inkább egyedül szeretnek dolgozni. Egy sikeres projektbe mind a három típusú ember szükséges.

XVI.3. Csoportok kezelése

A legtöbb professzionális szoftvert olyan csoportok fejlesztik, ahol **a létszám kettő és néhány száz** ember között változik. A nagy csapatban lehetetlen, hogy mindenki egy problémán dolgozzon, ezért kisebb csoportokra kell felosztani. Ezért a gyakorlatban **egy projektcsoport mérete nem nagyobb 8-10 főnél.**
A vezetői feladatok kritikus pontja a hatékonyan együttműködő csoport összeállítása. Nyilván fontos, hogy egy csoporton belül meglegyen a megfelelő szakmai tudás és gyakorlat, de az is kell, hogy csapatszellem is kialakuljon. A jó csapatban a tagokat a csoport sikere ugyanúgy ösztönzi, mint a saját, személyes céljaik.
A csoportmunkát a következő tényezők befolyásolják:

- **A csoport összetétele**. Megfelelő-e a csoporton belül a szaktudás, a tapasztalat és az egyéniség aránya?
- **A csoportösszetartás**. A csoport egy csapatnak vagy ideiglenesen együtt dolgozó egyének gyülekezetének tartja magát?
- **A csoportkommunikáció**. Hatékony-e csoporton belüli kommunikáció?
- **A csoport szerkezete**. Biztosítja-e, hogy minden tag megbecsülve érezze magát és elégedett lehet a csoportbeli szerepével?

Ha a csoport valóban összetartó, annak a következő **előnyei** lesznek:

- **Kifejleszthető a csoport minőségi szabványa**. Mivel ez a szabvány általános megegyezésen alapszik, jobban igazodnak hozzá, mint a csapatnak előírt külső szabványhoz.
- **A csoporttagok szorosan együttműködnek**. A csoport tagjai tanulnak egymástól, ezért minimálisra csökkenthető a tudatlanság okozta problémák száma.
- **A csoport tagjai megismerhetik egymás munkáját.** A csoport egy tagjának távozása nem okoz nagy problémát.
- **Csoportos programozás végezhető.** A programokat nem egyes személyek, hanem a csoport tulajdonának tekintik.

Két probléma szokott előfordulni, erős, összetartó csoportoknál:

5. **A vezetőváltás ellenzése**. Ha egy összetartó csoport vezetőjét csoporton kívüli személlyel akarják helyettesíteni, a csoporttagok összefoghatnak az új vezető ellen, pl. nem fogadják el a változtatásait. Ezért, ha csak lehet, az új vezetőket jobb a csoportból választani.
6. **Csoportgondolkodás.** Ez annak a helyzetnek a neve, amikor a csoporthűség eltompítja a tagok kritikai érzékét, vagyis minden kritika nélkül elfogadják azt a feltevést, amit a csoport többsége támogat. A csoportgondolkodás elkerülhető külső szakértők bevonásával, vagy formális összejövetelek szervezésével, ahol a csoport tagjai kritizálhatják a döntéseket.

A csoportkommunikáció
A szoftverfejlesztő csapat tagjai között elengedhetetlen a jó kommunikáció. A tagoknak információt kell cserélniük arról, hogy hol tartanak a munkában, milyen döntések születtek, milyen korábbi döntések változtak meg, amelyek fontosak lehetnek.
A jó kommunikáció a csoportösszetartást is erősíti, mivel a csoporttagok megismerik egymást, így tudni fogják, hogy mik a többiek erősségei, gyengeségei, mi motiválja őket.

A kommunikáció hatékonyságát befolyásoló tényezők:

- **A csoport mérete.** A csoport méretének növekedésével egyre nehezebb biztosítani, hogy minden tag hatékonyan kommunikáljon egymással. Egy n tagú csoport esetén mindenkinek n-1 másik emberrel kellene kapcsolatot tartania, így aztán egy 7-8 fős csoportban elég valószínű, hogy lesznek olyan emberek, akik ritkán érintkeznek másokkal.
- **A csoport szerkezete**. Az informális csoportokban a kommunikáció sokkal hatékonyabb, mint a formális, hierarchikus szervezetekben. A több fejlesztői csoportból álló, nagy projektek sajátos problémája, hogy a hierarchikus csoportokban csak felfelé és lefelé áramlik az információ, az egy szinten állók között nem. Így a különböző fejlesztői csoportok csak a vezetőiken keresztül érintkeznek.
- **A csoport összetétele.** Ha túl sok hasonló típusú személyiség dolgozik együtt, akkor ezek összeütközésbe kerülhetnek. A kommunikáció általában jobb a férfiakat és nőket is tartalmazó csoportokban.
- **A csoport munkájának fizikai környezete.** A munkahely elrendezése a kommunikációt segítheti vagy gátolhatja is.

A csoport összekovácsolódását a **munkakörnyezet** is befolyásolja. A programozóknak szükségük van **egyedüllétre**, hogy tudjanak összpontosítani és ne zavarják meg őket munka közben. Ezen kívül a személyes tér azért is fontos, mert mindenkinek mások a munkaszokásai és elképzelései a díszítésről. A környezetét az ember szereti **személyessé tenni**. Emellett az embereknek szükségük van **természetes fényre és természetes környezetre** (legalább az ablakból fákat, növényeket láthassanak).
A csoportnak szüksége van **közös térre (tárgyaló)** is, ahol a megbeszélések és egyéb közös munka zajlik. Emellett kávézó, konyha, vagy más **nem hivatalos helyiség** is ajánlott, mert a kávészünetben is lehet ötleteket cserélni vagy problémákat tisztázni.

ÚJ TECHNOLÓGIÁK

XVII. Biztonságos rendszerek tervezése

Az internet rohamos elterjedése új kihívással szembesítette a szoftvertervezőket. Olyan rendszereket kell tervezniük és implementálniuk, amelyek **biztonságosak.** Ha egy rendszer csatlakozik az internethez, akkor ki van téve szakértő és rosszindulatú külső támadásoknak.
Ma már elengedhetetlen úgy megtervezni a rendszereket, hogy állják a külső támadásokat és helyre tudjanak állni egy ártó kísérlet után.
A támadók kárt tehetnek a rendszer hardverében, bizalmas adatokat szerezhetnek meg, vagy akadályozhatják a rendszer működését.

XVII.1. A rendszer felépítése

A biztonsági kérdések megfontolásakor figyelembe kell venni, hogy nem csak az alkalmazásszoftvert támadhatják, hanem a rendszer alapját képező infrastruktúrát is. A mai rendszerek a következő rétegekből állnak:

- Alkalmazás
- Újrafelhasználható komponensek és könyvtárak (pl. a .NET Framework)
- Köztes szoftver (pl. adatbázis providerek, ODBC, OleDb, stb.)

- Adatbázis-kezelő
- Általános, megosztott alkalmazások (pl. böngészők, e-mail kliensek)
- Operációs rendszer

Az alkalmazás alatti rétegeket együtt nevezzük infrastruktúrának. Ez könnyebben támadható, mivel közismert és széles körben elterjedt komponenseket (pl. böngészők) tartalmaz.

Fontos megkülönböztetni az alkalmazás és az infrastruktúra biztonságát:

- Az alkalmazás biztonsága szoftvertervezési probléma, úgy kell megtervezni, hogy ellenálljon a támadásoknak.
- Az infrastruktúra biztonsága ezzel szemben rendszerkezelői, rendszergazdai probléma, úgy kell konfigurálni.

XVII.2. Biztonsági alapfogalmak

Vagyon – Értékes és védelmet igénylő erőforrás.
Függőség – Az a veszteség vagy kár, amit egy támadás okoz. Lehet adatvesztés vagy adatsérülés, vagy az az idő- és energiaveszteség, amit a rendszer helyreállítása okoz.
Sérülékenység – A rendszer gyenge pontja, amelyet kihasználva kár okozható.
Támadás – A sérülékenység kihasználása. Általában kívülről érkezik.
Fenyegetések – Olyan körülmények, amelyek veszteséget vagy kárt okozhatnak. Ilyen a rendszer gyenge pontjának támadása.
Ellenőrzés – A rendszer sérülékenységét csökkentő tevékenység. Pl. jelszóellenőrző rendszer, ami kiszűri a gyenge jelszavakat.

Az ellenőrzéseket három csoportba sorolhatjuk:

- A támadást megakadályozó ellenőrzések. Pl. egy érzékeny katonai rendszert eleve nem kötnek hozzá a nyilvános hálózathoz.
- A támadások felderítését és visszaverését segítő ellenőrzések. Figyeljük a rendszer működését és ha szokatlan tevékenységet tapasztalunk, akkor figyelmeztetést küldünk, valamint leállíthatjuk a rendszer egyes részeit vagy korlátozhatunk bizonyos felhasználókat a hozzáférésben.
- A rendszer helyreállítását támogató ellenőrzések. Automatikus mentések, biztosítás kötés, ami fedezi a helyreállítás költségeit, stb.

XVII.3. Tervezési irányelvek

Nincsenek előre definiált szabályok arra, hogy hogyan lehet biztonságossá tenni egy konkrét rendszert. Mindössze általános irányelvek vannak.

1. **Explicit biztonságpolitikára építsük a biztonsági döntéseket**. A biztonságpolitika a szervezet felső szintjének állásfoglalása. Ha a megrendelő biztonságpolitikája azt írja elő, hogy csak a meghatalmazott dolgozók módosíthatnak bizonyos adatokat, akkor ezt a számukra fejlesztett szoftverbe is bele kell építeni.
2. **Kerüljük a kritikus meghibásodási pontokat.** A rendszer egy adott pontjának sérülése ne vezessen teljes rendszerösszeomláshoz. Ne csak egyetlen védő mechanizmust alkalmazzunk,

hanem használjunk különféle technikákat. Például a jelszót kiegészíthetjük kérdés-válasz ellenőrzéssel, vagy a rendszerben tárolt adatokat lemásolhatjuk minden változtatás előtt.

3. **Biztonságos bukás.** A rendszer valamilyen szintű meghibásodása elkerülhetetlen, a cél az, hogy ilyenkor is mentsük a menthetőt. A támadó ne férhessen hozzá azokhoz az adatokhoz, amikhez normál körülmények között nem tudna. Pl. a szerverről a kliensre letöltött adatokat a szerver elleni támadás esetén is elérjük, de érdemes ezeket kódolni a kliensoldalon, hogy illetéktelenek ne érhessék el.
4. **A biztonság és a használhatóság legyen egyensúlyban.** Ezek egymásnak ellentmondó szempontok, ha sok az ellenőrzés, a felhasználó előbb-utóbb felírja a jelszavait egy cetlire és odarakja a gép közelébe. Ezért egy pont után már nincs értelme újabb és újabb biztonsági mechanizmusokat beleépíteni a rendszerbe.
5. **Gondoljunk a szociális tervezésre.** Ez azt takarja, hogy rászedik a jogosult felhasználót, hogy bizalmas információt szolgáltasson ki. Előfordulhat, hogy valaki főnöknek adja ki magát és az új dolgozótól megpróbál hozzáférést szerezni, mondván, hogy valamiért nem engedi be a rendszer. Tervezési szempontból nagyon nehéz ezt a problémát kezelni, ezért ha erősen biztonságkritikus a rendszer, akkor nem elég a felhasználónév-jelszó páros, hanem digitális tanúsítványokra, vagy biometrikus ellenőrzésre van szükség és persze állandó naplózásra, hogy ki, mikor, mihez fért hozzá.
6. **Használjunk redundanciát és diverzitást a kockázat csökkentésére.** A redundancia azt jelenti, hogy a szoftverből vagy az adatokból több példány is létezik a rendszerben. A diverzitás pedig azt jelenti, hogy a különböző verziók nem épülhetnek ugyanarra a platformra vagy nem használhatják ugyanazt a technológiát, hogy a platform vagy a technológia sebezhetősége ne rántsa magával az egész rendszert. Pl. lehet a szerveren Linuxot, a klienseken pedig Windows-t futtatni.
7. **Validáljuk a bemenetet.** Gyakori támadástípus, amikor a támadók váratlan bemenetet adnak a rendszernek, amitől az kiszámíthatatlan viselkedést produkál. Ezt a problémát elkerülhetjük, ha minden lehetséges bemenetet validálunk, ellenőrzünk. Pl. a neveknél ellenőrizzük a hosszát, tartalmazhat-e speciális karaktert (pl. szóközt), stb.
8. **Tagoljuk a vagyonokat.** A rendszerhez való hozzáférés nem mindent vagy semmit alapon történik, hanem minden felhasználó csak a számára engedélyezett részhalmazhoz férhet hozzá. Így egyes információk sérülhetnek vagy elveszhetnek, de ez nem fogja érinteni az adott hozzáféréssel nem elérhető részeket.
9. **Gondoljunk tervezéskor a rendszer bevezetésére.** Sok biztonsági problémának az a forrása, hogy a rendszert telepítéskor nem megfelelően konfigurálják. Konfigurálni kell a megengedett felhasználók listáját, megváltoztatni az alapértelmezett név-jelszó párost (admin-password). Megtekinthetővé kell tenni a konfigurációs beállításokat, minél kevesebb privilégiumot kell adni a felhasználóknak, csak éppen annyit, ami a munkájukhoz kell.
10. **Gondoljunk tervezéskor a rendszer helyreállítására.** Legyen terv meghibásodás esetére is. Pl. illetéktelen hozzáférés esetén könnyű legyen megváltoztatni az összes jelszót, és ne engedje be a rendszer a jogosult felhasználókat addig, amíg meg nem változtatják a régi jelszavukat.