Szoftverfejlesztés

Ficsor Lajos (1,3,4,7,8,9,10,11,12,13 fejezet)
Krizsán Zoltán (14,16 fejezet)
Dr. Mileff Péter (2,5,6,15 fejezet)
2011, Miskolci Egyetem, Általános Informatikai Tanszék

Szoftverfejlesztés

írta Ficsor Lajos (1,3,4,7,8,9,10,11,12,13 fejezet), Krizsán Zoltán (14,16 fejezet), Dr. Mileff Péter (2,5,6,15 fejezet), és 2011, Miskolci Egyetem, Általános Informatikai Tanszék

Kelet-Magyarországi Informatika Tananyag Tárház

Nemzeti Fejlesztési Ügynökség http://ujszechenyiterv.gov.hu/ *06 40 638-638*

Lektor

Dr. Stefán Péter

NIIF, Budapest

A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával a TÁMOP-4.1.2-08/1/A-2009-0046 számú Kelet-Magyarországi Informatika Tananyag Tárház projekt keretében valósult meg.

Tartalom

Az ábrák listája

A táblázatok listája

1. fejezet - Bevezetés

A mai követelményeknek megfelelő alkalmazások iparszerű fejlesztése összetett folyamat, és megfelelő technológia hátteret igényel. Az évek során kialakult szoftver technológiai folyamat ajánlások (módszertanok) eleinte az egyéb jellegű gyártási folyamatokat vették mintának, amelyek azonban a szoftver, mint termék és előállítási folyamatának jellegzetességei miatt csak korlátozottan voltak felhasználhatók. Napjainkra azonban kialakultak azok az alapvető elvek, amelyek ezen a területen is alkalmazhatók.

A jegyzet az informatikus alapszakok Szoftver technológia című tárgyának anyagához kapcsolódik. Célja megismertetni az olvasót az alkalmazások fejlesztésnek alapvető munkafolyamataival és az ennek segítésére kialakult eszközkészlettel. Olyan ismereteket foglal össze, melyeknek birtokában a hallgatók képesek lesznek végzés után a gyakorlatban folyó fejlesztési folyamatokba bekapcsolódni, és saját résztevékenységüket a teljes folyamatban elhelyezni, a folyamat többi részvevőjével megfelelően kommunikálni.

Ennek érdekében összefoglalja a szoftver technológia fogalmát, alapvető sajátosságait, a folyamat bonyolultságát eredményező tényezőket. Áttekinti az idők során kialakult szoftver folyamat modelleket, a klasszikus modellektől a modern személetmódokig. Hangsúlyozza a szoftver fejlesztés modell szemléletű megközelítését. Részletesen foglalkozik a szabványos UML (Unified Modelling Language) jelölésrendszerrel, amelynek ismerete ma már nélkülözhetetlen egy fejlesztés részvevői közötti kommunikációban. Az alapvető munkafolyamatok közül elsősorban az analízis és tervezés lépéseivel foglalkozik, de rövid kitekintést ad a további tevékenységekről is.

A jegyzet az elméleti ismeretekből azokat a legszükségesebbeket foglalja össze, amelyek a helyes szemléletmód kialakításához szükségesek, alapvetően a gyakorlatias megközelítésre törekszik.

Az UML áttekintése során nem csak elszigetelt jelölési példákat mutat be, hanem igyekszik bemutatni azt, hogy egy adott probléma megoldása során az egyes UML eszközök hogyan képesek reprezentálni egy összefüggő modell elemeit.

Bár elsősorban a legalapvetőbb ismeretek összegyűjtésére törekszik, az érdeklődőbb hallgatók számára a törzsanyagon túlmutató információkat is tartalmaz.

A mesterszakos hallgatók is felhasználhatják a jegyzetet ismereteik felújítására, az esetleges hiányzó ismeretanyag pótlására, ezáltal könnyebben be tudnak kapcsolódni a témával foglalkozó mester szintű tárgy feldolgozásába.

1. A jegyzet használata

A jegyzet négy fő részre tagolódik.

1. Az alapvető elméleti háttér összefoglalása: 1.-6. fejezetek.
2. Az UML elemeinek ismertetése: 7.-10. fejezetek. (A 8. fejezetből csak az első három alpont.)
3. A fejlesztés modell szemléletű bemutatása: a 8. fejezet negyedik és ötödik alpontja, valamint a 11.-13. fejezetek.
4. Kiegészítő anyagok, az érdeklődőbb hallgatóknak: 14.-15. fejezet. Nem feltétlenül képezik részét egy BSc szintű tárgynak.

A Miskolci Egyetemen a Szoftver technológia tárgy számonkérése elméleti és gyakorlati részből áll. Az elméleti számonkérés félév végi beszámoló formájában és a számítástechnika szigorlat részeként történik. Ezekre a számonkérésekre az első két blokk alapján célszerű felkészülni. A gyakorlati számonkérés formája csoportmunkában megoldandó féléves feladat. Ennek a feladatnak az elkészítéséhez nyújt segítséget a jegyzet harmadik blokkja.

2. Történelem: a szoftver technológia kialakulása és fejlődése

A szoftver fejlesztés történetében az 1960-as évekig nyúlunk vissza. Ebben az időszakban már léteztek olyan számítógépek, amelyekkel nem csak tudományos kutatások és katonai alkalmazások igényeit lehetett kielégíteni, hanem a gyakorlatban felmerülő problémákat is meg lehetett segítségükkel oldani. A programokat speciális tudású kutatók készítették, akik eredeti szakmájuk (tipikusan villamosmérnök vagy matematikus) mellett képezték át magukat. A programok készítése lényegében a processzor instrukciókészletét leképező assembly nyelven történt. Minden program egyedi darab volt, és erősen kötődött ahhoz a számítógép modellhez, amire készült.

A 60-as évek végén következett be a későbbiekben „szoftver-krízis”-nek nevezett probléma. A hardver fejlesztések eredményeként megjelentek az úgynevezett harmadik generációs számítógépek, az előző modellekhez képest lényegesen hatékonyabban erőforrásokat nyújtva. Ezzel olyan problémák megoldására is alkalmassá váltak, amelyek egy nagyobb és bonyolultabb programok írását tették szükségessé. A korábban használt eszközök azonban erre egyre alkalmatlanabbnak bizonyultak. A növekvő fejlesztési igényeket nem lehetett időben kielégíteni, így a csökkenő hardver árakkal szemben egyre növekvő szoftver költségek jelentkeztek, az elkészült programok pedig nem megfelelő minőségűek lettek.

1968-ban ennek a problémának az elemzésére tartottak az érintett szakemberek egy konferenciát. Ekkor jelent meg először a „szoftvertervezés” fogalma. A konferencia legfontosabb megállapításai a következők voltak:

1. hatékonyabb programozási eszközök szükségesek,
2. a szoftverek fejlesztésére valamilyen módszeres megközelítést kell kifejleszteni az eddigi ad-hoc munkamódszerek mellett.

A szoftver technológia kialakulását ettől az időszaktól számíthatjuk, és fejlődését azóta is az határozza meg, hogy ez előbbi két kérdésre milyen választ tudunk adni.

Az 1970-es évek eredményei nagy vonalakban:

1. az első úgynevezett magas szintű programozási nyelvek (Algol, Fortran, Cobol) kialakulása,
2. a programozói munka szakmává válása, a csoportmunka igényének megjelenése,
3. az algoritmusok és adatszerkezetek terén folyó kutatások,
4. a legelső rendszeres programozási módszerek (strukturált, majd moduláris programozás) kidolgozása.

Az 1980-as évek legfontosabb eseményei:

1. a számítógépek teljesítményének növekedése és sorozatgyártásuk megindulása egyre több szervezet számára tette lehetővé alkalmazásukat a mindennapi működésük során.
2. Megjelentek az interaktivitást lehetővé tevő perifériák (terminálok)
3. Kialakultak az első számítógépes hálózatok
4. Nagy mennyiségű adatok változatos módon történő felhasználása vált szükségessé – megjelent az adatbázis fogalma
5. Strukturált programozás nyelvek használata (C, PASCAL)
6. és egy újdonság, ami később nagy jelentőségű lesz: 1981-ben jelent meg az első PC.

Ezek az események nyilvánvalóvá tették, hogy a szoftver előállítása mérnöki tevékenység, és méretei, bonyolultsága miatt csak csoportmunkában végezhető. Kialakultak a kezdeti (funkcionális szemléletű) módszertanok, a hatékonyabb munkát lehetővé tevő integrált fejlesztő környezetek és a kora CASE (Computer Aided Software Engineering) rendszerek.

Az 1990-es évektől a fejlődés minden területen felgyorsult. A hálózati technológiák rohamos terjedése (az Internet kialakulása) és a PC kategória tömegessé válása fokozatosan a vállalatok mellett a magánemberek számára is lehetővé tette a számítógépek használatát. Tovább bonyolítja a helyzetet a számítógépes, telekommunikációs és műsorszolgáltató hálózatok folyamatos közeledés és összefonódása. Erre a robbanásszerű fejlődésre az objektum orientált technológiák, a komponens szemlélet, a szabványok kialakulása és alkalmazása,

egyre összetettebb fejlesztő környezetek és a szoftver technológia mérnöki szemléletű módszerei próbálnak választ adni.

3. A szoftver fogalma, sajátosságai

A szoftver mára termékké vált, és ez alapvetően meghatározza az előállításának a körülményeit. A szoftver, mint termék nem egyszerűen számítógépes programot vagy programok halmazát jelenti. Egy szoftver rendszernek természetesen alapvető részét képezik a funkcionalitást megvalósító programok, de kiegészülnek a használatát lehetővé tevő dokumentációkkal, konfigurációs adatokkal, információs webhelyekkel.

A szoftver terméknek tekinthető, mert hasonlóan bármely más ipari termékhez, valamilyen hasznos célt kell kiszolgálnia, elő kell állítani, az előállítási folyamatnak költségei és időkorlátai vannak, ezért a felhasználó hajlandó fizetni érte, cserébe valamilyen minőséget vár el.

A szoftvernek azonban a más ipari termékektől eltérő tulajdonságai is vannak:

1. Nem anyagi jellegű. Létezik ugyan tárgyiasult formája, de az nem más, mint adathordozók és dokumentációk halmaza.

2. Nincsenek egyedi példányai. Egy szoftvert lemásolhatunk tetszőleges példányban, de ezek a másolatok teljes mértékben egyenértékűek az eredetivel. Ennek következménye, hogy előállítása nem igényli a tervezés – sorozatgyártás fázisokat.

3. Tükröznie kell a valóságot, de nem fizikai törvények vonatkoznak rá.

4. Bonyolultsága (komplexitása) a legtöbb ipari terméket messze meghaladja.

Ezeket a speciális tulajdonságokat kell figyelembe venni, ha a szoftver előállításának technológiáját vizsgáljuk.

4. A szoftver technológia definíciói

Az legegyszerűbb definíció valójában csak a „technológia” szót bontja ki:

A szoftver technológia eszközök és módszerek együttese a szoftver termékszerű előállítására.

A szakma egyik klasszikusa, Boehm, 1976-ban az alábbi definíciót fogalmazta meg:

„The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them.”

Magyarul: „Tudományos ismeretek gyakorlati alkalmazása számítógépes programok és a fejlesztésükhöz, használatukhoz és karbantartásukhoz szükséges dokumentációk tervezésében és előállításában.”

A definíció fontosságát az adja, hogy a megfogalmazza: a szoftver előállítása nem csak programok készítését, hanem dokumentációk előállítását is jelenti.

Az IEEE szervezet, amely számos szabvány jegyzője, 1983-ban az alábbi definíciót szabványosította:

„The technological and managerial discipline concerned with systematic production and maintenance of software products that are develop and modified on time and within cost estimated.”

Magyarul: „Technológiai és vezetési alapelvek, amelyek lehetővé teszik programok termékszerű gyártását és karbantartását a költség- és határidő korlátok betartásával.”

Ebben a definícióban jelenik meg az a gondolat, amely egyértelműen mérnöki tevékenységnek minősíti a szoftver előállítását: a gyártási folyamatot (általában szigorú) idő- és költség korlátok betartása mellett kell lebonyolítani.

5. Kihívások a szoftver fejlesztés során

A szoftverek előállítása során az egyik legnagyobb problémát a szoftver, mint rendszer komplexitása okozza. Idézzük ezzel kapcsolatban a szakmaterület két ismert személyiségének megállapításait:

Fred Brooks:

“The complexity of software is an essential property, not an accidental one.” (A software komplexitása annak lényegi és nem esetleges tulajdonsága.)

Grady Booch:

"The task of the software development team is to engineer the illusion of simplicity." (Egy software fejlesztő csapat dolga az egyszerűség illúziójának megtervezése.)

A komplexitás kezelése tehát az alapvető (technikai) probléma.

Nincs olyan tehetséges ember, aki egy akár átlagos méretű, de valódi problémát megoldó szoftvert teljes egészében át tudna tekinteni. Nem lenne megoldás az sem, hogyha fejlesztő csoportot bíznánk meg ezzel a feladattal, mert nem lehet egyesíteni a csoporttagok tudását.

Hogy mégis készülnek szoftverek, az azon a tényen alapul, hogy ha egy rendszert részrendszerekre bontunk fel, a részrendszerek komplexitásainak összege kisebb, mint a teljes rendszeré. A csoportmunkát pedig az teszi lehetővé, hogy a részrendszerekre bontással el lehet jutni olyan rendszer elemekig, amelyeknek a komplexitása már egy ember által is kezelhető. Ez a folyamatot dekompozíciónak nevezzük.

A dekompozíció tehát a teljes rendszer módszeres részekre bontása. A megfelelő szintű dekompozíció után az egyes alrendszerek implementálhatók. Végül a részrendszerek szintén módszeres integrálásával állítjuk elő a teljes rendszert.

A fejlesztési módszertanok (amelyekkel egy kicsit részletesebben a 4. fejezetben foglalkozunk) lényegében a dekompozíció és az integráció elveit és módszereit határozzák meg.

A mai körülmények között egy szoftverfejlesztő csapatnak további, nem technikai eredetű kihívásokkal is szembe kell nézniük:

1. Heterogenitás. A mai alkalmazások többsége a szerver csomópontoktól a mobil kliensekig számos, eltérő hardver – szoftver architektúrával rendelkező részegység szoros együttműködését igényli. Az egyes elemeknek ráadásul „cserélhetőeknek” kell lenniük (például ugyannak az alkalmazásnak működnie kell asztali és mobil klienssel is). Arra is fel kell készülni, hogy az alkalmazás életciklusa alatt új, az eredeti fejlesztés idején esetleg még nem is létező elemeket is integrálni kell.

2. A követelmények gyors és gyakori változása. Az üzleti élet szereplőinek folyamatosan alkalmazkodniuk kell a megváltozott körülményekhez. Mivel ma már a szervezetek üzleti folyamatai elképzelhetetlenek az informatikai rendszerek támogatása nélkül, sőt, számos üzleti folyamat valójában az informatikai rendszeren belül, automatikusan zajlik, az alkalmazkodás a szoftver rendszerek változtatását is igényli.

3. Szállítási kényszer. A szoftver rendszerek kifejlesztésére és változtatására egyre rövidebb idő áll rendelkezésre, éppen a követelmények gyakori változása miatt.

4. A meglévő rendszerek integrálása. A szervezetek nem minden folyamata változik gyakran. Az állandónak tekintető folyamatokat kiszolgáló alrendszerek általában már régen implementálva lettek, de lecserélésük túl költséges és kockázatos lenne. Meg kell tudni oldani az újonnan fejlesztett modulok és a régi, „örökölt” modulok integrálását.

A fenti kihívásoknak csak úgy tudunk megfelelni, ha mind az implementációs technológiákat, mind a szoftver fejlesztés folyamatát állandóan továbbfejlesztjük.

A szoftver technológia tehát – a többi mérnöki területhez képest – rövid múlttal rendelkezik, és folyamatos változásnak van kitéve.

2. fejezet - A szoftverfejlesztés életciklus modelljei

A szoftverfejlesztések számának és legfőképpen az egy-egy fejlesztéshez szükséges erőforrások volumenének növekedésével természetes módon jelent meg az igény a fejlesztési folyamat racionalizálására, ami többek között az ütemezési és pénzügyi tervezhetőséget, az eredmények megvalósíthatóságának illetve tényleges megvalósulásának kontrollját, és a biztonsági problémák tervezhetőségét érinti. Ezeknek megfelelően alakultak ki a különböző életciklus modellek, melyek célja a fejlesztési folyamat modellezése. A szoftverfejlesztés életciklusában megjelenő általános feladatok a következők:

1. Követelmények megfogalmazása - funkcionális specifikáció

2. Rendszertervezés (design) - rendszerterv

3. Kódolás, testreszabás és tesztelés

4. Bevezetés

A szoftverfolyamat modellje a szoftverfolyamat absztrakt reprezentációja. Minden egyes modell különböző speciális perspektívából reprezentál egy folyamatot, de ily módon csak részleges információval szolgálhat magáról a folyamatról. Ezek az általános modellek nem a szoftverfolyamat pontos, végleges leírásai, hanem valójában inkább hasznos absztrakciók, amelyet a szoftverfejlesztés különböző megközelítési módjainak megértéséhez használhatunk. Az irodalomban számos szoftverfejlesztési modell alakult ki az évek során és található meg, melyekből a legismertebb folyamatmodellek a következők:

1. A vízesésmodell: a folyamat alapvető tevékenységeit a folyamat különálló fázisainak tekinti. Ezek a fázisok a követelményspecifikáció, a szoftvertervezés, az implementáció, a tesztelés, integráció és a karbantartás.

2. Evolúciós vagy iteratív fejlesztés: ez a megközelítési mód összefésüli a specifikáció, a fejlesztés és a validáció tevékenységeit.

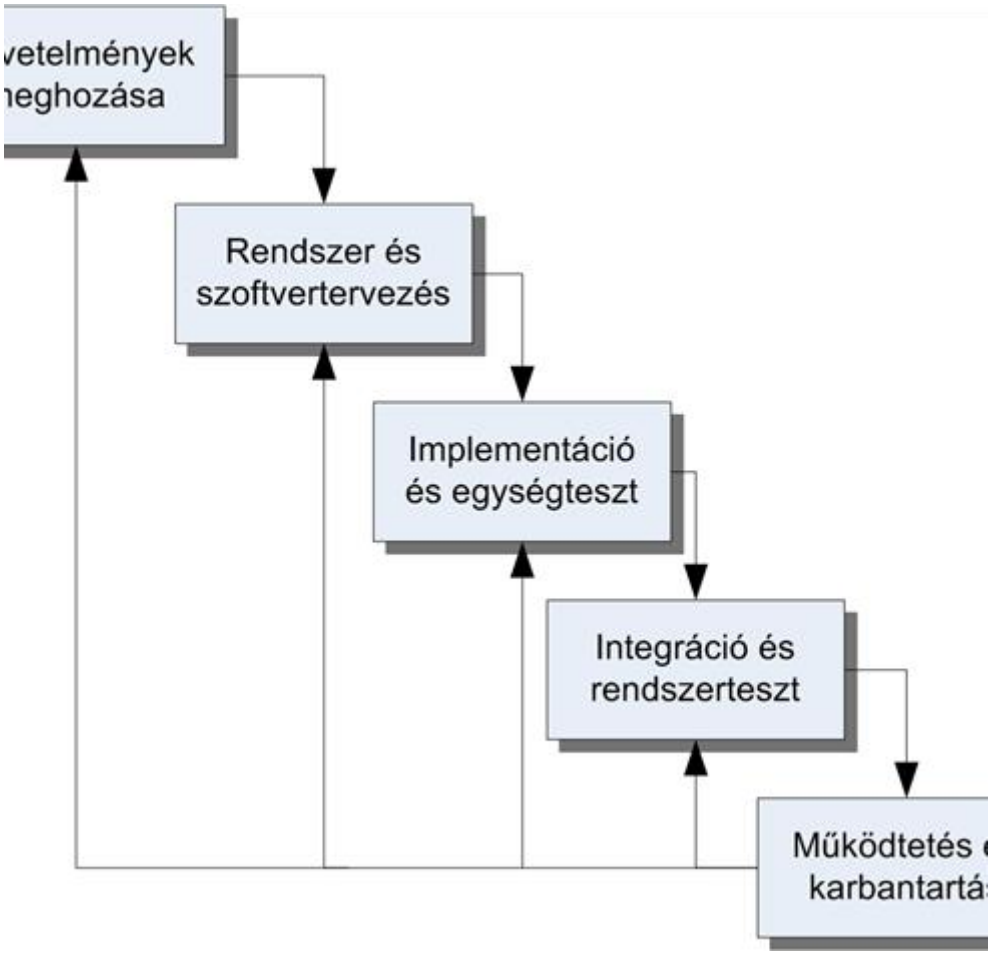
3. Komponens alapú fejlesztés: ez a megközelítés nagy mennyiségű újrafelhasználható komponensek létezésén alapszik.

A modellek használata azonban korántsem kizárólagos. Sőt, talán ki is jelenthetjük, hogy kevés olyan vállalat van, aki csupán egy modellt használ folyamatainak leírásához. A gyakorlatban inkább előszeretettel alkalmaznak több fejlesztési modellt egyszerre, főként nagy, komplex rendszerek fejlesztésekor. Ebben az esetben legfőbb cél mindig az, hogy az adott fejlesztési környezet mellett a modellek a környezetre vetített előnyős tulajdonságaikat emeljék ki, és adoptálják az adott környezetre.

1. A vízesésmodell

A szoftverfejlesztés történetének első publikált modellje (WaterfallModel, Royce1970), amely más tervezői modellekből származik. Az elnevezése onnan fakad, hogy a fejlesztésben felmerülő tevékenységeket jól elválasztható lépésekben, lépcsősen ábrázolja, ami alapján vízesésmodellként vált ismertté. Ezt illusztrálja az következő ábra.

2.1. ábra - A szoftver életciklusa



A modell alapvető szakaszai alapvető fejlesztési tevékenységekre képezhetők le. Ezek:

1. Követelmények elemzése és meghozása: a rendszer szolgáltatásai, megszorításai és célja a rendszer felhasználóival történő konzultáció alapján alakul ki. Ezeket később részletesen kifejtik, és ezek szolgáltatják a rendszer specifikációt.

1. Rendszer- és szoftvertervezés: a rendszer tervezési folyamatában választódnak szét a hardver- és szoftverkövetelmények. Itt kell kialakítani a rendszer átfogó architektúráját. A szoftver tervezése az alapvető szoftverrendszer-absztrakciók, illetve a közöttük levő kapcsolatok azonosítását és leírását is magában foglalja.

1. Implementáció és egységteszt: ebben a szakaszban a szoftverterv programok, illetve programegységek halmazaként realizálódik. Az egységteszt azt ellenőrzi, hogy minden egység megfelel-e a specifikációjának.

1. Integráció és rendszerteszt: megtörténik a különálló programegységek, illetve programok integrálása és teljes rendszerként való tesztelése, hogy a rendszer megfelel-e a követelményeknek. A tesztelés után a szoftverrendszer átadható az ügyfélnek.

1. Működtetés és karbantartás: általában ez a szoftver életciklusának leghosszabb fázisa. Megtörtént a rendszertelepítés és megtörtént a rendszer gyakorlati használatbavétele. A karbantartásba beletartozik az olyan hibák javítása, amelyekre nem derült fény az életciklus korábbi szakaszaiban, a rendszeregységek implementációjának továbbfejlesztése, valamint a rendszer szolgáltatásainak továbbfejlesztése a felmerülő új követelményeknek megfelelően.

A fázisok eredménye tulajdonképpen egy dokumentum. A modell fontos szabálya, hogy a következő fázis addig nem indulhat el, amíg az előző fázis be nem fejeződött. A gyakorlatban persze ezek a szakaszok kissé átfedhetik egymást. Maga a szoftverfolyamat nem egyszerű lineáris modell, hanem a fejlesztési tevékenységek iterációjának sorozata. Ez a vízesésmodellnél a visszacsatolásokban jelenik meg.

A dokumentumok előállításának költségéből adódóan az iterációk költségesek, és jelentős átdolgozást igényelnek. Ezért megszokott, hogy már kisszámú iteráció után is befagyasztják az adott fejlesztési fázist, és a fejlesztést későbbi fázisokkal folytatják. A problémák feloldását későbbre halasztják, kihagyják vagy kikerülik azokat. A követelmények ilyen idő előtti befagyasztása oda vezethet, hogy a rendszer nem azt fogja tenni, mint amit a felhasználó akart.

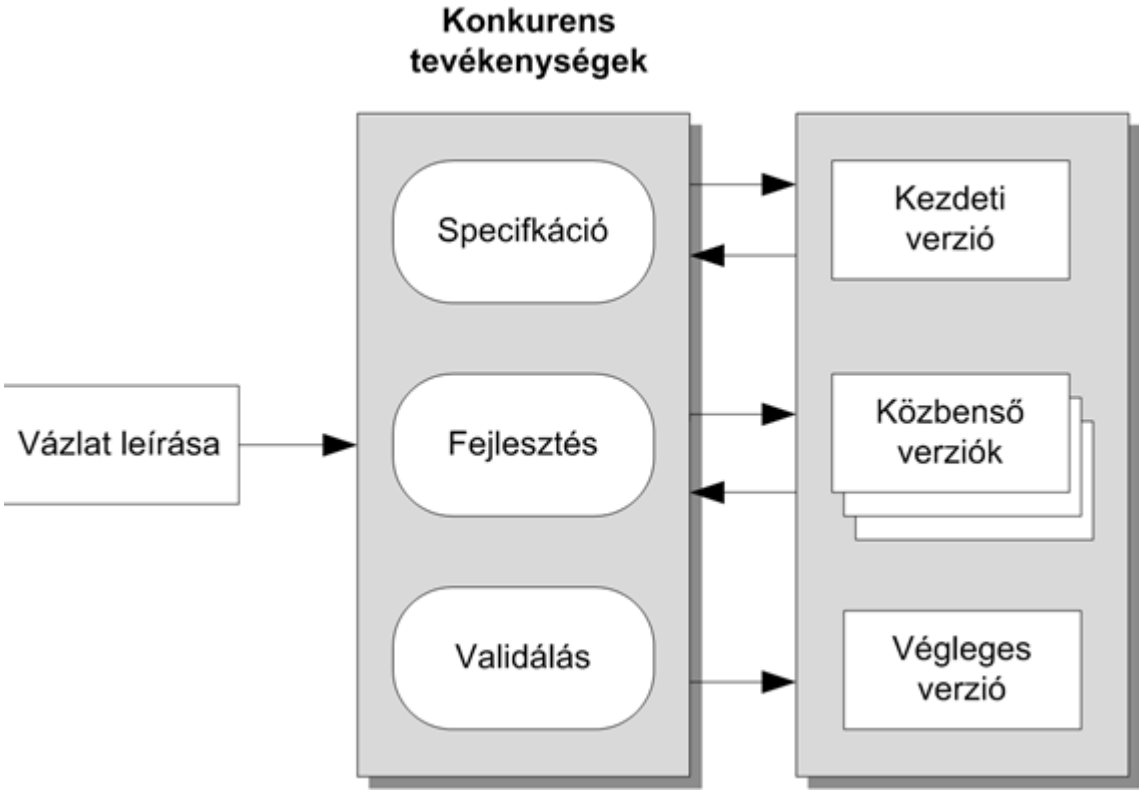
Az életciklus utolsó szakaszában a felhasználók használatba veszik a szoftvert. Ilyenkor derülnek ki az eredeti rendszerkövetelmények hibái és hiányosságai. Program és tervezési hibák kerülnek elő, továbbá felmerülhet új funkciók szükségessége is. Ezekből adódóan a rendszert át kell alakítani, amely néhány vagy akár az összes korábbi folyamatszakasz megismétlésével is járhat.

A vízesésmodell legfőbb problémáját a projekt szakaszainak különálló részekké történő nem flexibilis partícionálása okozza. Egy komplex, bonyolult probléma megoldása nem végezhető el hatékonyan ezzel a megközelítéssel. A vízesésmodell csak akkor használható jól, ha már előre jól ismerjük a követelményeket, melyeket részletes és pontos specifikáció követ.

2. Evolúciós fejlesztés

Az evolúciós fejlesztés a vízesésmodelltől eltérő alapgondolaton alapul. Az alapötlete az, hogy a fejlesztőcsapat kifejleszt egy kezdeti implementációt, majd azt a felhasználókkal véleményezteti, majd sok-sok verzión keresztül addig finomítják, amíg a megfelelő rendszert el nem érik. A szétválasztott specifikációs, fejlesztési és validációs tevékenységekhez képest ez a megközelítési mód sokkal inkább érvényesíti a tevékenységek közötti párhuzamosságot és a gyors visszacsatolásokat.

2.2. ábra - Evolúciós fejlesztési modell



Az evolúciós fejlesztésnek két különböző típusát szokás a gyakorlatban megkülönböztetni:

1. Feltáró fejlesztés: célja egy működőképes rendszer átadása a végfelhasználóknak. Ezért elsősorban a legjobban megértett és előtérbe helyezett követelményekkel kezdik a fejlesztés menetét. Ennek érdekében a megrendelővel együtt tárjuk fel a követelményeket, és alakítják ki a végleges rendszert, amely úgy alakul ki, hogy egyre több, az ügyfél által kért tulajdonságot társítunk a már meglévőkhöz. A kevésbé fontos és körvonalazatlanabb követelmények akkor kerülnek megvalósításra, amikor a felhasználók kérik.

1. Eldobható prototípus készítés: a fejlesztés célja ekkor az, hogy a lehető legjobban megértsük az ügyfél követelményeit, amelyekre alapozva pontosan definiáljuk azokat. A prototípusnak pedig azon részekre kell koncentrálni, amelyek kevésbé érthetők.

Próbáljuk meg összevetni az evolúciós megközelítést a vízesésmodellel. A fentiek alapján láttuk, hogy a vízesésmodell kevésbé rugalmas a menetközben szükséges változásokra, így érvelhetünk azzal, hogy az evolúciós megközelítés hatékonyabb a vízesésmodellnél, ha olyan rendszert kell fejleszteni, amely közvetlenül megfelel az ügyfél kívánságainak. További előnye, hogy a rendszerspecifikáció inkrementálisan fejleszthető. Mindezek ellenére a vezetőség szemszögéből két probléma merülhet fel:

1. A folyamat nem látható: a menedzsereknek rendszeresen szükségük van leszállítható részeredményekre, hogy mérhessék a fejlődést.

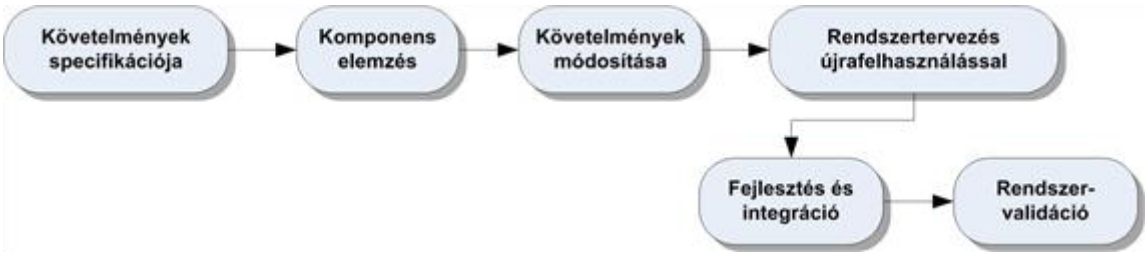
2. A rendszerek gyakran szegényesen strukturáltak: a folyamatos változtatások lerontják a rendszer struktúráját, így kevésbé érthetővé válik. A szoftver változásainak összevonása pedig egyre nehezebbé és költségesebbé válhat.

Felmerülhet akkor a kérdés, hogy mikor és kinek érdemes használni az evolúciós fejlesztési modellt? Nos a válasz természetesen nem lehet egyértelmű, de a gyakorlati tapasztalatok alapján a várhatóan rövid élettartalmú kis vagy közepes rendszerek esetén célszerű az alkalmazása. Körülbelül 500.000 programsorig terjedően. Ugyanis nagy, hosszú élettartalmú rendszerek esetén az evolúciós fejlesztés válságossá válhat pontosan az evolúciós jellege miatt.

3. Komponens alapú fejlesztés

A komponens alapú fejlesztés alapgondolata az, mint ahogy az elnevezés is utal rá, ahogy próbáljuk meg az elkészítendő szoftvert újrafelhasználható komponensekből felépíteni. Erre az ad okot, hogy a szoftverfolyamatok többségében megtalálható valamelyest a szoftverek újrafelhasználása. Ilyen esetekben előkeresik a korábbi kódot (komponenst) és újra átdolgozva, esetleg általánosítva beledolgozzák a rendszerbe.

2.3. ábra - Újrafelhasználás orientált modell



Az újrafelhasználás-orientált megközelítési mód nagymértékben az elérhető újrafelhasználható szoftverkomponensekre, illetve azok egységes szerkezetbe történő integrációjára támaszkodik. Néha ezek a komponensek saját létjogosultsággal rendelkeznek. Amíg a kezdeti követelményspecifikációs és validációs szakasz összehasonlítható más folyamatokkal, addig a közöttük található szakaszok az újrafelhasználás-orientált fejlesztésekben különböznek. Ezen szakaszok a következők:

1. Komponenselemzés: az adott követelményspecifikációnak megfelelően megkeressük azon komponenseket, amelyek megvalósítják azok funkcióit, implementálták azokat. A legtöbb esetben nincs egzakt illeszkedés, a felhasznált komponens a funkciók csak egy részét nyújtja.

1. Követelménymódosítás: a fázisban a megtalált komponensek információit felhasználva elemezzük a követelményeket. Majd módosítani kell azokat az elérhető komponenseknek megfelelően. Ahol nem lehetséges a követelmény módosítása, ott újra el kell végezni a komponenselemzést és alternatív megoldást kell keresni.

1. Rendszertervezés újrafelhasználással: ebben a szakaszban a rendszer szerkezetének tervezését hajtjuk végre. A tervezés kulcseleme az, hogy milyen komponenseket akarunk újrafelhasználni, és úgy alakítani a szerkezetet, hogy azok működhessenek. Amennyiben nincs elérhető újrafelhasználható komponens, akkor új szoftverek is kifejleszthetők.

1. Fejlesztés és integráció: a nem megvásárolt, illetve átalakításra kerülő komponenseket ki kell fejleszteni és a rendszerbe integrálni. A rendszer-integráció ebben a modellben sokkal inkább tekinthető a fejlesztési folyamat részének, mint különálló tevékenységnek.

A fejlesztés komponens alapokra való helyezése mind előnyökkel mind hátrányokkal is jár. Előnyként említhető, hogy csökkenti a kifejlesztendő szoftverek számát a komponensek újrafelhasználásával, ezzel pedig közvetve a költségeket redukálja, illetve felmerülő kockázati tényezőket. Sok esetben a rendszer így gyorsabban is leszállítható a megrendelőnek. Hátrányként azonban felmerül, hogy a követelményeknél elkerülhetetlenek a kompromisszumok. Mindezek oda vezethetnek, hogy az elkészült rendszer nem felel meg a megrendelő valódi kívánságainak.

4. A folyamatiteráció

Már a legelső szoftverek készítésének gyakorlati tapasztalatai alapján hamar felmerült, hogy magát a szoftverfolyamatot nem célszerű mindig egy egyszerű lineáris folyamatként értelmezni, hanem sokkal inkább a folyamattevékenységek rendszeresen ismétlődő folyamataként. Ez azt jelenti, hogy ciklikus ismétlődések – nevezhetjük iterációknak a továbbiakban – során a rendszert mindig átdolgozzuk az igényelt változások szerint. Ezen megközelítés oka, hogy a nagy rendszerek esetében elkerülhetetlenek a fejlesztés során a változtatások. Változhatnak a követelmények, az üzletmenet, és a külső behatások.

A folyamatiteráció támogatására több modell is kidolgozásra került. A két legismertebbet említve:

1. Inkrementális fejlesztés: a szoftverspecifikáció, a tervezés, az implementálás, kis inkrementációs lépésekre van felosztva.

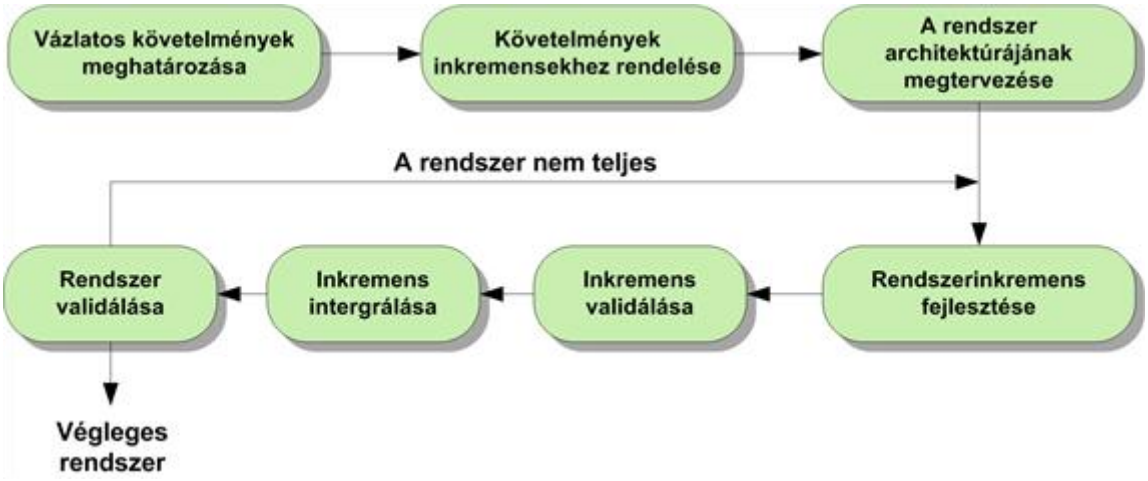
2. Spirális fejlesztés: a fejlesztési folyamat egy belülről kifelé tartó spirálvonalat követ.

Az iteratív folyamat lényege, hogy a specifikációt a szoftverrel összekapcsolva kell fejleszteni, nem pedig előre elkészíteni az egész dokumentumot. Ezáltal a fejlesztés sokkal rugalmasabban és könnyebben tud reagálni a menetközben történt változások követésére.

4.1. Inkrementális fejlesztés

Az inkrementális megközelítési mód egy köztes megközelítés a vízesésmodell és az evolúciós fejlesztési modellek között. A vízesésmodell megköveteli az ügyféltől, hogy véglegesítse a követelményeket mielőtt a tervezés elindulna, a tervezőtől pedig azt, hogy válasszon ki bizonyos tervezési stratégiákat az implementáció előtt. A vízesésmodell előnye, hogy egyszerűen menedzselhető, mert külön választja az egyes fázisokat. Ezzel szemben viszont olyan robosztus rendszerek jöhetnek létre, amik esetleg alkalmatlanok a változtatásokra. Az evolúciós megközelítésnél pedig megengedettek a követelményekkel és tervezésekkel kapcsolatos döntések elhagyása, ami pedig gyengén strukturált és nehezen megérthető rendszerekhez vezethetnek. A módszer lépései a következő ábrán figyelhetők meg:

2.4. ábra - Az inkrementális fejlesztés folyamata



1. nagy körvonalakban a rendszer által nyújtandó szolgáltatásokat,

2. mely szolgáltatások fontosabban, melyek kevésbé.

A követelmények meghatározása után a követelmények inkremensekben való megfogalmazása és hozzárendelése következik. A szolgáltatások inkremensekben való elhelyezése függ a szolgáltatás prioritásától is. A magasabb prioritású szolgáltatásokat hamarabb kell biztosítani a megrendelő felé.

Miután az inkrementációs lépéseket meghatároztuk, az első inkrementációs lépés által előállítandó szolgáltatások követelményeit részletesen definiálni kell. Ezután pedig következik az inkremens kifejlesztése. A fejlesztés ideje alatt sor kerülhet további követelmények elemzésére, de az aktuális inkrementációs lépés követelményei nem változtathatók.

Amennyiben egy inkremens elkészült, a rendszer bizonyos funkcióit akár be is üzemeltethetik korábban. Ez több szempontból is fontos lehet. Egyrészt tapasztalatokat szerezhetnek a rendszerrel kapcsolatban, amely a későbbi inkrementációs lépésekben segítségre lehet a követelmények tisztázásában, másrészt bizonyos kész vagy félkész funkciók akár a megrendelőnek is leszállíthatók bemutatási, tesztelési célokra.

Amikor az új inkremens elkészült, integrálni kell a már meglévő inkremensekkel. A rendszerfunkciók köre ezzel egyre bővül, míg végül elkészül a teljes rendszer. Az inkrementációs fejlesztés előnyei röviden a következő néhány pontban lehetne összefoglalni:

1. A megrendelőnek nem kell megvárnia míg a teljes rendszer elkészül. Már az első inkremens kielégítheti a legkritikusabb követelményeket, így a szoftver már menet közben használhatóvá válik.

2. A megrendelők használhatják a korábbi inkremenseket mint prototípusokat, ami által tapasztalatokat szerezhetnek.

3. Kisebb a kockázata annak, hogy a teljes projekt kudarcba fullad.

4. A fejlesztés során a magasabb prioritású inkremenseket szállítjuk le hamarabb, ezért mindig a legfontosabb szolgáltatások lesznek többet tesztelve. Ezért kisebb a hiba esélye a rendszer legfontosabb részeiben.

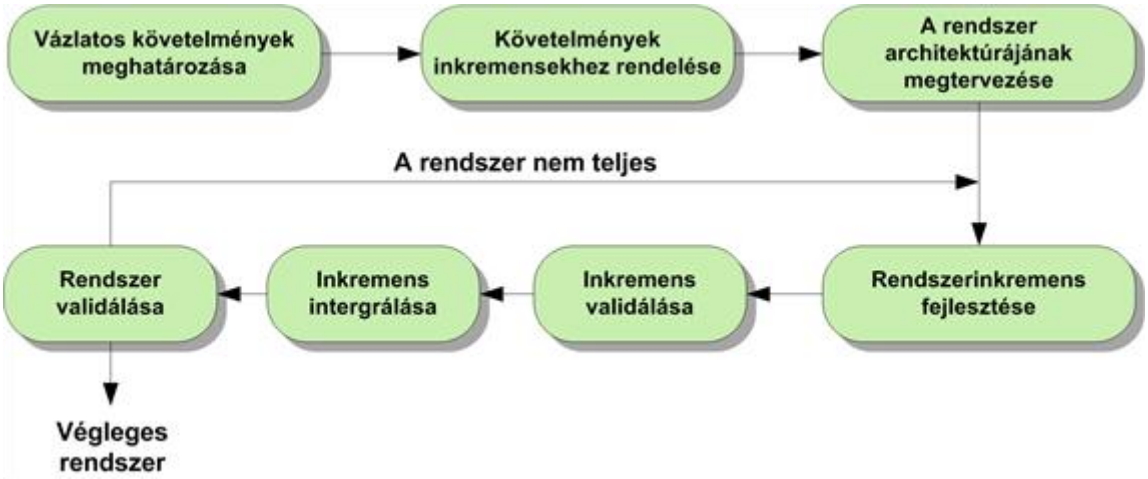
Mindezek ellenére az inkrementális fejlesztésnek is megvannak a maga hibái. Fontos, hogy az inkremensek megfelelően kis méretűk legyenek és minden inkrementációs lépésnek szolgáltatni kell valami rendszerfunkciót. Sokszor azonban nehéz a megrendelő követelményeit megfelelő méretű inkrementációs lépésekre bontani.

4.2. Spirális fejlesztési modell

A spirális fejlesztési modellt Boehm javasolta először már 1986-ban a „A Spiral Model of Software Development and Enhancement” címmel megjelent publikációjában, amely azóta széles körben elterjedt az irodalomban és a gyakorlatban is. A modell alapgondolata szintén eltér az eddigiektől, mert a szoftverfolyamatot nem tevékenységek és közöttük található esetleg visszalépések sorozataként tekinti, hanem inkább egy spirálként reprezentálja. A spirál minden egyes körben a szoftverfolyamat egy-egy fázisát reprezentálja. A spirálnak, mint a folyamatot reprezentáló vonalnak egy további sugallnivalója is van. Mégpedig az, hogy tetszőleges számú kör, mint iteráció tehető meg.

A legbelső kör a megvalósíthatósággal foglalkozik, a következő a rendszer követelményeinek meghatározása, a következő kör pedig a rendszer tervezésével foglalkozik, és így tovább.

2.5. ábra - Boehm féle spirálmodell (forrás: [1])



A spirál minden egyes ciklusát négy fő szektorra oszthatjuk fel:

1. Célok kijelölése: az adott projektfázis által kitűzött célok meghatározása. Azonosítani kell a folyamat megszorításait, a terméket, fel kell vázolni a kapcsolódó menedzselési tervet. Fel kell ismerni a projekt kockázati tényezőit, és azoktól függően alternatív stratégiákat kell tervezni ha lehetséges.

2. Kockázat becslése: minden egyes felismert kockázati tényező esetén részletes elemzésre kerül sor. Lépéseket kell tenni a kockázat csökkentése érdekében.

3. Fejlesztés és validálás: a kockázat kiértékelése után egy fejlesztési modellt kell választani a problémának megfelelően. Pl. evolúciós, vízesés, stb modellek.

4. Tervezés: A folyamat azon fázisa, amikor dönteni kell arról, hogy folytatódjon-e egy következő ciklussal, vagy sem. Ha a folytatás mellett döntünk, akkor fel kell vázolni a projekt következő fázisát.

Egy spirálciklus mindig a célok meghatározásával kezdődik. Ekkor fel kell sorolni a megvalósítás lehetőségeit, és meg kell vizsgálni azok megszorításait is. Minden egyes célhoz meg kell határozni egy lehetséges alternatívát, amely azt eredményezi, hogy azonosításra kerülnek a projekt kockázati forrásai is. A következő lépés ezeknek a kockázatoknak a kiértékelése, majd végül pedig a tervezési fázis következik, ahol eldönthetjük, hogy szükség van-e egy további ciklusra vagy sem.

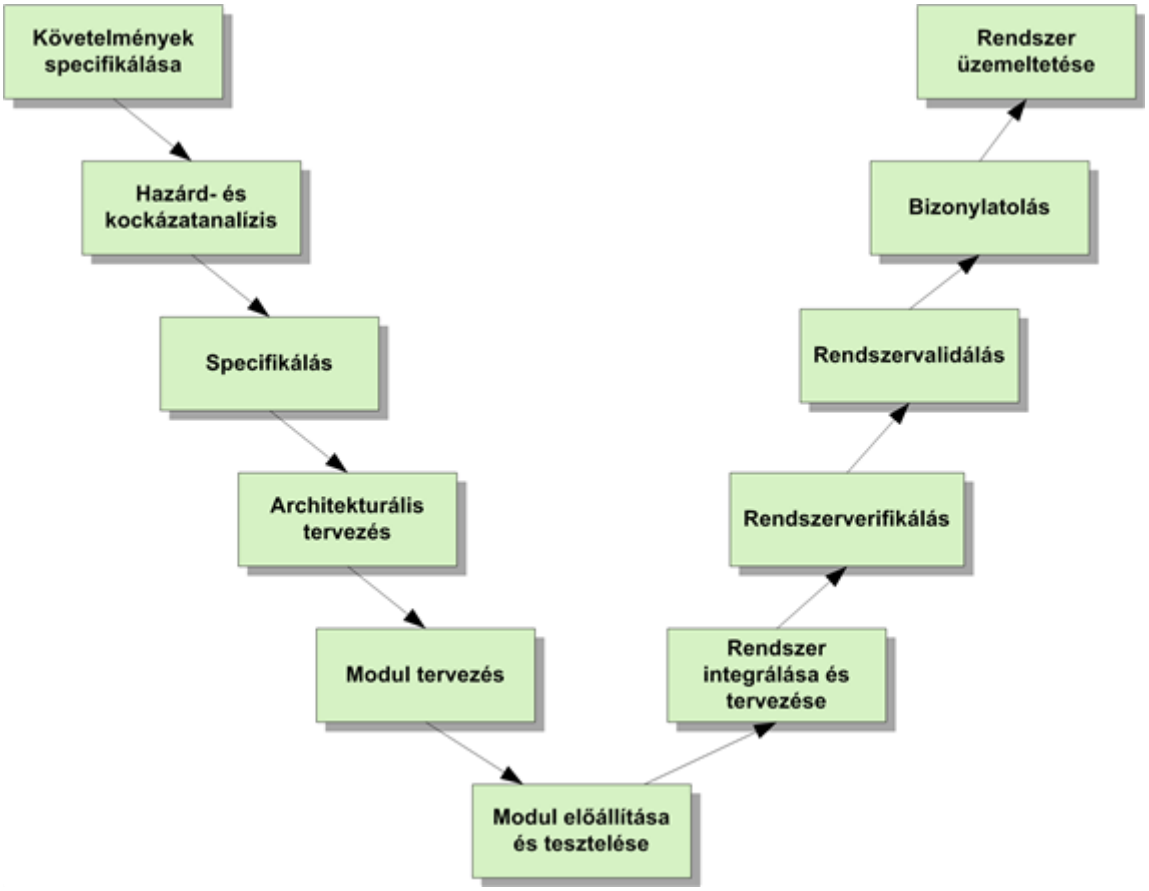
Miben más a spirális fejlesztési modell az egyéb szoftverfolyamat-modelltől? Itt a modell explicite számol a kockázati tényezőkkel, amelyek problémákat okozhatnak a projektben. Ilyen például a határidő- és költségtúllépések. A modell egy tipikus alkalmazási területe napjainkban a játékfejlesztés iparága, mégpedig azért, mert a mai vezető számítógépes játékok nagyon komplex, sok embert igénylő szoftverfejlesztési folyamat, ahol gyakran kell határidőcsúszásokkal is számolni.

5. A V-modell

A V-modell szintén a korai modellek családjába tartozik, melyet a német védelmi minisztérium fejlesztett ki, majd főleg a német hadsereg szoftverfejlesztéseiben vált használatossá a továbbiakban. Maga az elnevezés nemcsak életciklus modellt, hanem egy teljes módszertant jelöl, aminek több elemét az ISO 12207 szabvány is átvette. A V-modell életciklus elképzelése nemcsak az egyes fázisok időbeli sorrendjéről szól, hanem arról is, hogy az egyes fázisokban mely korábbi fázisok termékeit kell felhasználni; illetve az adott fázistevékenységét és termékét mely korábbi fázisban leírt követelmények, illetve elkészített tervek alapján kell ellenőrizni.

A V-modell használata főleg a biztonságkritikus számítógéprendszerek fejlesztése esetében a terjedt el. A következő ábrán a modell fázisait mutatjuk be, ahol az egyes fázisok V alakú elrendezésben követik egymást.

2.6. ábra - A V modell



A modellábrája a fejlesztés két folyamat két megközelítését tükrözi. Top-down megközelítésként kifejezi a tervezési folyamat fentről lefelé történő haladását a diagram baloldali ágában, míg a tesztelési folyamat lentről felfelé halad bottom-up megközelítésben a jobboldali ágban. Az ilyen ábrázolás csak megközelítőleg írja le a fejlesztést. A gyakorlatban a különböző fázisok nem szigorúan a megadott sorrendben hajtódnak végre. A tervezés gyakran nagyszámú iterációt foglal magában, olyan műveletek sorával, amelyeket addig kell ismételni, amíg kielégítő eredményre nem jutunk. Párhuzamosságok ugyancsak lehetségesek.

Vizsgáljuk meg az egyes fejlesztési állomásokat a V-modellben:

1. Követelmények specifikálása: a fejlesztési folyamat kiindulási pontját képező követelmények feltárása, elemzése majd specifikációja történik. A fázis eredménye egy dokumentum, amely részletes információt tartalmaz a rendszer szolgáltatásairól és megszorításairól.

1. Hazárdok és kockázatok elemzése: célja a lehetséges veszélyhelyzetek meghatározása a rendszerben, a megelőző kiszűrés érdekében. Az analízisek elvégzéséhez különféle módszerek állnak rendelkezésre. A kockázatok elemzésére valószínűségszámítási segédeszközöket alkalmaznak leginkább, aminek eredményeként számszerű értékeket rendelnek az egyes veszélyes következményekhez. Az elemzési folyamatok eredményeként létrehozandó a biztonsági követelmények dokumentációja. Ez a dokumentum arra ad előírást, hogy mit kell betartani a rendszernél, mit szabad, ill. mit nem szabad megengedni a rendszer működése során.

2. Teljes rendszer-specifikáció: a funkcionális követelmények valamint a biztonsági követelmények együttese alkotja. Mindezen specifikáció alapján megkezdhető a teljes rendszer konkrét tervezési folyamata.

1. Architekturális tervezés: a teljes informatikai rendszer hardver és szoftver architektúrájának megtervezése. A tervezésnek ebben a fázisában azt kell eldönteni, hogy mely funkciók legyenek megvalósítva hardver, és melyek szoftver által.

2. A szoftver modulokra bontása: a fázisban a fejlesztési folyamatot további kisebb részekre, úgynevezett modulokra bontjuk fel a tervezési folyamat egyszerűsítése, áttekinthetőbbé tétele végett. A tervezés

eredményeként a szoftver modulok specifikációja, valamint a köztük levő kapcsolódási folyamatok terve készül el.

3. A modulok elkészítése és tesztelése: a szakaszban egyes modulok teljes implementációja kell megvalósuljon, majd ezt követően pedig az elkészült modulok önálló tesztelése. Célszerű a tesztelési folyamatokat szintén előzetesen megtervezni. A tesztelés már szerves része a szoftver verifikációs folyamatának, amelyben azt döntjük el, hogy egy-egy modul megfelel-e a specifikációjának.

4. Rendszerintegráció: a fázisban az elkészült szoftver-modulok integrálása történik egy teljes rendszerré miután mindegyik modul átment a tesztelésen.

5. Rendszerverifikáció: a fázis feladatta annak az eldöntése, hogy rendszer megfelel-e a specifikációjának, funkcionálisan teljesíti-e az összes specifikációs pontot.

6. Rendszervalidáció: el kell dönteni, hogy a teljes rendszer megfelel-e minden további nem funkcionális követelmények. Ebbe beletartozik a biztonsági feltételek teljesítésének eldöntése is: az ún. biztonság-igazolás.

7. Bizonylatolás (certification): a hatósági előírások és szabványok szerinti megfelelés eldöntése, és az erre vonatkozó bizonylatok kiállítása.

8. A rendszer üzemeltetése: üzembe helyezés, üzemeltetés, karbantartás, elavulás, üzemeltetés megszüntetése.

A modellt mind a mai napig alkalmazzák, főleg Német területeken. Bár a bemutatott első modell is kiválóan alkalmas a fejlesztési folyamatok szervezésére, számos új kiegészítés és átalakítás született a későbbiekben a kisebb hiányosságok pótlására, vagy egy-egy részterületen való alkalmazásának specifikálására.

6. RUP folyamatmodell

A Rational Unified Process (RUP) jó példája a modern folyamatmodelleknek, melyek az UML-ből és a hozzá kapcsolódó Unified Software Development Process-ből származnak. A szoftverfejlesztési modellt a Rational Software Corporation fejlesztett ki, melyek később az IBM is felvásárolt. A RUP már a szemléletmód alapjaiban szeretne különbözni a közismert folyamatmodellektől. Felismeri, hogy a konvencionális folyamatmodellek a folyamatoknak csak egy egyszerű nézetét adják, ezért erősen érződik a felépítés hibrid jellege, ugyanis mindegyik korábban tárgyalt általános folyamatmodellből tartalmaz elemeket. Támogatja az iterációt, és jól illusztrálja a specifikáció és a tervezés tevékenységeit. A RUP nem egy kész, követendő eljárást ad minden projektre, sokkal inkább egy könnyen változtatható keretet azok kézbentartásához.

Célja az olyan termékek előállításának biztosítása, melyek rendelkeznek elég magas minőségi színvonallal és a kívánt szolgáltatásokkal, miközben az időbeli ütemezés és a költségvetés határai nem lettek megsértve. Ennek a folyamatnak négy szerepe van:

Ezért az RUP a rendszerfejlesztés folyamatát alapvetően három dimenzióval írja le:

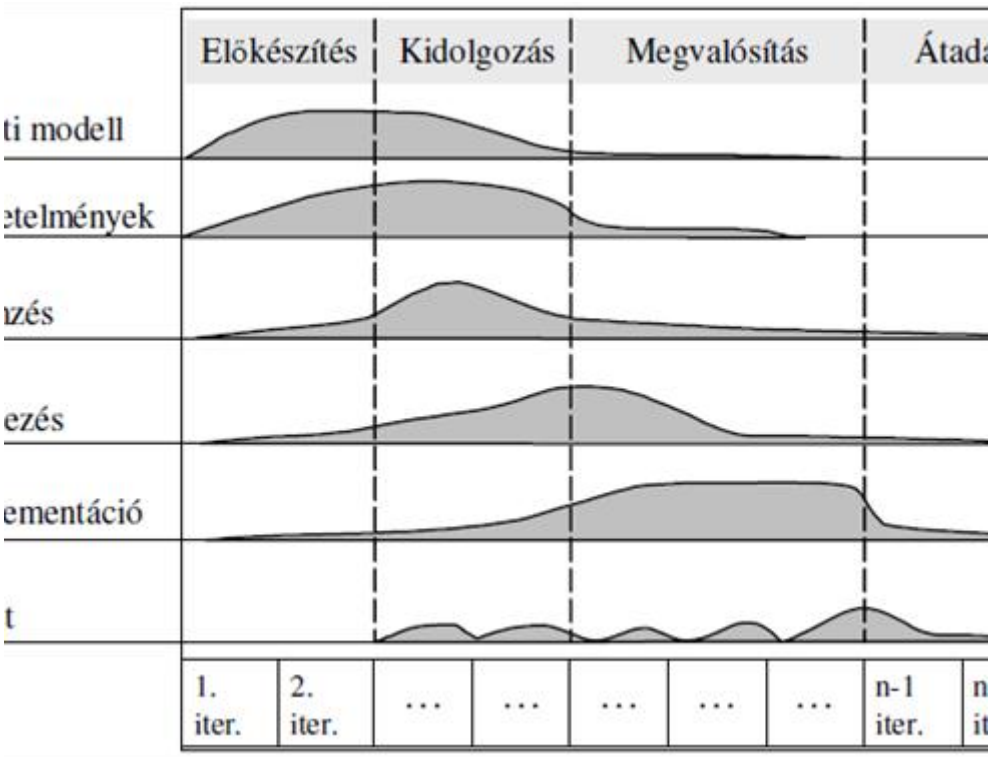
1. Dinamikus perspektívával, amely a modell fázisait mutatja.

2. Statikus perspektívával, amely a végrehajtandó folyamattevékenységeket mutatja.

3. Gyakorlati perspektívával, amely jól hasznosítható gyakorlatokat javasol a folyamat alatt.

Az időbeliség alapján az RUP a rendszerfejlesztést négy nagyobb egységre, négy diszkrét fázisrabontja. Ezek a kijelölt fázisok sokkal közelebb állnak az üzleti vonatkozásokhoz, mint a technikaiakhoz. A következő ábra bemutatja ezeket:

2.7. ábra - Fázisok és munkafolyamatok



6.1. RUP rendszerfejlesztési fázisok

A következőkben bemutatjuk a modellre jellemző rendszerfejlesztési fázisokat.

Előkészítés

Az Előkészítés (inception) fázisában a rendszer eredeti ötletét olyan részletes elképzeléssé dolgozzuk át, mely alapján a fejlesztés tervezhető lesz, a költségei pedig megbecsülhetők. Ebben a fázisban megfogalmazzuk, hogy a felhasználók milyen módon fogják használni a rendszert és hogy annak milyen alapvetőbelső szerkezetet, architektúrát alakítunk ki.

Kidolgozás

A Kidolgozás (elaboration) fázisában a használati módokat, a „használati eseteket” részleteiben is kidolgozzuk, valamint össze kell állítanunk egy stabil alaparchitektúrát (architecturebaseline). A Unified Process készítőinek a képe alapján a teljes rendszer egy testnek tekinthető, csontváznak, bőrnek és izmoknak. Az alaparchitektúra ebből a bőrrel borított csontváz, mely mindössze a minimális összekötő izomzatot tartalmazza, annyit, amennyi a legalapvetőbb mozdulatokhoz elegendő. Az alap-architektúra segítségével a teljes fejlesztés folyamata ütemezhető és a költségei is tisztázhatók.

Megvalósítás

A Megvalósítás (construction) során a teljes rendszert kifejlesztjük, beépítjük az összes „izomzatot”. Legfőképpen a rendszertervvel, a programozással és a teszteléssel foglalkozik. A rendszer különböző részei párhuzamosan fejleszthetők, majd ez alatt a fázis alatt integrálhatók. A fázis teljesítése után már rendelkezünk egy működő szoftverrendszerrel és a hozzá csatlakozó dokumentációval, amely készen áll, hogy leszállítsuk a felhasználónak.

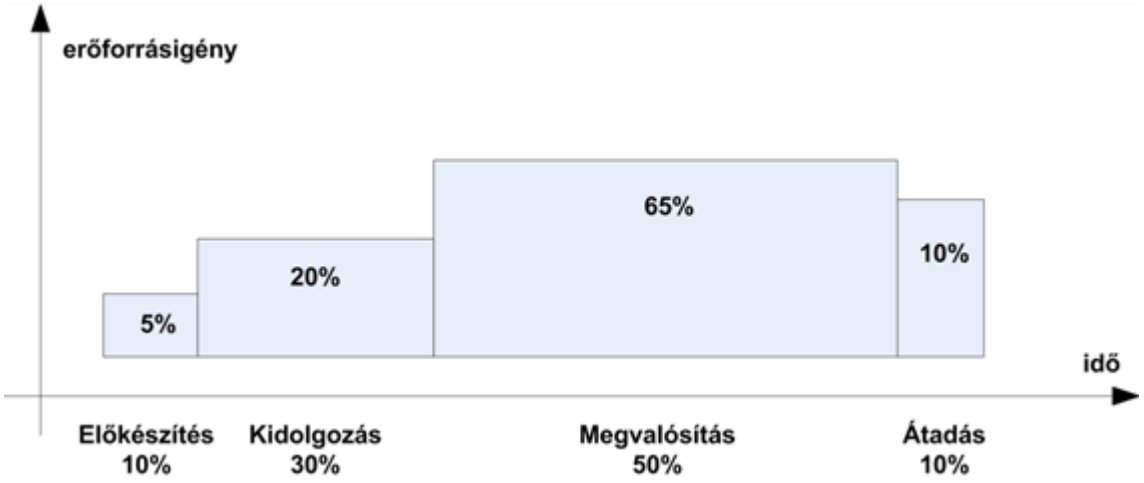
Átadás

Az Átadás (transition) a rendszer bétaváltozatának kipróbálását jelenti, mely során néhány gyakorlott felhasználó teszteli a rendszert és jelentést készít annak helyességéről vagy a hibáiról és hiányosságairól. A megjelenő hibákat ki kell küszöbölni, a még hiányzó részeket ki kell fejleszteni.

6.2. Mérföldkövek

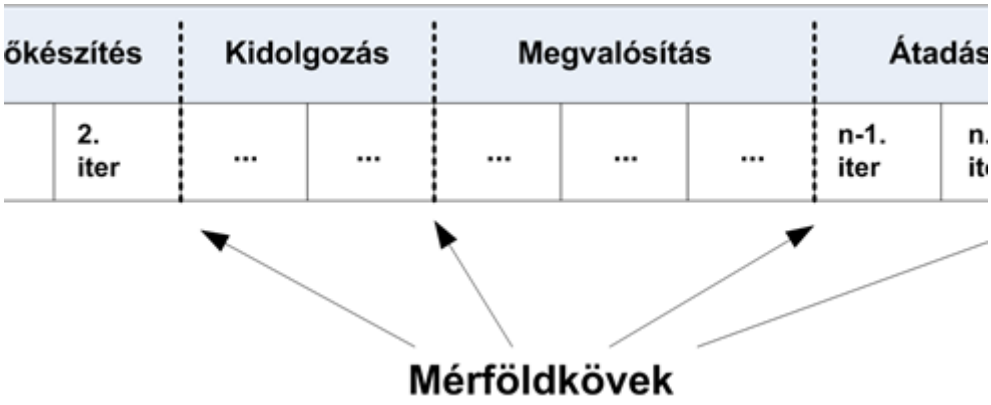
Minden fázis vége a fejlesztés egy-egy jól meghatározott mérföldkövét (milestone) jelenti, azaz olyan pontot, ahol egy célt kell elérnünk, illetve ahol kritikus döntéseket kell meghozni. Minden fázis végén megvizsgáljuk az eredményeket és döntünk a folytatásról.

2.8. ábra - Fázisok idő- és erőforrásigénye



A fejlesztés nagyobb egységeit jelentő fázisok további kisebb egységekre, iterációkra (iteration) bonthatók. Minden iteráció egy teljes, illetve részben önálló fejlesztési ciklust jelent, mivel az iteráció végén egy működő és végrehajtható alkalmazásnak kell előállnia. Minden iteráció végén így a végső, teljes rendszer egyre bővülő részét kapjuk eredményül, melyeket a rendszer egymás utáni kibocsátásainak (release), vagy belső változatainak nevezünk. A belső változatok lehetővé teszik, hogy azt a fejlesztők kipróbálhassák és annak tapasztalatai alapján esetleg módosíthassák a fejlesztés ütemezését.

2.9. ábra - Mérföldkövek



6.3. A fejlesztés további dimenziója

A Unified Process felbontását követve a fejlesztés menetének másik dimenziója az eljárás elemeit határozza meg. Ezen elemek azt szimbolizálják, hogy a fejlesztés során milyen dokumentumokat, diagramokat, forráskódokat – összefoglaló néven produktumokat (artifact) – készítsünk el. Az elkészítendő produktumok természetesen a megfelelő tevékenységekkel állíthatók össze, mely tevékenységeket pedig adott szakismerettel rendelkező személyek („dolgozók”), adott sorrendben hajthatnak végre. A tevékenységek, azok időbeli sorrendje és az azt végrehajtó dolgozók együttesen egy munkafolyamattal (workflow) írhatók le. A RUP gyakorlatilag az UML-lel együtt került kifejlesztésre, így a munkafolyamatok leírása UML-modellekkel történik.

A fejlesztéshez kezdetben mindig egy megfelelő kiindulópontot kell keresni. Az első tevékenységcsoport így az üzleti modellezés (business model) lesz, mely során megkeressük a készítendő rendszer üzleti vagy más néven szakterületi környezetét, mely alapvetően az üzleti fogalmakat és folyamatokat jelentik, illetve az azokra hatást gyakorló üzleti munkatársakat.

Második fontos tevékenység a követelmények meghatározása (requirement capture). Ezen munkafolyamat során összegyűjtjük és felsoroljuk a rendszer működésével szemben támasztott kezdeti elképzeléseket, leírjuk azt, hogy a rendszernek milyen környezetben kell működnie, valamint felsoroljuk a funkcionális (működéssel kapcsolatos) és nem-funkcionális (pl. válaszidők, bővíthetőség, alkalmazott technológiák, stb.) követelményeket. A követelmények meghatározása során alapvetően a felhasználók szempontjából írjuk le a rendszert, így annak egy külső képét rögzítjük.

A következő munkafolyamat, az elemzés (analysis) folyamán a követelményeket a fejlesztők szempontjának megfelelően rendezzük át, így azok együttesen a rendszer egy belső képét határozzák meg, mely a további fejlesztés kiindulópontja lesz. Az elemzés során rendszerezzük és részletezzük az összegyűjtött használati eseteket, valamint azok alapján meghatározzuk a rendszer alapstruktúráját.

Az elemzés célja a szerkezeti váz kialakítása, mely vázat a következő munkafolyamat, a tervezés (design) formálja teljes alakká és tölti fel konkrét tartalommal, mely az összes – funkcionális és nem-funkcionális – követelménynek is eleget tesz. A tervezésnek az implementációval kapcsolatos összes kérdést meg kell válaszolnia, így részletesen le kell írni az összes felhasznált technológiát, a rendszert független fejlesztői csoportok által kezelhető részekre kell bontani, meg kell határozni az alrendszereket és közöttük a kapcsolódási módokat, protokollokat. A tervezésnek a rendszert olyan részletezettségi szinten kell vázolnia, melyből az közvetlenül, egyetlen kérdés és probléma felvetése nélkül implementálható.

Az implementáció (implementation) során a rendszert az UML terminológiája szerinti komponensekként állítjuk elő, melyek forráskódokat, bináris és futtatható állományokat, szövegeket (pl. súgó), képeket, stb. jelentenek. Az állományok előállítása egyben azok függetlenül végrehajtható, önálló tesztjeit is jelentik. Az implementáció feladata még az architektúra, illetve a rendszer, mint egésszel kapcsolatos kérdések megválaszolása, így az iteráció esetén szükséges rendszerintegráció tervezése, az osztottság (distribution) tervezése.

Az utolsó munkafolyamat, a teszt (test) során összeállítjuk az iterációkon belüli integrációs tesztek és az iterációk végén végrehajtandó rendszertesztek ütemtervét. Azaz megtervezzük és implementáljuk a teszteket, tehát teszt-esetekként megadjuk, hogy mit kell tesztelnünk, teszt-eljárásokként. Megadjuk azok végrehajtási módját, és programokat készítünk, ha lehetséges a tesztek automatizálása. A tesztek végrehajtásával párhuzamosan azok eredményeit szisztematikusan feldolgozzuk, majd hibák vagy hiányosságok esetén újabb tervezési vagy implementációs tevékenységeket hajtunk végre.

7. Ellenőrző kérdések

1. Sorolja fel a szoftverfejlesztés életciklusában megjelenő általános feladatokat.

2. Mi az oka annak a vízesésmodell alkalmazása során már kisszámú iteráció után is befagyasztják az adott fejlesztési fázist?

3. Sorolja fel az evolúciós modellnél alkalmazott konkurens tevékenységeket.

4. Röviden értelmezze az eldobható prototípust.

5. Hogyan csökkenti a komponens alapú fejlesztés a kifejlesztendő szoftverek számát?

6. Mit nevezünk inkremensnek, mi a szerepe az inkrementális fejlesztési modellben?

7. Miben más a spirális fejlesztési modell az egyéb szoftverfolyamat-modelltől?

8. Miért mondják azt, hogy iteratív fejlesztés során kisebb a kockázata annak, hogy a teljes projekt kudarcba fullad?

9. Sorolja fel az RUP a rendszerfejlesztés négy diszkrét fázisát.

10. Mit nevezünk mérföldkőnek?

3. fejezet - A szoftver fejlesztés mint modellezési tevékenység

A komplex rendszerek megértésének és vizsgálatának számos tudományos és mérnöki tevékenység során alkalmazott módszere, hogy nem közvetlenül a valóságos rendszert elemezzük, hanem elkészítjük annak egy, a céljainknak megfelelő modelljét. A modelleken végzett vizsgálatok eredményét aztán alkalmazzuk a valóságos rendszerre.

Ezt az elvet felhasználhatjuk a szoftver rendszerek elemzése és fejlesztése esetén is.

1. A modell fogalma

A modell készítése egy absztrakciós folyamat. A modellezendő eredeti egy lehetséges leképezése. A leképezési folyamat két kulcseleme a kiemelés és az elhanyagolás:

1. kiemeljük, hangsúlyosabbá tesszük az eredeti lényeges elemeit, tulajdonságait, viselkedés formáit,

2. elhanyagoljuk, elhagyjuk az eredeti lényegtelen elemeit, tulajdonságait viselkedés formáit.

Hogy mi számít „lényegesnek” és „lényegtelennek”, az a modellalkotás céljától függ. Ugyanannak az eredetinek számtalan, különböző célokat szolgáló modellje lehet. Ha egy embert egyetemi oktatóként akarunk modellezni, lényegtelen a vércsoportja (pedig van ilyen tulajdonsága) és hogy például jól zongorázik, de lényeges a végzettsége és hogy tud vizsgáztatni. Ugyanennek a személynek a modellje egy kórházi rendszer pácienseként pedig fontos tulajdonságként fogja tartalmazni a vércsoportját, de lényegtelen, hogy mi a végzettsége.

Bár a modellkészítés egy absztrakciós folyamat, eredménye, megjelenési formája lehet:

1. Fizikailag létező, mint például egy épület makettje, vagy egy autó szélcsatornába szánt modellje.

2. Maga is absztrakt, azaz valamilyen leíró eszközzel definiált. A leíró eszköz lehet teljesen formalizált, ami a modell precíz definícióját teszi lehetővé (például a szélcsatornában kialakuló áramlási viszonyokat leíró matematikai modell). Ennek ellenpontja a minden formalizmust nélkülöző szöveges leírás.

Érdemes megemlíteni, hogy éppen a mérnöki munkát segítő szoftverek elterjedésével a fizikailag létezőként megvalósítható modellek is egyre gyakrabban absztrakt formát öltenek. Például egy épület makettjének tényleges elkészítése helyett az épület látványát szemléltethetjük egy háromdimenziós megjelenítést lehetővé tevő vizualizációs programmal is.

A szoftverfejlesztés során az eredeti rendszer vagy a szoftver modelljei absztrakt modellek, szükséges tehát valamilyen modell leíró eszköz a dokumentálásukhoz. A rendszermodellek definíciójára szolgáló, a gyakorlatban is használható eszközök a teljesen formális és a szöveges leírás között helyezkednek el: szabványosított diagramokat és azokat kiegészítő szöveges elemeket egyaránt tartalmaznak. A rendszermodellek rögzítésére szabványosított UML modellező nyelvet a jegyzet 7.-10. fejezetei ismertetik.

2. A modell készítés folyamata

A szoftver rendszerek komplexitása szükségessé teszi a rendszermodellek készítési folyamatának átgondolását. A modellalkotás során az alábbi alapvető problémákat kell megoldani:

1. A fejlesztendő rendszer komplex, tehát a modellje is az.

2. Biztosítani (és ellenőrizni) kell, hogy a modell valóban a megoldandó feladatot reprezentálja.

Az egyes módszertanok úgy is tekinthetők, mint amelyek a modellkészítés szabályaira fogalmaznak meg előírásokat vagy ajánlásokat. A konkrét eljárásoktól függetlenül is vannak azonban olyan általános alapelvek, amelyeket a modern fejlesztési folyamatokban alkalmazunk:

1. Evolúciós fejlesztés,

2. különböző modellek az egyes fejlesztési fázisokban,

3. modell nézetek.

2.1. Az evolúciós fejlesztés

Az egy időben kezelendő komplexitás csökkentésének egyik módja az evolúciós szoftver folyamat modell követése. Ilyenkor nem egyszerre építjük fel a teljes modellt, hanem egy egyszerű modellből kiindulva, folyamatos finomításokkal (inkrementumokkal) érjük el a célként kitűzött rendszer megvalósítását. Ha szükséges, még az inkrementumok felépítését is részekre (iterációkra) osztjuk.

Ennek a munkamódszernek az előnyei:

1. Egy lépésben viszonylag egyszerű fejlesztési lépést kell végrehajtani, ami a fejlesztők számára könnyen átlátható.

2. A fejlesztő csapat tagjai gyakran jutnak sikerélményhez, mert lezártak egy fejlesztési szakaszt, ezáltal motiváltabban lesznek.

3. Minden inkrementum eredménye egy működő modell, amely ellenőrizhető, és a megrendelőnek is prezentálható, így a megrendelő is követheti a munka előrehaladását. A megrendelővel való egyeztetés még idejében kiszűrheti az esetleges félreértéseket.

2.2. Az egyes fejlesztési fázisok modelljei

Mint azt már az előző fejezetben is említettük, minden fejlesztési folyamatot fázisokra, résztevékenységekre bonthatunk. Minden egyes résztevékenységhez hozzátartozik az a modell, amelyet annak végrehajtása során kell megalkotnunk. Ennek megfelelően egy fejlesztési folyamatban az alábbi modelleket tudjuk megkülönböztetni:

1. Analízis modell (más néven szakterületi modell). Célja a rendszerrel szemben támasztott követelmények, a rendszer elemeinek és az alapvető folyamatainak a tisztázása. Ebben a fázisban a modellezendő rendszer fogalmaival dolgozunk.

2. Tervezési modell. Az analízis modellben leírt rendszer elemek megvalósításához szükséges elemeket tartalmazza. Az analízis modell kiegészítése az implementációhoz szükséges elemekkel és részletekkel.

3. Implementációs modell. Tartalmazza a tényleges implementációhoz szükséges valamennyi információt.

Amint a fenti felsorolásból is kitűnik, az egyes munkafázisok felhasználják, kiegészítik és pontosítják az előző fázisban létrehozott modelleket.

2.3. Modell nézetek

Az evolúciós szemlélet és a fejlesztési munka résztevékenységekre bontása segít abban, hogy uralni tudjuk a rendszer komplexitásából adódó nehézségeket, de az így létrehozandó modellek még mindig túl bonyolultak lennének, mert az adott rendszer túl sok aspektusát kellene reprezentálniuk.

Magyarország egy lehetséges modellje lehet egy térkép. Ha azonban minden olyan információt, ami az országra jellemző, egy térképen szeretnénk megjeleníteni, az egy áttekinthetetlen modellt eredményezne, hiszen egyszerre kellene például a hegy- és vízrajzot, az utakat, a településeket, az ásványvagyon elhelyezkedését, a közigazgatási, a népességi, a mezőgazdasági műveléssel, a turizmussal és még nagyon sok egyébbel kapcsolatos adatokat ábrázolnia.

Ehelyett nyilván célszerűbb a különböző szempontokat tükröző, többféle térképet készíteni. A közlekedéshez autós térképet használunk, a túrázáshoz a domborzati viszonyokat pontosabban jelölni tudó turista térképet stb. Ezzel ugyanannak a modellnek a különböző nézeteit jelenítjük meg.

A gépészmérnöki szakmában a bonyolult térbeli testek ábrázolásakor a legtöbbször alkalmazott módszer (géprajz) szerint a testet egy derékszögű koordinátarendszerbe helyezik el, és a három síkra vetített vetületét rajzolják le.

A nézetek alkalmazása tehát valójában azt jelenti, hogy nem egy összetett modellt készítünk el, hanem a rendszert különböző nézőpontokból ábrázoljuk. A nézetek és az összetett, eredeti modell szorosan összefüggenek. Ha a géprajz valamelyik vetületén változtatunk, az a többi vetületen is változásokat okoz, és megváltoztatja a test térbeli alakját is. Ez visszafelé is érvényes: ha a térbeli alakzaton (a modellen) változtatunk, a változásoknak az egyes vetületekben is meg kell jelennie.

A nézetek alkalmazása egyszerűsíti a modellezési munkát, mert egyszerre csak egy szempont szerint kell vizsgálni a modellt.

Mivel azonban a különböző nézőpontok ugyanannak a modellnek a „vetületei”, az egyes nézőpontok összevethetők, és a modell készítési munka ellenőrzésére használhatók fel. Ha például az autós térképen egy útszakaszt erősen emelkedőnek jelöltünk meg, akkor a domborzati térképen ott kell lennie az emelkedést okozó hegynek.

A szoftver fejlesztési munka során általában az alábbi nézőpontokat szoktuk megkülönböztetni.

2.3.1. Funkcionális nézet

A rendszer a felhasználó nézőpontjából: milyen funkciókat biztosít számára. Ezt a nézetet a követelmény analízis során kezdjük el építeni, és általában a további fázisokban módosul.

2.3.2. Strukturális, statikus nézet

A rendszer elemeit (objektumait) és azok kapcsolatait ábrázolja. Kezdetben az analízis modell egy nézete, de a fejlesztési folyamat során a megoldási tér elemei is kiegészítik.

2.3.3. Dinamikus nézet

A rendszer egyes részeinek (objektumainak) viselkedése a működés során. Tartalmazhatja:

1. a részek lehetséges állapotait,
2. azt, hogy milyen események következtében megy egyik állapotból a másikba,
3. az üzenetküldések sorozatait (időben elhelyezve),
4. egy adott állapotban végrehajtandó tevékenységsort.

2.3.4. Implementációs nézet

A rendszer működéséhez szükséges szoftver elemeket és azok kapcsolatait ábarázolja.

2.3.5. Környezeti nézet

Ábrázolja a rendszer működéséhez szükséges külső hardver és szoftver erőforrásokat, ezek kapcsolatait, a rendszernek és a környezetének az együttműködését.

4. fejezet - Fejlesztési módszertanok

A fejlesztési módszertanok a szoftver fejlesztési munka technológiai leírásai. Angol megfelelője: methodology. Az első módszertanok a szoftver krízis jelenségének felismerése után, az 1970-es évek elején keletkeztek, azóta számos módszertant fejlesztettek ki és próbáltak ki a gyakorlatban. A módszertanok fejlődése máig sem fejeződött be, és folytatódni fog még jó ideig, mert a megoldandó problémák jellege és a megoldásokhoz használható technológiák folyamatosan fejlődnek, és ezeket a változásokat a szoftverfejlesztési folyamatnak is követnie kell.

1. A módszertan fogalma

Egy szoftvertervezési módszertan a fejlesztési tevékenység strukturált megközelítése, azon szabályok, ajánlások, a gyakorlatban bevált tapasztalatok összegzése, amelyek elősegítik, hogy a szoftver határidőre, adott költségkereten belül és megfelelő minőségben elkészülhessen.

A módszertan egy klasszikus definíciója (Orr, 1989)

„A módszertan különböző, közös filozófiára épülő módszerek összessége, amelyek egységes keretbe illesztve egyértelműen meghatározzák a rendszerfejlesztés életciklusát.”

Röviden összefoglalva minden módszertan arra fogalmaz meg szabályokat, hogy a szoftverfejlesztési folyamat (egy szoftver projekt) során ki – mikor –mit – hogyan csináljon a végeredmény elérése érdekében. Vizsgájuk meg az ebben a rövid definícióban szereplő négy összetevőt.

Ki. A fejlesztésben résztvevő szereplőket határozza meg. A munka szempontjából nem a személyek, hanem azok szerepe (szerepköre) a lényeges, azaz hogy milyen munkafolyamatot kell elvégeznie. A fejlesztéshez szükséges tipikus szerepkörök lehetnek például

1. elemző,
2. vezető tervező (elterjedt angol kifejezéssel „architect”),
3. vezető és beosztott programozó,
4. teszt tervező, teszt mérnök,
5. szakterületi szakértő,
6. dokumentátor,
7. a fejlesztési projekt lebonyolításáért felelős projekt menedzser és további munkatársak.

Ezeket a szerepköröket természetesen egy adott projekt esetén valódi munkatársakhoz kell rendelni. A hozzárendelés általában a projekt egy időszakára vonatkozik, és egy munkatárs több szerepkört is elláthat. (Azaz a szerepkör – személy összerendelés több-több jellegű, és nem állandó). Egy gyakorlott munkatárs például gyakran betölti az elemző, vezető tervező, vezető programozó szerepköröket, de egy kis létszámú projekt esetén akár még több szerepkört is el kell látnia.

Mikor. Az egyes módszertanok definiálják a projekt során követendő munkafolyamatot (work flow-t). Ehhez a 2. fejezetben ismertetett valamelyik szoftverfolyamat modellt, vagy azok kombinációját veszik alapul. A munkafolyamat definíciója előírja a fejlesztőknek, hogy az egyes résztevékenységeket (amelyet a mit kérdésre válaszolva határoz meg a módszertan) milyen sorrendben végezzék. A projekt menedzser számára megadja, mikor, milyen tevékenységek végrehajtását kell ellenőriznie, esetleg ezek közül melyek párhuzamosíthatók, hogy a fejlesztés átfutási ideje csökkenthető legyen.

Mit. A módszertanok a fejlesztési folyamatot elemi lépések sorozatára bontják. Meghatározzák, hogy egy elemi lépés a korábbi lépések eredményeiből mit vagy miket használjon fel, és mi legyen ennek a lépésnek a végeredménye (terméke).

Hogyan. Az elemi lépések végrehajtásának módját definiálja (szokás módszernek is nevezni). A módszertanok eljárásokat definiálnak a fejlesztési lépések végrehajtására. Előírják, hogy egy elemi lépés végeredményét milyen módon kell dokumentálni. A dokumentáció elkészítését valamilyen technikával (jelölésrendszerrel) támogatják. A technika a fejlesztési munkát, a fejlesztők közötti, valamint a felhasználó-fejlesztő közötti kommunikációt segítő szimbólumrendszer, diagramok, ábrázolási és dokumentálási eszközök összessége

A különböző módszertanok tartalmazhatnak azonos lépéseket, és azonos technikákat. Korábban minden módszertanhoz saját jelölésrendszert dolgoztak ki. Az UML megjelenése a jelölésrendszert szabványosította, így a mai módszertanok kivétel nélkül azt használják.

Fel kell hívnunk a figyelmet arra, hogy a magyar szakirodalom szóhasználata nem teljesen következetes. Szigorúan véve a módszer (angolul method) egy elemi fejlesztési lépés, a módszertan a teljes fejlesztési folyamat végrehajtásának módját jelenti. A módszer szó azonban néha a módszertan szinonímájaként jelenik meg, így a szövegkörnyezetből kell megállapítami, hogy melyik jelentésben használják.

2. A módszertanok fejlődése

Mint már említettük, a szoftver krízis jelenségének felismerése indította el a szakmán belül a módszeres szoftverfejlesztési folyamat kifejlesztésére tett erőfeszítéseket, rámutatva a szoftverfejlesztés mérnöki megközelítésének fontosságára.

Már az első tanulmányok felismerték a fejlesztési folyamat elemzési, tervezési és implementációs szakaszokra bontásának fontosságát. A korai módszertanok a strukturális programozás elvén alapultak. Ennek lényege, hogy a megoldandó problémát strukturális elemzés alapján kell implementálható elemekre bontani.

Az 1970-es évek egyben a funkcionális szemléletű programozási nyelvek elterjedését is hozták. Ezeknek a nyelveknek az alapgondolata, hogy a program adatszerkezetek és azokon manipuláló algoritmusok (processzek, feldolgozási lépések) együttese. Ezért a strukturális elemzés is ezen két összetevő közül az egyiket tekintette kiindulási alapnak.

2.1. Processz alapú módszertanok

Ez a megközelítés a rendszert a funkcionális dekompozíció segítségével bontja kisebb funkcionális részekre. Minden funkcionális modul esetén elemzi az input és az output adatokat, és a modul által megvalósítandó leképezést ez alapján tervezi meg. A processz alapú módszerek legismertebb képviselői:

1. HIPO (Hierarchy Input Processing Output), IBM

2. SADT (Structured Analysis and Design Technique), Douglas T. Ross

2.2. Adat alapú módszertanok

A processz alapú megközelítés duálisaként a strukturálás alapját az adatszerkezetek képezik. Alapgondolatuk, hogy az adatszerkezetek hierarchiája alapján tervezhető meg a processzek, feldolgozási lépések struktúrája. Az adat alapú módszerek legismertebb képviselői:

1. JSD (Jackson System Development)

2. SSADM (Structured Systems Analysis and Design Method)

2.3. Objektum orientált módszertanok

A korai módszerek közül a tapasztalatok alapján az adat alapú fejlesztési módszertanok váltak be jobban a gyakorlatban. A legnagyobb „karriert” talán az SSADM futotta be, amely a brit (és az 1990-es évek elején a magyar) kormányzati projektek szabványa lett.

Az objektum orientált programozás nyelvek megjelenésével a program definíciója is megváltozott. Az objektum orientált programozás szemléletében a program egymással kommunikáló objektumok halmaza, amelynek struktúráját az osztályok közötti relációk, viselkedését az objektumok közötti üzenetváltások határozzák meg. A

szoftver technológia kutatásának egy ismert képviselőjétől, Bertrand Meyertől származó definíció az objektum orientált fejlesztésre:

„Software rendszerek felépítése absztrakt adattípusok implementációinak strukturált együtteséből.”

Rövidesen nyilvánvalóvá vált, hogy az új szemléletű nyelvek előnyeit csak akkor lehet hatékonyan kihasználni, ha az implementációt megelőző fázisok is ezt a szemléletet követik. Így először a tervezés, majd az elemzés fázisait is lefedő új módszertanok jelentek meg. Ezzel párhuzamosan jelent meg a fejlesztés modell szemléletű megközelítése is.

Az objektum orientált módszertanok néhány ismertebb képviselője (a teljesség igénye nélkül):

1. OMT (Object Modelling Technique, Rumbaugh, 1991, majd 1993)
2. OOD (Object Oriented Design, Booch, 1991, majd 1993)
3. OOA (Object Oriented Analysis, Coad & Yourdon, 1991)
4. OOSD (Object-Oriented Structured Design, Wasserman, 1990)
5. HOOD (Hierachical Object Oriented Design, 1989)
6. Responsibility -Driven Design, Wirfs & Buck, 1990
7. OOSE (Object Oriented Software Engineering, Jacobson, 1992)
8. RUP (Rational Unified Process) (UML + fejlesztési folyamat ajánlás, 1998-1999)
9. scrum (Ken Schwaber, Mike Beedle, 2001)

A RUP néhány alapvető tulajdonságát a 2. fejezetben a benne megfogalmazott szoftver folyamat modell ismertetése során vázoltuk. Mivel a mai viszonyok között is alkalmazható, keretrendszer jellegű módszertan, ismertetése jelen jegyzet kereteit meghaladja, részletesebb tárgyalása a mesterképzés vonatkozó tárgyának a feladata.

A scrum az úgynevezett „agilis szemléletet” támogató módszertan, áttekintése szintén mesterképzés tárgya lehet.

A felsoroltak közül az OMT, OOD és OOSE az 1990-es években számos fejlesztési projektben bizonyították használhatóságukat. Bár a mai fejlesztések igényeinek már nem felelnek meg, jelölésrendszereik az UML modellező nyelv elődjeinek tekinthetők, ezért szerepük máig meghatározó.

3. Az OMT módszertan

Az OMT az egyik legkorábban kifejlesztett objektum orientált fejlesztési módszertan. Elterjedéséhez hozzájárultak az alábbi jellemzői:

1. Világos, jól definiált fogalmakkal dolgozik.
2. Jó átmenetet biztosított a strukturált és az objektum orientált módszerek között.
3. Konkrét és gyakorlatias tanácsokat fogalmazott meg a fejlesztésben részvevők számára.
4. Jól definiált, könnyen követhető lépések sorozataként határozta meg a fejlesztési munka menetét.

A kifejlesztése óta eltelt majd két évtized során a fejlesztésekkel szemben támasztott követelmények változásai miatt a módszertant ma már nem alkalmazzuk a mindennapi munkában, elsősorban az alábbi hátrányi miatt:

1. Nehezen képes kezelni a mai igényeknek megfelelő bonyolultságú rendszereket.
2. A követelmény analízis dokumentációját tekinti kiindulópontnak, tehát erre a fázisra nem ad tanácsokat.

Annak ellenére, hogy ma már közvetlenül nem alkalmazzuk, mint módszertant, didaktikai szempontból mégis fontos szerepe van.

1. Viszonylagos egyszerűsége lehetővé teszi az áttekintését egy tárgy keretein belül.

2. Már ebben a korai módszertanban is megjelennek olyan fontos megközelítési módok, amelyek a modern módszertanoknak is részét képezik.

3. Gyakorlatias tanácsai a mai fejlesztési projektekben is jól használhatók.

Ezért ebben az alpontban áttekintjük az OMT módszertant, és összefoglaljuk, melyek azok az elemei, amelyek ma is jól használhatók.

3.1. Az OMT szemlélete

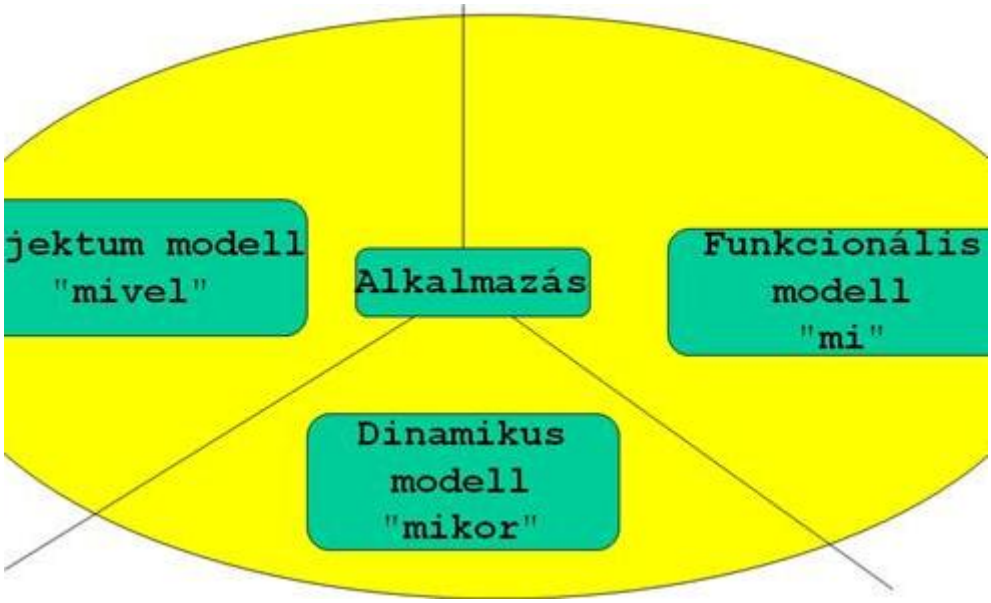
Az OMT szemlélete szerint a rendszert három, ortogonális nézet segítségével modellezzük. Az ortogonalitás a nézetek függetlenségét jelenti. Ez a három nézet:

1. objektum modell,

2. dinamikus modell,

3. funkcionális modell.

4.1. ábra - Az OMT modelljei



Bár az OMT a modell kifejezést használja, ezek valójában nézetek.

A fejlesztés implementációt megelőző tevékenységeit négy időbeli fázisra osztja:

1. analízis,

2. rendszertervezés,

3. objektum tervezés,

Az implementációval nem foglalkozik.

Mindhárom fázisban mindhárom modell nézetet használnunk kell, de különböző absztrakciós szinteken.

Eredetileg definiált egy saját jelölésrendszert, de a ma már szabványos UML nyelv elemeit rendeljük az egyes tevékenységekhez.

3.2. Az objektum modell

Az objektum modell a rendszer általános struktúráját ábrázolja. Mai szóhasználattal ez a rendszer statikus nézete.

Használható jelölésrendszer:

1. osztály diagram,
2. csomag diagram,
3. összetett struktúra diagram,
4. komponens diagram.

3.3. A dinamikus modell

A rendszer építőelemeinek viselkedése (időbeli változása). Mai szóhasználattal a dinamikus nézet egyik része.

Minden objektumra megvizsgáljuk:

1. hogyan változtatja állapotát,
2. hogyan hat a környezetére.

Használható eszközök:

1. állapotgép diagram (UML),
2. szekvencia diagram (UML),
3. kommunikációs diagram (UML).

3.4. A funkcionális modell

A rendszeren belüli számítások, feldolgozások. Mai szóhasználattal a dinamikus nézet másik része.

Minden objektumra megtervezzük

1. az objektum modell műveleteit,
2. a dinamikus modell akcióit,
3. az objektum modell korlátozó feltételeit.

Használható eszközök:

1. használati eset diagram
2. aktivitás diagram

4. Az OMT és a modern fejlesztési folyamat

Az OMT számos olyan elvet tartalmaz, amely a mai fejlesztési környezetben is használható.

1. Modell szemlélet.
2. Nézetrendszer. Felismerte, hogy a modellezési munka nézetek alkalmazásával egyszerűsíthető, és hogy a nézetek összevetése, az esetleges ellentmondások feltárása a modellezési folyamat ellenőrzésére használható.
3. A folyamatos ellenőrzés elve. Minden ajánlott elemi tevékenység után a módosított nézet ellenőrzését írja elő-

4. A statikus nézet előállítását azzal segíti, hogy tanácsokat a rendszer strukturális elemeinek azonosítására.

Az OMT ajánlásai kisebb fejlesztések esetén ma is jól használhatók, az alábbi kiegészítésekkel:

1. A követelmények összegyűjtése, strukturálása egy használati eset modell elkészítésével oldható meg, ezt az OMT késznek tételezi fel.

2. Az OMT eredetileg a követelmények szöveges leírásából indul ki, e helyett az analízis fázis a használati eset modell elemzését jelenti.

3. Jelölésrendszerként az UML-t használjuk.

4. Ha a rendszer összetettsége szükségessé teszi, több absztakciós szinten, iteratív módon alkalmazzuk az eredeti módszertan elveit.

A jelen jegyzet az OMT elveire támaszkodva mutat példát a 10.-13. fejezetekben az analízis, tervezési és implementációs modell elkészítésére olyan szinten, hogy a hallgatók féléves feladatának a megoldását segítse.

5. fejezet - Követelmény analízis

A szoftverfolyamat, amelynek eredményeképpen előáll egy kész szoftvertermék, tevékenységek és kapcsolódó eredmények soraként értelmezhető. Ezek a folyamatok komplexek, és nagyon összetettek, valamint erősen függenek az emberi tevékenységektől és döntésektől. Emiatt a folyamatok nem automatizálhatók a megfelelő mértékben, mindig szükség van az emberi tényezőre. Fontos cél lehet, hogy az emberi tényezők mellett minél több részfolyamatot lehessen teljesen vagy féli automatizálni. A számítógéppel segített szoftvertervezés eszközei (CASE) képesek a folyamatok bizonyos tevékenységeinek támogatására, de nincs lehetőség nagyobb mértékű automatizációra. Nyugodtan kijelenthetjük továbbá azt is, hogy nincs olyan folyamat, amely minden számára megfelelő lenne. Különböző szervezetek a szoftverfejlesztést homlokegyenest különböző nézőpontokból közelítik meg. Mégpedig úgy, hogy a folyamatokat úgy alakítják ki, hogy kiaknázzák a szervezeten belül az emberek különféle képességeit és a fejlesztő rendszer jellegzetességeit.

Bár minden szoftverfolyamat különböző, vannak olyan alapvető tevékenységek, amelyek minden szoftverfolyamatban közösek:

1. Szoftverspecifikáció: a szoftver funkcióit, illetve annak megszorításait definiálja.

2. Szoftvertervezés és implementáció: A specifikációnak megfelelő szoftvert elő kell állítani.

3. Szoftvervalidáció: a szoftvert validálni kell, hogy biztosítsuk, azt fejlesztettük, amit az ügyfél kíván.

4. Szoftverevolúció: A szoftvert úgy kell kialakítani, hogy megfeleljen a megrendelő kívánsága szerint történő változtatásoknak.

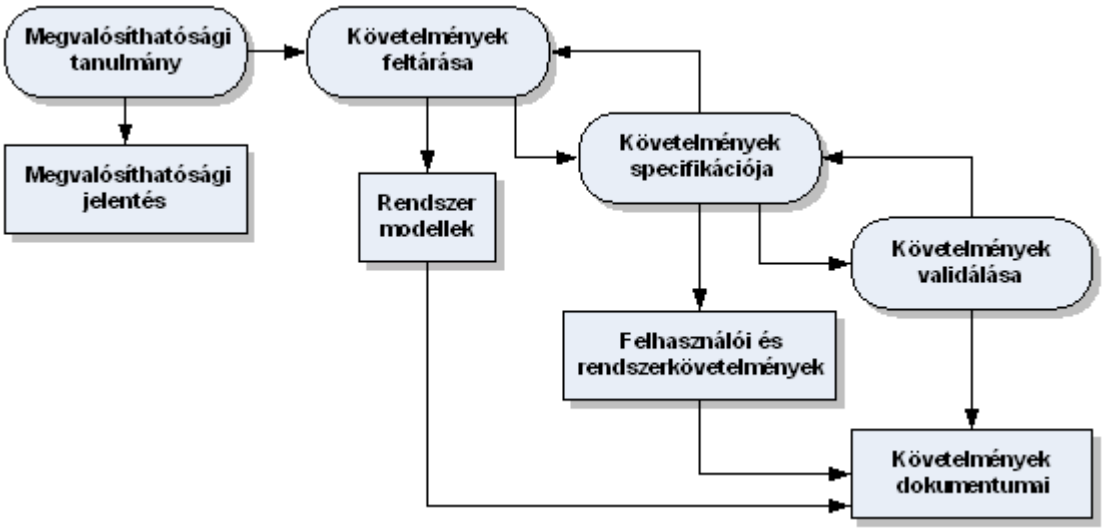
Bárki számára felmerülhet a kérdés, hogy annak ellenére, hogy tudjuk nincs „ideális” szoftverfolyamat, miért érdemes alkalmazni a szoftverfejlesztési elveket? Bár a folyamatok különböznek, számos terület van, ahol a szervezeten belüli szoftverfolyamatokon javíthatunk, mert gyakran elavult technikákat tartalmaznak, vagy esetleg nem is élnek a tervezési módszerek lehetőségeivel.

A fejezetben a továbbiakban a szoftverkövetelmények analízisével, a szoftverspecifikációval foglalkozunk részletesen.

1. A szoftverspecifikáció

A szoftverspecifikáció során, vagy más néven a követelménytervezés fázisában elsődleges célunk az, hogy részletesen megértsük és definiáljuk az adott rendszer működését. Meg kell határozni, hogy a rendszernek milyen szolgáltatásokat kell biztosítania. Mindezek mellett azonosítjuk a rendszer üzemeltetésének és fejlesztésének megszorításait is. A követelmények tervezése a szoftverfolyamat különösen kritikus szakasza. Az ebben a szakaszban vétett hibák elkerülhetetlenül problémákhoz vezetnek majd a rendszertervezés későbbi szakaszában és az implementációban. A folyamat eredménye a követelmény dokumentum előállítása, amely a rendszer specifikációja. A követelmények tervezési folyamatát a következő ábra mutatja be.

5.1. ábra - A követelménytervezés folyamata



A követelmények tervezésének négy nagy fázisát különböztethetjük meg. Ezeket találjuk az ábra felső vonalában:

1. Megvalósíthatósági tanulmány: első lépésként meg kell becsülni, hogy a felhasználók kívánságai kielégíthetők-e az adott szoftver- és hardvertechnológia mellett. A vizsgálatoknak el kell dönteniük, hogy a rendszer költséghatékony-e, és hogy az kivitelezhető-e. A megvalósíthatóság elemzésének relatíve olcsónak és gyorsnak kell lennie. Eredménye a megvalósíthatósági jelentés.

2. Követelmények feltárása és elemzése: ez a folyamat a rendszerkövetelmények meglévő rendszereken történő megfigyelésén, a potenciális felhasználókkal és beszerzőkkel folytatott megbeszéléseken, tevékenységelemzéseken alapszik. Akár egy vagy több különböző rendszermodell, illetve prototípus elkészítését is magában foglalhatja.

3. Követelmény specifikáció: a követelményspecifikáció az elemzési tevékenységek során összegyűjtött információk egységes dokumentummá alakítása. A dokumentumnak a követelmények két típusát, a felhasználói követelményeket, és a konkrét rendszerkövetelményeket kell tartalmaznia.

4. Követelmény-validáció: a tevékenység során ellenőrizzük, hogy mennyire valószerűek, konzisztensek és teljesek a követelmények. Fontos cél, hogy a folyamat során feltárjuk a követelmények dokumentumában található hibákat, és kijavítani.

Nagyon fontos megjegyeznünk, hogy nem mindig célszerű a követelménytervezés különböző tevékenységeit szigorú sorrendben végrehajtani. Például a követelmények elemzése folytatható a meghatározásuk és specifikálásuk alatt, továbbá a folyamat során bármikor napvilágra kerülhetnek új követelmények is. Bizonyos esetekben ilyenkor az elemzés, a meghatározás és a specifikáció tevékenységei összefésülhetők, és egymást átfedhetik a folyamatban.

2. A szoftver követelmények

Egy-egy szoftver tervezése során sokszor nagyon nehéz pontosan leírni, hogy hogyan kell működnie a rendszernek. Korábban említettük, hogy a rendszert szolgáltatások és megszakítások összességével írjuk le, amely leírásokat pedig követelményeknek nevezzük. Ez az elemzés és tervezés egyfajta alapegységeként értelmezhető, amelyből a kész rendszert építjük fel. A követelmények tervezése pedig az a folyamatot nevezzük, amely során meghatározzuk és elemezzük a szolgáltatásokat és megszorításokat, majd dokumentáljuk és ellenőrizzük azokat.

A követelmény elnevezés nagyon tág fogalomként jelenik meg a szoftveriparban. Nem használják következetes módon, így a továbbiakban szükséges, hogy első megközelítésben két különböző szintre bontva folytassuk a tárgyalását.

1. A felhasználói követelmények: magas szintű absztrakt követelmények, melyek az ügyfelek és a fejlesztők képviselői (menedzserek) számára készülnek elsősorban, akik nem rendelkeznek részletes technikai ismerettel a rendszerről. Technikai értelemben diagramokkal kiegészített természetes nyelvű dokumentum,

amely pontosan leírja mely szolgáltatásokat várunk el a rendszertől, és annak mely megszorítások mellett kell működnie.

1. A rendszerkövetelmények: ezek a követelmény-leírásoka rendszer funkcióit, szolgáltatásait és működési megszorításait jelölik ki részletesen. A rendszerkövetelmények dokumentumának pontosnak kell lennie. Pontosan meg kell határoznia, mit kell implementálni. Ez a rendszer vásárlója és a szoftverfejlesztő közötti szerződés része is lehet.

A rendszer-specifikációt tipikusan ezen a két különböző szinten írják le. Nagyon fontos, hogy a két szint leírása egyaránt megjelenjen a specifikációban, mert a rendszerről különböző típusú olvasók számára közölnek információt. Míg a felhasználói követelmények inkább absztrakt leírásai a rendszernek, addig a rendszerkövetelmények részletező leírások, megmagyarázva a fejlesztendő rendszer által biztosítandó szolgáltatásokat és funkciókat.

Az eddig bemutatott két típus leginkább a leírás részletezettsége alapján szeparálja a követelményeket. Egy másik irányú megközelítés, a megvalósítandó funkciók alapján további három csoportot jelölhetünk ki:

A funkcionális követelmények: a rendszerfunkciók (szolgáltatások) ismertetései, hogy hogyan reagáljon a rendszernek bizonyos bemenetekre.

Nemfunkcionális követelmények: a funkciókra és szolgáltatásokra tett megszorítások. Általában a rendszer eredő tulajdonságaira vonatkoznak. Magukban foglalnak időbeli korlátozásokat, a fejlesztési folyamatra tett megszorításokat, szabványokat.

Szakterületi követelmények: a rendszer alkalmazási szakterületéről származnak és e szakterület jellegzetességeit és megszorításait tükrözik.

A valóságban természetesen a különböző követelménytípusok közötti különbségek nem értelmezhetők ilyen élesen, mint az alábbi tárgyalásban.

2.1. Funkcionális követelmények

Egy rendszer funkcionális követelményei leírják, hogy a rendszernek milyen funkciókkal kell rendelkezni, hogyan kellene működnie (Például aktualizálások, lekérdezések, jelentések, kimenetek, adatok, más rendszerekkel való kapcsolat). Ezek a követelmények a fejlesztett szoftver típusától, a szoftver leendő felhasználóitól függenek. Természetesen itt is megjelenik a követelmények másik megközelítése, miszerint kifejthetők mind felhasználói követelményekként, melyek rendszerint egészen absztrakt leírások, mind pedig rendszerkövetelményként, amelyek a rendszerfunkciókat részletesen írják le.

Nagyon fontos, hogy a rendszer funkcionális követelményt leíró specifikációjának tartalmaznia és definiálnia kell a megrendelő által igényelt összes szolgáltatást, az az teljesnek és ellentmondásmentesnek kellene lennie. Az ellentmondás-mentesség szintén fontos, amely azt jelenti, hogy ne legyenek ellentmondó meghatározások a leírásokban. A gyakorlatban azonban egy nagyméretű, összetett rendszerek fejlesztésénél gyakorlatilag lehetetlen a követelmények teljességét és ellentmondás-mentességét elérni. Ennek egyik oka, hogy nagyméretű, összetett rendszerek specifikációinak írásakor könnyű kifelejteni dolgokat, illetve hibát elkövetni. Másik ok, hogy a különböző kulcsfiguráknak eltérő – és gyakran ellentmondó – igényeik vannak.

2.2. Nemfunkcionális követelmények

A nemfunkcionális követelmények a funkcionális követelményekkel ellentétben nem közvetlenül a rendszer által biztosított specifikus funkciókkal foglalkoznak, hanem inkább a rendszer egészére vonatkozó eredő rendszertulajdonságokra koncentrálnak. Mit is jelentenek ezek? Példaként néhányat említve: megbízhatóság, válaszidő, tárfoglalás, rugalmasság, robosztusság, hordozhatóság, stb..

A követelmények ezen csoportja gyakran kritikusabb, mint az egyedi funkcionális követelmények csoportja. Ha nem teljesítjük ezeket, vagy kerültek megfogalmazásra konkrét követelményként, gyakran a teljes rendszert használhatatlanná tehetik. Példaként említhetünk egy banki rendszert, amely nem teljesíti a vele szemben támasztott összes biztonsági követelményeket.

Néhány fontos tényező, amely felvetheti a nemfunkcionális követelményeket: felhasználói igények, költségvetési megszorítások, a szervezeti szabályzat, más szoftver- vagy hardverrendszerekkel való együttműködés igénye vagy olyan külső tényezők, mint a biztonsági szabályozások, adatvédelmi rendelkezések.

A nemfunkcionális követelményeket a következőképpen csoportosíthatjuk:

1. Termékre vonatkozó követelmények: olyan követelmények, amelyek alapvetően meghatározzák a termék viselkedését. Néhány példa:

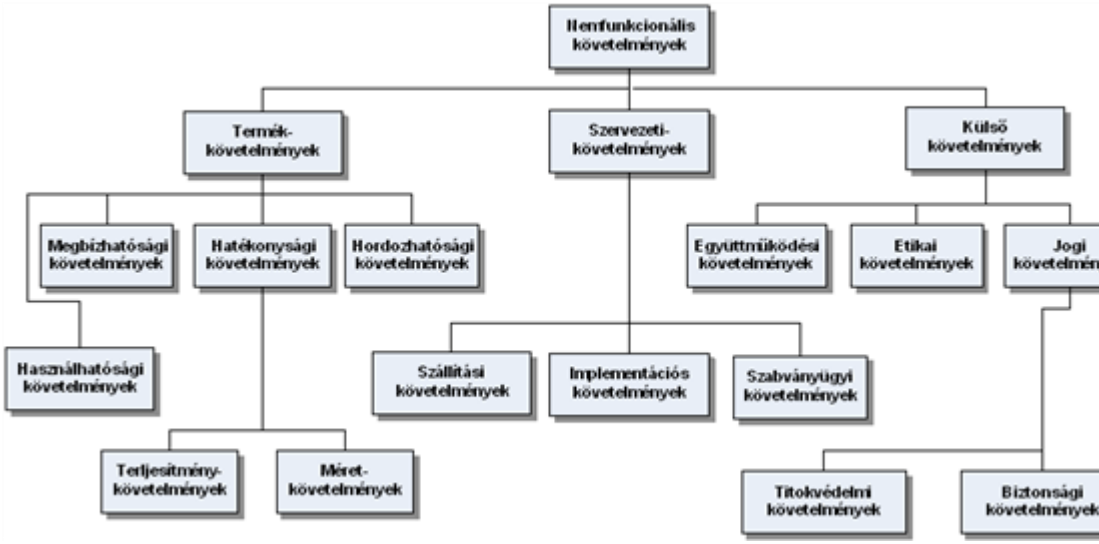
Jól látszik, hogy nem teljesítésük alapvető használhatósági problémákat vetnek fel. Példa: Egy információs weblap felülete frame-ek és applete-ek nélkül legyen implementálva.

1. Szervezeti követelmények: a követelmények ezen alcsoportja mindig a megrendelő vagy/és a fejlesztő cég szervezetének belső szabályzataiból és ügyrendjéből fakadnak. Példa:

A nemfunkcionális követelményekkel kritikus problémája a verifikálhatóság. Míg a funkcionális követelmények esetében a verifikáció egyszerű folyamat, hiszen az ellenőrzés mindig a kérdéses funkció meglétére és helyes működésére vonatkozik, addig a nem funkcionális követelmények gyakran általános elvi megkötések, mint például „a rendszernek könnyen használhatónak kell lennie”. A verifikálhatóság érdekében általános elv az, hogy törekedjünk arra, hogy amikor csak lehetséges a kérdéses nemfunkcionális követelményt valamilyen mennyiségileg, objektíven tesztelhető metrika segítségével fejezzük ki. A gyakorlatban ez azonban sokszor szinte megvalósíthatatlan, mert a rendszer megrendelői számára gyakorlatilag lehetetlen céljaikat mennyiségi követelményekre váltani.

A következő ábra a nemfunkcionális követelmények típusait és azok kapcsolatát mutatja be.

5.2. ábra - Nem funkcionális követelmények alcsoportjai



2.3. Szakterületi követelmények

A követelmények e csoportja nem a rendszer felhasználójának egyéni igényeiből származnak, hanem arról a szakterületről, ahol a rendszer a jövőben alkalmazásra kerül. Általában nagyon speciálisak, és sok problémát okozhatnak a gyakorlatban. Ennek oka, hogy a követelmények az alkalmazás szakterületén használt speciális terminológiával kerülnek megfogalmazásra a megrendelői oldalról, amelyhez a szoftvertervezők általában nem értenek. A megrendelői oldal szakértői kihagyhatnak bizonyos információkat a követelményből, mert az számukra teljesen nyilvánvaló, de a szoftver fejlesztőinek nem. Azonban ha ezek a követelmények nem teljesülnek, nem lehetséges a rendszert kielégítően működtetni.

A probléma elkerülése érdekében a fejlesztő cégnél szükség van legalább egy olyan személyre, aki az igényeknek megfelelően kisebb vagy nagyobb mértékben megismeri az adott szakterületet és segíti a megrendelő és a fejlesztő közötti kapcsolatot. Ezzel minimálisra csökkenthető a meg nem valósított vagy félreértett követelménynek és az újratervezés veszélye.

2.4. Felhasználói- és rendszerkövetelmények

Miután a követelmények csoportosításának második típusát megvizsgáltuk, térjünk vissza az első csoportosításhoz. A rendszer felhasználói szintű követelményei tehát funkcionális, nemfunkcionális és szakterületi követelmények összességéből fog összeállni mégpedig olyan leírási formában, hogy a rendszer azon felhasználói is megértsék azokat, akiknek nincsenek részletes technikai ismereteik. Tipikus olvasói a menedzserek. Éppen ezért olyan magas szintű dokumentációk, melyek a rendszernek csak a külső viselkedését írják le. A felhasználói követelményeket egyszerű nyelven, egyszerű táblázatok, űrlapok és könnyen érthető diagramok segítségével kell közreadni.

Tipikus probléma, ha a felhasználói követelmények túl sok információt tartalmaznak. Ezzel nem csak korlátozza a rendszerfejlesztő szabadságát abban, hogy újító megoldással szolgáljon a felhasználói problémákra, hanem nehezen érthetővé teszi a követelményeket is. A felhasználói követelményeknek egyszerűen a kulcsfontosságú igényekre kell összpontosítania.

A rendszerkövetelmények a felhasználói követelmények részletesebb leírásai. Tehát szintén funkcionális, nemfunkcionális és szakterületi követelmények összessége. Céljuk, hogy részletes magyarázatot adjanak a rendszer működésének egészéről, hogy a rendszernek hogyan kell biztosítania a felhasználói követelményeket. Sokszor a rendszerkövetelmények alapul szolgálnak a rendszer megvalósítási szerződéséhez, és ezért az egész rendszer teljes és konzisztens meghatározását tartalmazniuk kell.

A rendszerkövetelmények ideális esetben egyszerűen a rendszer külső viselkedését és működési megszorításait írják le, de nem szabad, hogy a tervezés és az implementálás miként tjével foglalkozzanak.

2.4.1. Alapvető problémák a követelmények megfogalmazásakor

A természetes nyelvet gyakran használják mind a felhasználói, mind rendszerkövetelmények specifikációjának megfogalmazásához és dokumentálásához. Ebben a formában írt követelményeknél változatos problémák merülhetnek föl. 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. A következő néhány pontban összefoglaljuk, hogy melyek azok a problémák, amely miatt a természetes nyelven írt követelményspecifikációk félreérthetők lehetnek:

1. A természetes nyelvű követelményspecifikáció túl rugalmas. Ugyanazt a dolgot teljesen különbözőféleképpen elmondhatjuk.

2. Az egyértelműség hiánya. Bizonyos esetekben nehéz a nyelvet pontos, egyértelmű módon használni anélkül, hogy a dokumentumot terjengőssé és nehezen olvashatóvá ne tennénk.

3. Követelmények keveredése. A funkcionális követelmények, nemfunkcionális követelmények, rendszercélok és tervezési információk nem különíthetők el tisztán.

4. Követelmények ötvöződése. Több különböző követelmény egyetlen követelményként fogalmazódik meg.

A természetes nyelvű leírások tehát sok bonyodalmat okozhatnak. Megoldás lenne a problémára, ha modularizálni lehetne ezeket követelmények, azonban erre nincs könnyű módszer. Lehet, hogy bonyolult az összes kapcsolódó követelményt megtalálni. Ahhoz pedig, hogy felfedezzük egy változtatás következményét, előfordulhat, hogy minden már vázolt követelményt ellenőriznünk kell.

A rendszerkövetelmények esetében kicsit könnyebb a probléma megoldása, mert itt a követelmények leírását speciális jelölésekkel is kifejezhetjük. Beszélhetünk strukturált természetes nyelvről, tervleíró nyelvről, grafikus jelölésekről, és matematikai specifikációkról.

Strukturált természetes nyelv: a természetes nyelv egyfajta leszűkítését jelenti, melynek az az előnye, hogy egy egységet nyújt a rendszerkövetelmények kifejezésére, mindamellett pedig jórészt megtartja a nyelv kifejezőképességét és érthetőségét. A követelmény-író szabadságát előre definiált követelmény-sablonok korlátozzák. Minden követelményt egy standard módon írunk meg, valamint a leírásban használható terminológia korlátozva lehet.

Példaként az űrlap alapú megközelítést említhetnénk, ahol egy vagy több szabványos űrlapot kell definiálnunk, és végig ezeket használjuk a követelmények kifejtéséhez. Egy általános űrlap felépítése a következő lehet:

1. A funkció vagy entitás definíciója.

2. A bementek leírása, és hogy honnak erednek.

3. A kimenetek leírása, és hogy hová tartanak.

4. Más felhasznált entitások felsorolása.

5. Elő- és utófeltételek (pre-, post-condition).

6. Mellékhatások leírása.

A strukturált természetes nyelvi leírásokhoz tartozik a táblázatos specifikáció módszere is, amely során táblázatos formában készítjük a leírás. A módszer különösen hasznos akkor, amikor alternatív végrehajtási módokat definiálunk.

Tervleíró nyelv (PDL - Program Description Language): szintén a természetes nyelvű specifikáció többértelműségének kivédésére találták ki a programleíró nyelveket. A PDL egy programozási nyelvekhez hasonló, általános szintaktikát alkalmazó kiegészítés a természetes nyelvű specifikációkhoz. Segítségével a természetes nyelven nehezen, vagy terjengősen megfogalmazható tartalmakat könnyedén, és érthetően írhatjuk le. Például valamely ismétlődő folyamatot egy általános programozási nyelvi ciklussal definiálhatunk. Nagy előnye, hogy a leírás nemcsak egyszerűbb és egyértelműbb lesz, hanem szoftveres eszközökkel szintaktikailag és szemantikailag is ellenőrizhetők. Két esetben javasolt a használatuk:

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

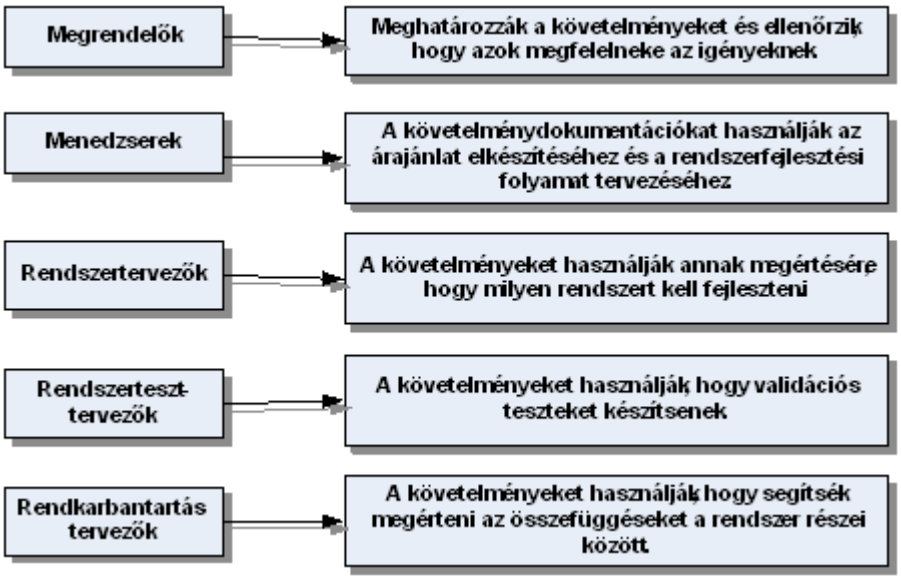
Grafikus jelölések: a rendszer funkcionális követelményeit egy szöveges megjegyzésekkel bővített grafikus nyelv segítségével írja el. Korai példa: SADT. Manapság a használati eset (use-case) leírások és a sorrend (sequence) diagramok használatosak.

Matematikai specifikáció: matematikai elveken (pl. véges állapotú automaták, halmazok) alapuló leírási módok. Egy egyértelmű leírása a rendszer funkcionalitásának, amely kizárja a későbbi vitát a megrendelő és a szállító között. A legtöbb megrendelő azonban nem érti a formális leírásokat és nem hajlandó ilyet a szerződésbe foglalni.

3. Szoftverkövetelmények dokumentuma

A szoftverkövetelmények dokumentuma a szoftverspecifikációs fázis végeredménye, amely tartalmazza a rendszer felhasználói követelményeit és a rendszerkövetelmények részletes specifikációját. Hivatalos leírása annak, amit a rendszerfejlesztőknek meg kell valósítani. Mielőtt javaslatot tennék a dokumentum felépítésére, vizsgáljuk meg azt, hogy kik a követelménydokumentum felhasználói. Ők a rendszerért fizető felsővezetőktől kezdve egészen a fejlesztőkig.

5.3. ábra - A követelménydokumentum használói



Látható, hogy több felhasználó is megjelenik, ezért a dokumentumnak minden rétegre kiterjedőnek kell lennie. A fejlesztett rendszer típusától és a használt fejlesztési folyamattól függ, hogy milyen szintű részleteket érdemes tartalmaznia a követelmények dokumentumának. Ha a rendszert egy kívülálló vállalkozó fogja fejleszteni, akkor rendszerspecifikációnak pontosnak és nagyon részletesnek kell lenni. Ha azonban a fejlesztés házon belül iteratív módon megy végbe, akkor a követelmények dokumentuma kevésbé részletes is lehet.

3.1. A dokumentum felépítése

A követelmények dokumentumának felépítése nagyon változatos lehet. Különböző szervezetek más és más megoldásokat követhetnek a leírás elkészítéséhez. Természetesen nem definiálható olyan dokumentum felépítés, amely minden cég számára kielégítő formát alkothatna, de azért javaslatokat tehetünk a dokumentum gerincének felépítésére. A dokumentum felépítésére számos nagy szervezet definiált szabványokat. A legismertebb szabvány az IEEE/ANSI 830-1998-as. A szabvány nem ideális, de rengeteg jó tanácsot fogalmaz meg a követelmények lefektetésével kapcsolatban. Az alábbi felsorolásban egy IEEE szabványon alapuló lehetséges követelménydokumentum lehetséges szervezését mutatjuk be:

1. Előszó

Körvonalazza a dokumentum célzott olvasóit, ós leírja a verzió történetét, beleértve egy új verzió létrehozásának magyarázatát és az egyes verziókban végzett változtatások összefoglalását.

1. Bevezetés

Indokolja, miért van szükség a rendszerre. Röviden leírja annak funkcióit, és elmagyarázza, hogyan fog együttműködni más rendszerekkel Leírja, hogyan illeszkedik a rendszer a szoftvert megrendelő vállalat átfogó üzleti vagy stratégiai céljai közé.

1. Szójegyzék

A szójegyzékben a leírásban használt szakkifejezéseket vázolhatjuk. Ezt fontos megtenni, mert az olvasó szakképzettségét nem tudjuk megbecsülni.

1. Felhasználói követelmények definíciói

Minden felhasználói és a nem funkcionális követelményt ebben a fejezetben célszerű vázolni. A leírásban alkalmazhatunk természetes nyelvet, diagramokat vagy egyéb, a megrendelő számára érthető jelöléseket. A követendő termék- és folyamatszabványokat is célszerű megadnunk.

1. A rendszer felépítése

A tervezett rendszerfelépítés magas szintű áttekintése, bemutatva, hogyan oszlanak el a funkciók a különböző rendszermodulok között. Az újrafelhasznált, architektúrális komponenseket ki kell emelnünk.

1. Rendszerkövetelmény specifikáció

A funkcionális és nem funkcionális követelmények részletesebb leírása Amennyiben szükséges, a nem funkcionális követelmények további részletekkel egészíthetők ki, például definiálhatunk interfészeket más rendszerekhez.

1. Rendszermodellek

Egy vagy több rendszermodellt ad meg, megmutatva a rendszerkomponensek, a rendszer és annak környezete közötti kapcsolatokat. Ezek lehetnek objektummodellek, adatfolyam modellek és szemantikus adatmodellek.

1. Rendszerevolúció

Leírja a rendszer alapját képező feltételezéseket, továbbá vázolja a hardver fejlesztése, a változó felhasználói igények stb. következtében bekövetkező előre látható változásokat.

1. Függelék

Részletes, specifikus információt kell nyújtani itt a fejlesztett alkalmazásról. A csatolt függelékekre példa a hardver- vagy az adatbázis-leírások. A hardverkövetelmények meghatározzák a rendszer minimális és optimális konfigurációját. Az adatbázis-követelmények megadják a rendszer által használt adatok logikai szerkezetét és a köztük lévő kapcsolatokat.

1. Tárgymutató

A dokumentum több tárgymutatót is tartalmazhat. Szerepelhet még a diagramok, funkciók indexe is.

A dokumentációnak tehát mindig egy egységes anyagnak kell lennie. A fejlesztési módszertan alapjaiban befolyásolhatja a dokumentum felépítését, illetve annak részletességét. Ha például egy fejlesztő cég az evolúciós modellt választja irányelvként egy termék kifejlesztésekor, akkor több javasolt részletes fejezet hiányozni fog. Ilyenkor a hangsúly inkább a felhasználói követelmények és a magas szintű, nem funkcionális rendszerkövetelmények irányába tolódik el.

Szintén a követelménydokumentum csak részleges kidolgozása mellett érvelnek az agilis fejlesztési módszertant alkalmazó fejlesztők csoportja. Véleményük szerint a követelmények olyan gyorsan változnak, hogy a követelmények dokumentuma már akkor elavult, amikor megírták, így az érte tett erőfeszítés javarészt hiábavaló volt.

Egyes megközelítések, mint az extrém programozás a dokumentumok gyártása helyett inkább azt javasolják, hogy a felhasználói követelményeket folyamatosan gyűjtsük, és írjuk kártyákra. Majd ezek után a felhasználó rangsorolja a követelményeket, azaz meghatározza, melyik legyen megvalósítva a következő rendszerfejlesztési iteráció során.

Mint azt már korábban érzékelhettük, egy elkészítendő szoftver esetén nagyon fontos, hogy kellő figyelmet fordítsunk a követelmények meghatározására, feltárására, és dokumentálására. Az ebben a fázisban vétett hibák, vagy pongyola megfogalmazások nagymértékben befolyásolják a fejlesztés későbbi menetét. Ezért a követelményeket tervezni kell, amely folyamatnak a célja a rendszerkövetelmények dokumentumának létrehozása és karbantartása. A teljes folyamat négy magas szintű tervezési folyamatot foglal magába:

1. a rendszer üzleti használatának felmérése (megvalósíthatósági tanulmány)

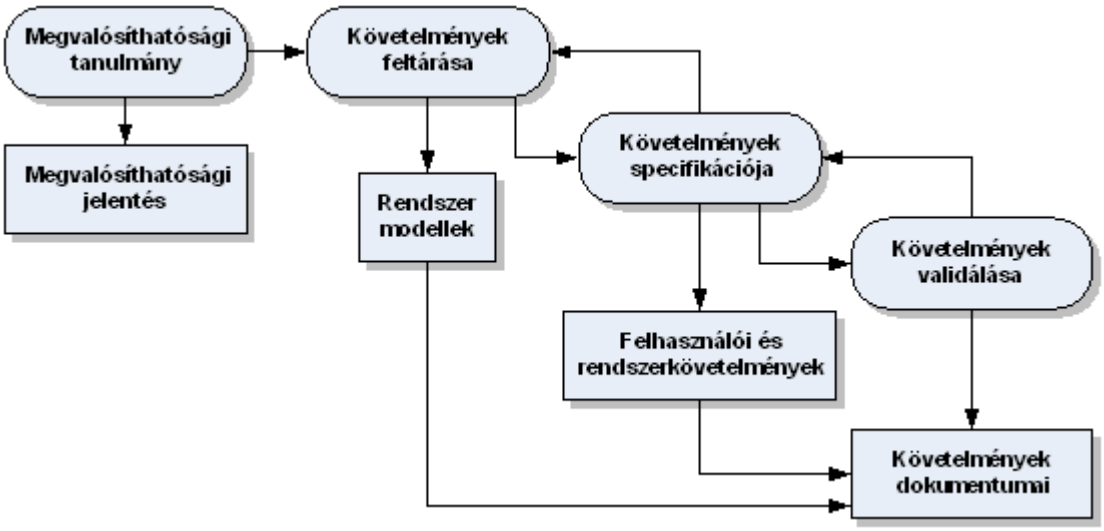
2. a követelmények feltárása és elemzése

3. a követelmények átalakítása valamilyen szabványos formátumra (specifikáció)

4. ellenőrzés, hogy a követelmények a megrendelő által kívánt rendszert definiálják-e (validálás)

A következő ábra a követelmények tervezésének folyamatát mutatja be.

5.4. ábra - A követelménytervezés folyamata



3.2. Megvalósíthatósági tanulmány

A követelménytervezési folyamat első lépése minden esetben egy megvalósíthatósági tanulmány elkészítése kell legyen. Az eredményeket majd egy jelentésben foglaljuk össze, melynek bemenetéül az üzleti követelmények kezdeti változatai, a rendszer körvonalazott leírása szolgál. A tanulmány egy rövid, tömör dokumentum, amely a következő kérdésekre próbál választ adni:

1. Támogatja-e a rendszer a vállalat általános célkitűzéseit

2. Megvalósítható-e a rendszer a jelenlegi technológiával adott költségen belül és adott ütemezés szerint?

3. Integrálható-e más, már használatban lévő rendszerekkel?

Az első kérdés, amely arra ad választ, hogy a rendszer hozzájárul-e az üzleti célokhoz nagyon kritikus, mert amennyiben nem járul hozzá, nincs valódi üzleti értéke a szoftvernek. A gyakorlatban ennek ellenére vannak olyan esetek, amikor vállalatok olyan rendszert fejlesztenek, ami nem járul hozzá a céljaikhoz. Ennek oka lehet az, hogy a célkitűzések nincsenek világosan megfogalmazva, vagy nem sikerült definiálni a rendszer követelményeket.

3.3. Követelmények feltárása és elemzése

A követelmények feltárása és elemzése a követelménytervezési folyamat második nagy részfeladata. E tevékenység során együtt kell működni a szoftverfejlesztőknek a megrendelőkkel és a végfelhasználókkal, azért, hogy kiderítsék milyen szolgáltatásokat kellene biztosítani a rendszernek. A folyamat során a tervezett vagy meglévő rendszerekről információkat gyűjtünk, szisztematikusan megvizsgáljuk azok működését, majd ebből kiszűrjük a felhasználói és rendszerkövetelményeket. Itt kerülnek kiválasztásra az úgynevezett kulcsfigurák, akik olyan személyek vagy csoportok, akiknek közvetett vagy közvetlen befolyása lehet a rendszerkövetelményekre. A feltárás és elemzés nagy odafigyelés igénylő folyamat, nehézségeinek több oka lehet:

1. A gazdasági és az üzleti környezet dinamikusan változhat, amely során új kulcsfigurák jelenhetnek meg, újabb követelményekkel.

2. Sok esetben a kulcsfigurák a követelmények meghatározásában félrevezetőek lehetnek, mert tudják pontosan, mit várnak el a számítógépes rendszertől. Bonyolult lehet számukra annak kifejezése, mit akarnak a rendszertől, vagy több esetben valóságtól elrugaszkodott kívánságaik lehetnek.

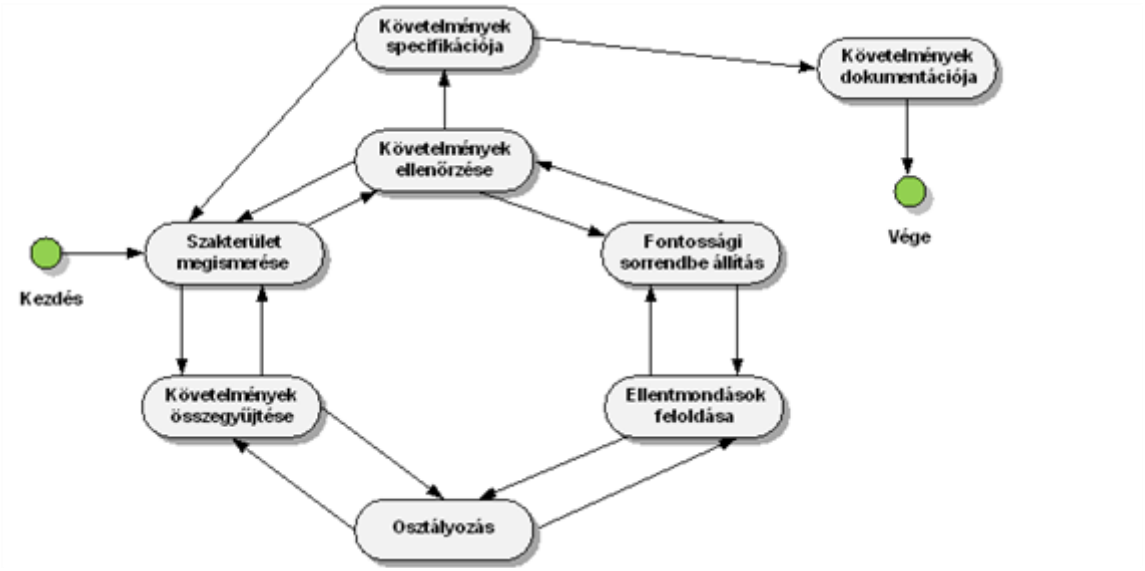
3. A rendszer kulcsfigurái a saját szakterületi fogalmaikkal fejtik ki a követelményeket.

4. Az egyes kulcsfiguráknak különböző követelményeik lehetnek. Ebben a követelménytervezőknek fel kell fedezniük a közös dolgokat és ellentmondásokat.

5. A rendszerkövetelményeket politikai tényezők is befolyásolhatják. Lehetnek olyan vezetők, akik azért igényelnek specifikus rendszerkövetelményeket, hogy ezzel növeljék a szervezeten belüli befolyásukat.

A feltárási és elemzési folyamat általános modellje:

5.5. ábra - A feltárás és elemzés általános modellje



A folyamat lépései rövid magyarázattal:

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 összefüggőcsoportokba szervezi.

4. Ellentmondások feloldása. Ahol több kulcsfigura is érintett, elkerülhetetlen, hogy a követelmények ellentmondásba 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 követelmények feltárásának és elemzésének támogatására több technika alakult ki az évek során. A következőkben három technikát tekintünk át röviden.

3.3.1. Nézőpont-orientált szemlélet

A nézőpont-orientált szemléletek a feltárási folyamatot, és magukat a követelményeket nézőpontok segítségével rendszerezik. A legfőbb erőssége az, hogy felismeri a többféle perspektívát, és eszközöket nyújt a követelmények ellentmondásainak felderítésére.

A nézőpontok a kulcsfigurák és egyéb követelményforrások egyfajta osztályozására használhatók. A nézőpontok típusai:

1. Interaktor nézőpontok: olyan személyek vagy rendszerek, amelyek a rendszerrel közvetlenül érintkeznek.

2. Közvetett nézőpontok: azon kulcsfigurák, akik nem használják ugyan közvetlenül a rendszert, de a követelményeket valahogyan befolyásolják. Például egy bank vezetősége és a bank biztonsági személyzete.

3. Szakterületi nézőpontok: a szakterület azon jellegzetességei és megszorításai, amelyek a rendszerkövetelményeket közvetlenül befolyásolják. Például a bankok közötti kommunikációra kifejlesztett szabványok.

A felsorolt nézőpont típusok a követelményeket jellemzően különbözőféleképpen értelmezik. Az interaktor nézőpontok részletes rendszerkövetelményeket írnak le, amelyek lefedik a rendszer eszközeit és interfészeit. A közvetett nézőpontok inkább magas szintű szervezeti követelményeket és megszorításokat közvetítenek, végül a szakterületi nézőpontok pedig rendszerint a rendszerre vonatkozó szakterületi megszorításokat nyújtják.

A követelményeket adó nézőpontok származhatnak még a vállalat marketing- és a külső kapcsolatokért felelős osztályáról is. Ez különösen igaz a web alapú rendszereknél, főleg az e-kereskedelmi rendszereknél és a dobozos szoftvertermékeknél. Ennek oka, hogy a web alapú rendszereknek kedvező képet is kell mutatniuk a vállalatról.

Nem triviális, komplex szoftverek esetén rengeteg lehetséges nézőpont van, és gyakorlatilag lehetetlen mindegyikből feltárni a követelményeket. Ilyenkor egy segítség lehet a nézőpontok rendszerezése és hierarchiába szervezése. Az azonos ágon lévő nézőpontoknak nagy valószínűséggel közösek lesznek a követelményeik.

3.3.2. Forgatókönyvek

A forgatókönyvek interakció-sorozat leírások, amelyek különösen akkor hasznosak, ha további részletekkel szeretnénk kiegészíteni a követelmények körvonalazott leírását. Minden forgatókönyv egy vagy több lehetséges interakciót takar. Számos forgatókönyvtípust kifejlesztettek már, ezek mind különböző részletezettségű, különböző típusú információkat nyújtanak a rendszerről.

Egy forgatókönyv készítése mindig az interakció körvonalazásával kezdődik. A feltárás alatt további részletekkel bővítjük, hogy végül az interakció teljes leírása megszülessen. Egy forgatókönyv általánosan a alábbiakat tartalmazhatja:

1. a rendszer, illetve a felhasználói elvárások;

2. a forgatókönyvbeli események normális menetének leírása;

3. leírást arról, mi romolhat el, és ezt hogyan kezeli a rendszer;

4. egyéb tevékenységek leírása, amelyek ugyanabban az időben mehetnek végbe;

5. a rendszer végállapotának leírását a forgatókönyv befejeződésekor.

A forgatókönyvek megfogalmazhatók szövegesen, kiegészíthetők diagramokkal, képernyőképekkel és így tovább. Másik lehetőségként alkalmazhatunk strukturáltabb megközelítést jelentő esemény-forgatókönyveket vagy használati eseteket. A használati esetek (use-case-ek) az UML jelölésrendszer részei.

3.3.3. Etnográfia

Az etnográfia egy olyan technika, amelynek célja a társadalmi és szervezeti követelmények jobb megértése a követelmény-feltárási folyamatban. Megfigyelésen alapul, miszerint az elemző elmélyed abban a munkakörnyezetben, ahol a rendszert majd használni fogják. Megfigyeli a napi munkát, és jegyzeteket készít az aktuális feladatokról.

Azért tartják fontosnak ezt a technikát, mert az emberek gyakran nehéznek találják kifejezni munkájuk részleteit, ez természetükből fakad. Sok esetben megértik azt a munkát amit végeznek, de azt gyakran nem, hogy ez a munka milyen összefüggésben áll a szervezet többi munkájával. Ezek a társadalmi és szervezeti tényezők viszont rögtön világossá válnak, ha egy kívülálló személy, egy tárgyilagos megfigyelő figyelemmel kíséri a folyamatokat.

Az etnográfiai elemzés során készített tanulmányok a folyamat azon kritikus részleteit tárhatják fel, amelyek más feltárási technikáknál gyakran elmaradnak. Ugyanakkor, mivel a hangsúly a végfelhasználón van, ez a megközelítés nem alkalmas a szervezeti vagy a szakterületi követelmények felderítésére. Az etnográfia ezért nem alkalmas a követelmények önálló, teljes értékű feltárására, célszerű más módszerekkel együtt alkalmazni. Jellegzetes példáként említhető a használati esetek elemzése.

4. Ellenőrző kérdések

1. Sorolja fel azokat az alapvető tevékenységeket, amelyek minden szoftverfolyamatban közösek.
2. Röviden vázolja mit értünk megvalósíthatósági tanulmány alatt. Milyen esetekben használjuk?
3. Miért van szükség a követelmények dokumentum két különböző részletezettségi szintű leírására?
4. Foglalja össze mi a nem funkcionális követelmények legnagyobb problémája.
5. Röviden vázolja mit értünk a követelmények ellentmondás-mentességének fogalmán.
6. Vázolja a követelmény dokumentum egy javasolt felépítését.
7. Ismertesse a követelmény feltárás és elemzés folyamatának lépéseit.
8. Mit értünk Forgatókönyvek alatt?
9. Soroljon fel legalább négy termékre vonatkozó funkcionális követelményt.
10. Mire szolgál a Tervleíró nyelv? Hol alkalmazzák?

6. fejezet - A szoftvertervezés folyamata

A szoftvertervezés a követelmények tisztázása és dokumentálása után a következő mérföldkőnek számító fontos tevékenység, amely fejlesztési modelltől függetlenül egységesen jelenik meg bármely szoftver fejlesztésénél. A fejezet célja, hogy egy általános áttekintést nyújtson a tervezési folyamat egészéről, az egyes fejlesztési részterületeken alkalmazható modellekről.

A szoftvertervezés lényegében egy emberi tényezőt igénylő olyan tervezési folyamat, amely során a szoftver logikai szerkezetére vonatkozóan döntéseket kell meghozni. Mikre is vonatkoznak valójában ezek a döntési kérdések? Röviden az alábbi pontokat kell megválaszolni a tervnek:

1. az implementálandó szoftver struktúrája

2. az adatok szervezése és áramlása a rendszerben,

3. a (rendszer)komponensek közötti interfészek tisztázása,

4. a használt algoritmusok leírása.

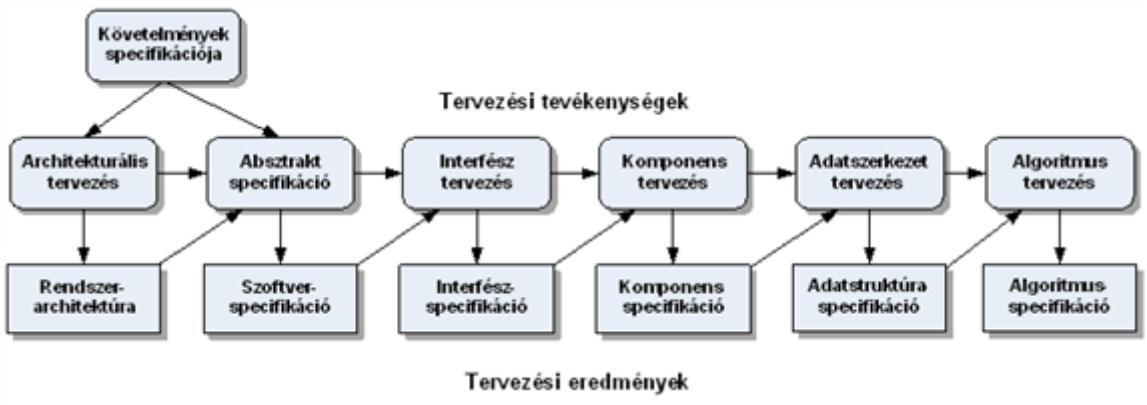
A tervezési folyamat „indítómotorja” az követelmény-feltárás és specifikációs fázis eredményeképpen előállított követelménydokumentum. A korábbi fejezetben már kihangsúlyoztuk eme fázis fontosságát. Nem véletlen, hiszen a félreértett vagy rosszul dokumentált követelmények itt, a tervezés folyamatában további problémákat generálhatnak. Ekkor a rendszer logikai szerkezetét a rossz követelményeknek megfelelően alakítjuk ki, amely pedig egy hibásan működő rendszert eredményez. Mindez pedig egy-egy komponens, vagy akár a teljes rendszer újratervezéséhez vezethet, amely jelentős idő és anyagi forrásokat emészthet fel.

A tervezési folyamat során a tervezők nem egyenes úton haladnak, hanem iteratív módon számos különféle verzió kifejlesztésén keresztül. Erre sokszor azért van szükség, mert a kifejlesztendő rendszer bizonyos részei kevésbé érthetők lehetnek (pl.: bizonyos szakterületi követelmények esetén), így úgynevezett prototípusok létrehozásával tapasztalatokat gyűjthetnek a készítendő rendszerről.

A tervezési folyamat járulékos formalitást és kifejtést is magában foglal a terv fejlesztése közben, valamint folytonos visszalépéseket a korábbi tervek javítására. Nem ritka azonban az sem, hogy a tervezés során különböző követelményproblémák (ellentmondás, többértelműség, stb) merülnek fel. Ilyenkor a specifikáció finomítása, a hibák korrigálása szükséges.

A tervezési folyamat számos különféle absztrakciós szinten levő rendszermodell kifejlesztését is tartalmazhatja. Amint a tervet részekre osztjuk, napvilágra kerülnek a korábbi hibák és hiányosságok. Ezek a visszacsatolások biztosítják, hogy továbbfejleszthessük a korábbi tervezési modelleket. A tervezési folyamat tevékenységei átfedik egymást. A következő ábra a tervezés folyamatát általánosságban mutatja be.

6.1. ábra - A tervezési folyamat általános modellje



Az ábrán jól megfigyelhetők a tervezési folyamat egyes állomásai, valamint az, hogy minden fázis mellett egy a tevékenységnek megfelelő specifikáció készül el. Ezek a dokumentumok az adott terv szintjének részletes leírásai. Ahogy folytatódik a tervezés folyamata, ezek a specifikációk egyre részletesebbé válnak. A folyamat végeredménye pedig az implementálandó algoritmusok és adatszerkezetek precíz specifikációja.

Nagyon fontos, hogy a folyamat során részletesen kitérjünk minden felmerülő kérdésre. Ebben az esetben az implementáció során nem kell meg nem válaszolt tervezési kérdésekkel foglalkozni. A tervezési folyamat tevékenységei röviden a következők:

1. Architekturális tervezés: a rendszert felépítő alrendszerek és a köztük található kapcsolatok azonosítása és dokumentálása.

2. Absztrakt specifikáció: meg kell adni az azonosított alrendszerek megszorításainak és szolgáltatásainak absztrakt specifikációját, melyek mellett azok működnek.

3. Interfész tervezése: meg kell tervezni és dokumentálni az egyes alrendszerek egyéb alrendszerek felé mutatott interfészeit. Az interfész specifikációjának egyértelműnek kell lennie.

4. Komponens tervezése: a szolgáltatásokat azokat megvalósító komponensekre kell bontani, és meg kell tervezni a komponensek interfészeit.

5. Adatszerkezet tervezése: meg kell határozni és részletesen meg kell tervezni a rendszer implementációjában használt adatszerkezeteket.

6. Algoritmus tervezése: meg kell tervezni és pontosan meg kell határozni a szolgáltatások biztosításához szükséges algoritmusokat.

Bizonyos esetekben a szoftver logikai szerkezetét modellező nyelvek segítségével célszerű szemléletesebbé és jobban érthetőbbé tenni. Ilyenkor egy modellezőnyelv segítségével alkotjuk meg a szoftver modelljét. Az irodalomban számos ilyen nyelv fejlődött ki, amelyek közül talán az UML (Unified Modelling Language) a legelterjedtebb a gyakorlatban. Más esetekben a tervet informális jelölésrendszerrel leírt vázlatokkal reprezentálhatjuk.

1. Architekturális tervezés

Az architekturális tervezés a legelső lépcsőfok a tervezés folyamatban. Ebben a lépésben meg kell próbálni a nagy rendszereket egymással kapcsolatban álló, egymásnak szolgáltatásokat nyújtani képes alrendszerek csoportjára bontani. Természetesen mindezt úgy, hogy a terv eleget tegyen a követelmények dokumentumában rögzített funkcionális és nem funkcionális követelményrendszernek. Architekturális tervezésen pedig azt a folyamatot értjük, amely ezen alrendszerek azonosítására, az alrendszer vezérlésére és a kommunikációjára szolgáló keretrendszer létrehozásával foglalkozik.

Egy fontos és kreatív folyamat, amely alapjában véve befolyásolja a rendszer teljesítményét, robusztusságát, eloszthatóságát és karbantarthatóságát. Biztosítani kell tehát azt, hogy a szoftvertervezők a tervezés kulcsfontosságú aspektusaival már a folyamat korai szakaszában foglalkozzanak. Egy alkalmazás számára választott szerkezet olyan nemfunkcionális rendszerkövetelményektől is függhet, mint például a teljesítmény, védettség, biztonságosság, rendelkezésre állás, és karbantarthatóság. Amennyiben ezeket a megkötéseket a tervezés eme fázisában nem vagy kevésbé vesszük figyelembe, a későbbi folyamatokban számos további probléma merülhet fel.

Az architekturális tervezés magában foglalja a rendszerek alrendszerekre bontását is. Az alrendszerek tervezése tulajdonképpen a rendszer durva szemcsézettségű komponensekre történő absztrakt felbontása, amely komponensek lehetnek önálló rendszerek. Az alrendszerek terveit általában blokkdiagram segítségével írjuk le.

1.1. Architekturális tervezési döntések

Az architekturális tervezési folyamat során a rendszer tervezőinek számos olyan döntést kell meghozniuk, amelyek alapvetően kihatnak a rendszerre és a fejlesztési folyamatra. Tudásuk és tapasztalataik alapján a következő alapvető kérdésekre kell válaszolniuk:

1. Létezik-e olyan általános alkalmazás architektúra, amely a tervezendő rendszer számára mintául szolgálhat?

2. Hogyan osztjuk szét a rendszert processzorokra?

3. Milyen architekturális stílus lenne a rendszer számára megfelelő?

4. Milyen alapvető megoldásokat alkalmazunk a rendszer strukturálására?

5. Hogyan lehet a rendszer szerkezeti egységeit modulokra felbontani?

6. Milyen stratégiát kell alkalmazni az egységek működésének vezérlésével kapcsolatban?

7. Hogyan értékelik majd ki az architekturális tervet?

8. Hogyan kell a rendszer-architektúrát dokumentálni?

Természetesen a kérdések és a válaszok köre nagymértékben függ a kifejlesztendő szoftver típusától. Egy egyszerű személyi számítógépes rendszer esetén például nincs szükség elosztott architektúrára.

Az architekturális tervezési folyamat eredménye egy architekturális tervezési dokumentum, amely tartalmazza a rendszer grafikus reprezentációit és a hozzájuk kapcsolódó leíró szöveget. Célszerű az általános logikai felépítést mindig valamilyen grafikus reprezentációval modellezni, mert az emberi agy vizualizált objektumokkal könnyebben dolgozik. Továbbá a dokumentumnak részletesen vázolnia kell azt, hogy a rendszert hogyan lehet alrendszerekre bontani és az egyes alrendszerek miképpen oszthatók modulokra.

Bár minden szoftver architektúrája valamilyen szinten egyedi, az alkalmazott megoldások általánosításával a következő modellcsoportok merülnek fel:

1. Statikus szerkezeti modell: megmutatja, hogyan lehet az alrendszereket és komponenseket különálló egységként fejleszteni.

2. Dinamikus folyamatmodell: ábrázolja, hogy a rendszer hogyan szervezhető futási idejű folyamatokba. Különbözhet a statikus modelltől.

3. Interfészmodell: az alrendszerek által nyújtott szolgáltatásait írja le.

4. Kapcsolatmodellek: az alrendszerek közötti kapcsolatokat (például adatáramlás) mutatják be.

5. Elosztási modell: azt adja meg, hogy az egyes alrendszereket hogyan kell elosztani a számítógépek között.

Azt, milyen eszközzel reprezentáljuk a rendszer-architektúráját, nincs megkötés. Az irodalomban azonban számos kutató az architektúra leíró nyelvek (ArchitecturalDescriptionLanguages, ADL) használatát és az UML nyelvet javasolja.

1.2. 6.1.2 A rendszer felépítése

Az architekturális tervezési folyamat korai szakaszában döntenünk kell a rendszer teljes szerkezeti modelljéről. Ezt szervezési mód az alrendszerek akár közvetlenül is tükrözhetik, de gyakran előfordul, hogy az alrendszer modellje részletesebb a szerkezeti modellnél.

A rendszer felépítését számos modell segítségével írhatjuk le. Bár minden rendszer architektúrája más lehet, vannak széles körben alkalmazott módszertanok. Most ezekből nézünk meg a három legáltalánosabbat.

1.2.1. 6.1.2.1 A tárolási modell

A tárolási modell alapvetően az adatok tárolási módjára helyezi a fő hangsúlyt az alapján, hogy a rendszert felépítő alrendszerek együttműködnek és információkat cserélnek egymás között. Ez az információcsere alapvetően két módon történhet:

1. Az adatotokat minden alrendszer által elérhető, központi adatbázisban helyezik el.

2. Minden alrendszer saját adatbázist tart fent. Ekkor az egyéb alrendszerek közötti adatcsere üzenetküldés segítségével történik.

A megosztott adatbázison alapuló rendszermodellt tárolási modellnek nevezzük. A nagy mennyiségű adatokkal dolgozó rendszerek mindig valamilyen osztott adatbázisok és tárolók köré szerveződnek. Ez a modell azon alkalmazások számára megfelelő, ahol az egyik alrendszerben keletkező adatok egy másik alrendszerben kerülnek felhasználásra.

A megosztott tárolók előnyei és hátrányai a következők:

1. Hatékony módszer nagy mennyiségű adat megosztására, mert nem szükséges az adatokat explicit módon átvinni egyik alrendszerből a másikba közvetlenül.

2. Tárolási modell esetén az alrendszerek azonos adattárolási modellel kell rendelkezniük. Ez rosszirányban befolyásolhatja a teljesítményt, mert kompromisszumokra van szükség az egyes eszközök között. A közös sémának nem megfelelő adatmodellel rendelkező új alrendszerek integrálása nehézzé vagy lehetetlenné válik.

3. Az adatokat előállító alrendszereknek foglalkozniuk kell azzal, hogy a többi alrendszer hogyan fogja használni azokat az adatokat.

4. Bizonyos tevékenységek (pl.: biztonsági mentés, a védelem, a hozzáférés szabályozás és a hiba utáni visszaállítás) központosítottak, így az eszközök lényeges feladataikra összpontosíthatnak.

5. A tárolási modell viszont ugyanazt a politikát kényszeríti rá minden alrendszerre.

6. A megosztottság modellje a tárolási sémán keresztül látható. Ez nagyban megkönnyíti az új eszközök integrálását.

7. A tároló több gép közötti elosztása bonyolult lehet. Bár megoldható egy logikailag központosított tároló elosztása, problémák léphetnek fel az adatok redundanciájával és inkonzisztenciájával kapcsolatban.

1.2.2. 6.1.2.2 A kliens-szerver modell

A kliens-szerver architektúra az egyik legkorábban kialakult rendszermodell. Alapgondolata egy klasszikus osztott rendszermodellje, amely megmutatja, hogy az adat és a feldolgozás hogyan oszlik meg feldolgozóegységek között. Három fő komponensből, a kliensek és szerverek halmazából, valamint a hálózatból tevődik össze:

1. Kliensek halmaza: olyan számítógépek, szoftverek, amelyek a szerverek által biztosított szolgáltatásokat igényelnek. Egy kliensprogramnak számos példánya futhat egyidejűleg.

2. Hálózat: lehetővé teszi, hogy a kliensek hozzáférjenek a szolgáltatásokhoz. Ez nem szükséges akkor, ha mind a kliensek, mind pedig a szerverek egyetlen gépen futnak.

3. Szerverek halmaza: más alrendszerek, számítógépek számára szolgáltatásokat nyújtanak. Ilyenek például a nyomtatószerverek.

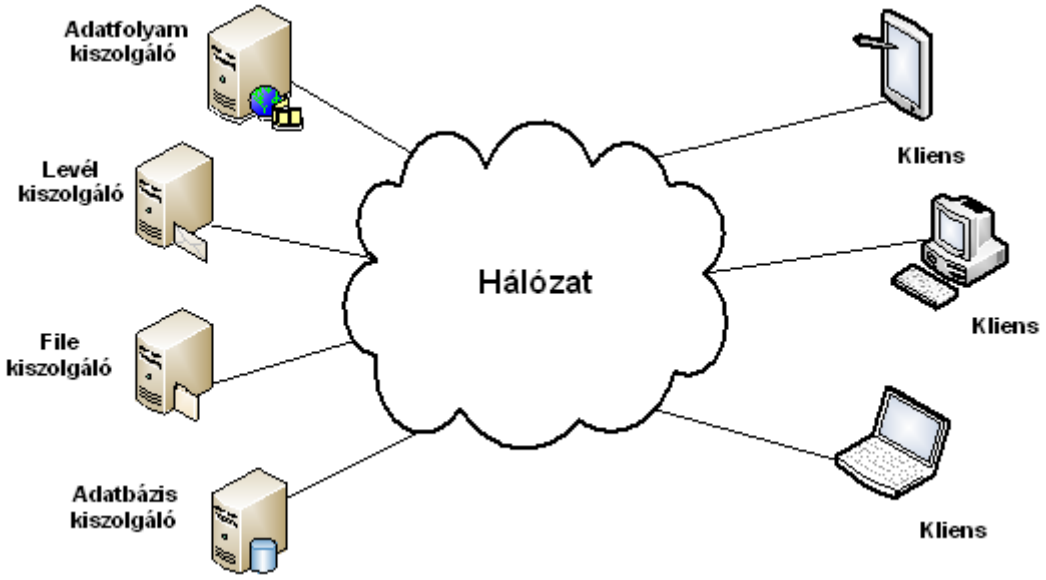
A modell jellegzetes tulajdonsága, hogy a klienseknek explicite ismerni kell az elérhető szerverek és az általuk biztosított szolgáltatások neveit. Csak így vehetik igénybe a szolgáltatásokat. A szervereknek azonban nem kell tudniuk sem a kliens azonosságát, sem pedig azt, hogy hány kliens van.

A kliensek a szerverek által biztosított szolgáltatásokat távoli eljáráshívásokkal érik el, egy kérés-válasz alapú protokoll segítségével, mint például a www esetén használt http protokoll. Lényegében a kliens elküld egy kérést a szervernek, és addig várakozik, amíg választ nem kap.

A kliens-szerver modell legfontosabb előnye osztott architektúrája. Leghatékonyabban igazán olyan környezetben használható, ahol sok klienssel, osztott egységgel rendelkező hálózattal dolgozunk. Egy új szerver könnyen hozzáadható a rendszerhez és integrálható annak többi részével.

A következő ábra egy példát mutat be a kliens-szerver architektúrára:

6.2. ábra - Általános kliens szerver architektúra

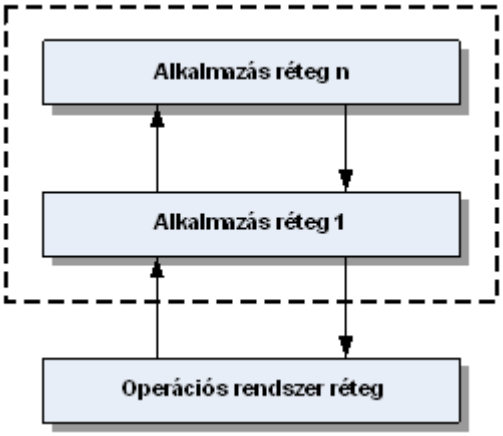


1.2.3. 6.1.2.3 Rétegzett modell

A rétegzett modell a rendszert egymástól jól elkülöníthető, alá-fölérendeltséggel rendelkező rétegekbe szervezi. Akkor beszélhetünk külön rétegről, ha az adott szint valamilyen szolgáltatást biztosít az alatta és felette lévő réteg számára. Az egyes szintek általános megközelítésben így egy absztrakt gépként értelmezhetők, amelynek gépi nyelvét a réteg által biztosított szolgáltatások definiálják. Az adott szint nyelve arra szolgál, hogy eltakarja az alatta lévő szintek megvalósításait és alapul szolgáljon az absztrakt gép következő szintjének megvalósításához. Példaként jól említhető a hálózati protokollok OSI-referenciamodellje.

A következő ábra a rétegzett modell általános felépítését mutatja.

6.3. ábra - Rétegzett rendszerek



A rétegalapú megközelítés egyik előnye, hogy jól összeköthető az inkrementális fejlesztés gondolatmenetével. Amennyiben egy réteg elkészült, az általa biztosított szolgáltatások elérhetővé tehetők a megrendelő vagy a felhasználók számára. További előnye, hogy mivel a rétegek jól elkülöníthetők egymástól, melyek interfésze jól definiált, így egy réteg könnyedén helyettesíthető egy másik, azzal ekvivalens réteggel, ha interfésze nem változik meg. Még az sem okoz nagy problémát, ha egy réteg interfésze megváltozik vagy új szolgáltatásokkal bővítették, mert a módosítások csakis szomszédos réteget érintik. A rétegzett modell csak egy ponton, egy rétegen keresztül tartja a kapcsolatot az operációs rendszerrel. Ez a legalsó réteg, ahol a gép és operációs rendszerfüggő részek elhelyezkednek. Változás esetén csak ebben a belső, gépfüggő rétegben kell módosítani, hogy figyelembe vegyük egy másik operációs rendszer vagy adatbázis lehetőségeit.

A rétegzett megközelítés hátránya, hogy ily módon bizonyos rendszerek strukturálása igen bonyolulttá is válhat. Az alapvető adottságokat (pl.: fájlkezelés), amelyre minden absztrakt gépnek szüksége van a belső rétegek biztosíthatják. A külső réteg felhasználó által igényelt szolgáltatásainak le kell egészen jutniuk több szomszédos

rétegen keresztül, hogy hozzáférjenek a több réteggel alattuk elhelyezkedő réteg szolgáltatásaihoz. Ez ellentmond a modell alapgondolatával, mert így egy külső réteg a továbbiakban már nem csupán egyszerűen a közvetlen megelőzőjétől függ.

1.2.4. Moduláris felbontás

Amennyiben döntés született a rendszer átfogó architektúrájáról, meg kell határozni, hogy az alrendszerek modulokra bontása során milyen megközelítést célszerű alkalmazni. Azt, hogy mit ért a szakirodalom modul kifejezés alatt nem egységes, és nem is különböztethető meg egyértelműen az alrendszer fogalmától. Egy szokásos megkülönböztetés a következő lehet:

1. Alrendszer: önálló rendszer, működése nem függ más alrendszerek szolgáltatásaitól. Az alrendszerek modulokból épülnek fel.

2. Modul: rendszerint egyszerű rendszerkomponens, amely más modulok számára szolgáltatás(oka)t biztosít.

Az alrendszerek modulokra bontása során két fő stratégia ismeretes. Egyik széles körben ismert megoldás az objektumorientált felbontás, amikor is a rendszert egymással kommunikáló objektumok halmazára bontjuk fel. Egy kevésbé ismert megoldás a funkcióorientált csővezetékek használata, amely során a rendszert bemenő adatokat elfogadó és azokat kimenő adatokká alakító funkcionális transzformációkra bontjuk fel.

1.2.4.1. Objektumorientált felbontás

Egy objektumorientált architekturális modell a rendszert lazán kapcsolódó, jól definiált interfészekkel rendelkező objektumok halmazára tagolja. Az objektumok szolgáltatásokat biztosítanak más objektumok részére. Az objektumorientált felbontás objektumosztályokkal, azok attribútumaival és műveleteivel foglalkozik, hogy hogyan építhető fel a rendszer ezen komponensekből. A szoftver implementációjakor az objektumok ezekből az osztályokból jönnek létre, és az objektum műveleteinek koordinálásához valamilyen vezérlési modellt alkalmaznak.

Az objektumorientált megközelítés előnye, hogy az objektumok lazán kapcsolódnak, így az objektumok implementációja változtatható anélkül, hogy az hatással lenne más objektumokra. A komponensek közvetlen implementációjának biztosítására objektumorientált programozási nyelveket fejlesztettek ki.

Azonban az objektumorientált megközelítésnek vannak hátránya: az objektumoknak explicit módon kell hivatkoznia a többi objektum nevére és interfészére. Ha a rendszerben interfész változtatás történik, akkor a változtatást minden, a megváltozott objektumot használó helyen át kell vezetni.

1.2.4.2. Funkcionált csővezeték

A funkcióorientált csővezetéket más néven adatfolyam-modellnek is nevezik. Alapgondolata az, a rendszer moduljai között adatok áramlanak (főként egy irányban). Minden modul bemenetét funkcionális transzformációk dolgozzák fel, majd hoznak létre kimeneteket. A modulok egymásutánisága egy transzformáció sorozatot alkot, ahol minden feldolgozási lépést egy külön transzformáció valósít meg. A rendszer adatok ezeken a transzformációkon keresztül menve a végleges kimeneti adatokká alakulnak. A transzformációk mind szekvenciálisan, mind pedig párhuzamosan végrehajthatók. Az adatok pedig egyesével vagy kötegelve is feldolgozhatók. Példaként említhetjük a Unix rendszerekben alkalmazott csővezetéket, ahol a parancsok jelentik az egyes funkcionális transzformációkat.

A modellt régóta alkalmazzák a számítástechnikában, főleg a korai években főleg automatikus feldolgozásokra. Interaktív rendszerek készítésére nem ajánlott, nehéz csővezetékelvű modell alapján megírni.

Előszeretettel alkalmazzák, mert támogatja a transzformációk újrafelhasználhatóságát, könnyen érthető és bővíthető új transzformációkkal, implementálható konkurens és szekvenciális környezetben egyaránt. Hátránya viszont az, hogy az egyes transzformációknak vagy egyeztetniük kell egymással az átadandó adatok formátumát, vagy a kommunikációban részt vevő adatok számára egy szabványos formátumot kell kialakítani.

1.3. Vezérlési stílusok

Az alrendszerek és modulok logikai struktúrájának kialakítása után szükség van a vezérlés menetének meghatározására. A vezérlés biztosítja, hogy az alrendszerek szolgáltatásai a megfelelő helyre a megfelelő

időben eljussanak. Mivel a strukturális modellek nem tartalmaznak ilyen elemeket ezért a rendszer kiépítőjének az alrendszereket valamilyen vezérlési modellnek megfelelően kell szerveznie. A vezérlési modellek az alrendszerek közötti vezérlési folyamatokkal foglalkoznak.

A szoftverrendszerekben két általános vezérlési stílust alkalmaznak:

1. Központosított vezérlés: létezik egy központi vezérlő alrendszer, amely a vezérlés teljes felelősségét ellátja. Beindítja és leállítja a többi alrendszert.

2. Eseményalapú vezérlés: a vezérlést nem egyetlen egy alrendszer végzi központosítottan, hanem minden alrendszer válaszolhat egy külsőleg létrejött eseményre. Ezek az események vagy más alrendszerekből, vagy a rendszer környezetéből származhatnak.

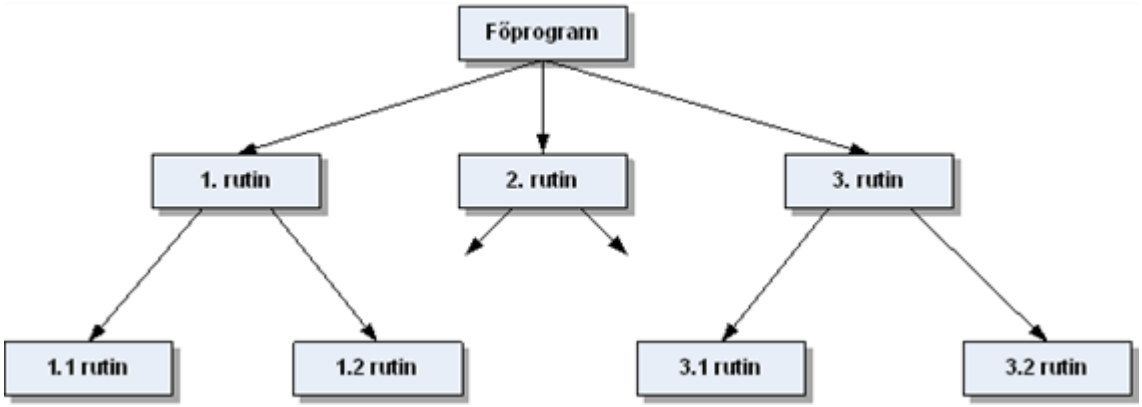
A vezérlési stílusok kiegészítik a strukturális stílusokat. Minden korábban bemutatott strukturális stílust meg lehet valósítani központosított és eseményalapú vezérléssel egyaránt.

1.3.1. 6.1.3.1 Központosított vezérlés

A modell alapja egy kitüntetett alrendszer, a rendszervezérlő, amely a többi alrendszer végrehajtásáért felelős. Ő irányít mindent, a többi alrendszer indítását, leállítását, és az információ áramlását is. A központosított vezérlési modellek két csoportba oszthatók attól függően, hogy a vezérelt alrendszerek szekvenciálisan vagy párhuzamosan hajtódnak-e végre.

1. A hívás-visszatérés modell: a megszokott fentről le alprogram modellje, ahol a vezérlés az alprogram-hierarchia csúcsán kezdődik, és alprogramhívások segítségével jut el a fa alsóbb szintjeire. Ez a modell csak szekvenciális rendszerek esetén alkalmazható.

6.4. ábra - A hívás-visszatérés modell



A vezérlési hierarchia egy fastruktúrát alkot, amelyben vezérlés menete a magasabb szintű rutintól jut el az alacsonyabb szinten lévőhöz, majd visszatér a hívás helyére. A hívás-visszatérés modell jól alkalmazható modulszinten a tevékenységek és objektumok vezérlésére, mert sok objektumorientált rendszerben az objektumok műveletei (metódusok) eljárásként vagy függvényként vannak implementálva.

A modell viszonylag merev és korlátozott. Ez azonban nemcsak hátránya, hanem egyben erőssége is. Erőssége abban rejlik, hogy a rendszer válasza viszonylag egyszerűen kiszámítható a bemenetek függvényében. Gyakorlatilag a fa ágain lefelé a rendszer válasza felderíthető. Hátránya pedig az, hogy a normálistól eltérő működés esetén használata kényelmetlen lehet.

2. A kezelőmodell: ez a modell alkalmazható konkurens és szekvenciális rendszerek esetén is. Egy kijelölt rendszerkezelő rendszerkomponens irányítja a többi rendszerfolyamat indítását, leállítását, valamint koordinálja azokat. A folyamat olyan alrendszer vagy modul, amely végrehajtható más folyamatokkal párhuzamosan.

   A rendszert központi vezérlő folyamata folyamatosan felügyeli a rendszer állapotát. A rendszer állapotváltozói alapján eldönti, hogy az egyes folyamatokat mikor kell elindítani, illetve leállítani. A vezérlő általában ciklikusan ellenőrzi az érzékelőket és folyamatokat, hogy bekövetkezett-e valamilyen esemény

vagy állapotváltozás. Ellenőrzi, hogy a többi folyamat hozott-e létre feldolgozandó vagy feldolgozásra továbbküldendő információt. Ezt a modellt más néven eseményciklus-modellnek is szokás nevezni.

A rendszert központi vezérlő folyamata folyamatosan felügyeli a rendszer állapotát. A rendszer állapotváltozói alapján eldönti, hogy az egyes folyamatokat mikor kell elindítani, illetve leállítani. A vezérlő általában ciklikusan ellenőrzi az érzékelőket és folyamatokat, hogy bekövetkezett-e valamilyen esemény vagy állapotváltozás. Ellenőrzi, hogy a többi folyamat hozott-e létre feldolgozandó vagy feldolgozásra továbbküldendő információt. Ezt a modellt más néven eseményciklus-modellnek is szokás nevezni.

1.3.2. 6.1.3.2 Eseményvezérelt rendszerek

Az eseményvezérelt vezérlési modelleket külsőleg bekövetkezett események irányítják. Az eseményvezérelt rendszereknek sok fajtája ismeretes, beleértve a szerkesztőket, ahol a szerkesztőutasításokat felhasználói felület események jelzik. A továbbiakban két eseményvezérelt vezérlési modellt mutatunk be:

1. Eseményszóró modellek: ezekben a modellekben egy esemény minden alrendszerhez eljut, és bármelyik, az esemény kezelésére programozott alrendszer reagálhat rá. A vezérlési politika nincs beépítve az esemény- és üzenetkezelőbe. Az alrendszerek eldöntik, hogy mely eseményekre tartanak igényt, az esemény- és üzenetkezelő pedig biztosítja, hogy ezek az események eljussanak hozzájuk.

2. Megszakítás vezérelt modellek: ezeket kizárólag olyan valós idejű rendszerekben használják, ahol egy megszakítás-kezelő észleli a külső megszakításokat. Ezek aztán valamely más komponenshez kerülnek feldolgozásra.

2. Objektumorientált tervezés

Egy objektumorientált rendszer egymással együttműködő objektumokból áll, amely az objektum saját állapotát karbantartja és erről az állapotról információs műveleteket biztosít. Az állapot reprezentációja privát, az objektumon kívülről közvetlenül nem hozzáférhető. Egy objektumorientált tervezési folyamat az objektum-osztályoknak és az azok közötti kapcsolatoknak a megtervezéséből áll.

Az objektumorientált tervezés az objektumorientált fejlesztés része, amelyben a fejlesztési folyamat során objektumorientált stratégiát használunk:

1. Objektumorientált elemzés:a szoftver objektumorientált modelljének kialakításával foglalkozik.

1. Objektumorientált tervezés: a meghatározott követelményeknek megfelelő szoftverrendszer objektumorientált modelljének kialakítása. Az objektumorientált tervezés objektumai a probléma megoldásával kapcsolatosak.

1. Objektumorientált programozás: a szoftverterv objektumorientált programozási nyelven történő megvalósítása. Pl.: C++, D, Java, C#, stb.

A különböző lépések közötti átmenetnek észrevehetetlennek kell lennie, és mindegyik lépésben kompatibilis jelölésrendszert kell használni. Az objektumorientált rendszereket a más elven fejlesztett rendszerekkel szemben könnyebb megváltoztatni, mert az objektumok egymástól függetlenek. Egy objektum implementációjának megváltozása vagy új szolgáltatásokkal történő bővülése nem befolyásolhatja a rendszer többi objektumát. Ezért azt mondhatjuk, hogy az objektumok potenciálisan újrafelhasználható komponensek, mivel az állapotnak és a műveleteknek független egységbe zárásai. Ez növeli az érthetőséget és így a terv karbantarthatóságát is.

2.1. Objektumok és objektumosztályok

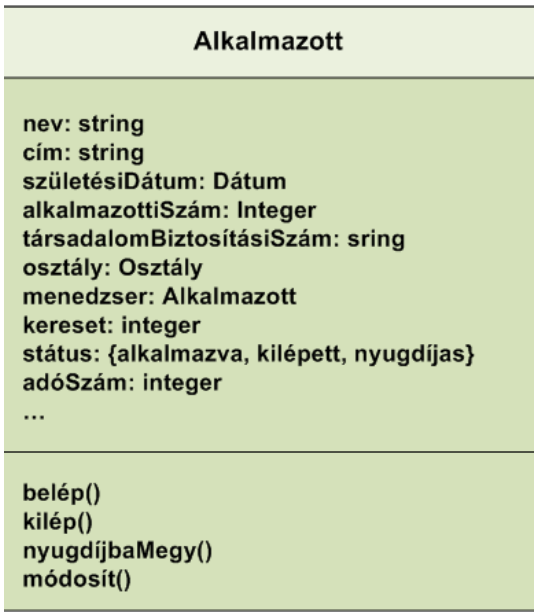
Általánosan elfogadott, hogy az objektum az információt elrejti. Ian Sommerville a következő definíciót adja [1]:

„Egy objektum egy állapottal és az ezen az állapoton ható, meghatározott műveletekkel rendelkező entitás. Az állapotot objektum-attribútumok halmazaként adjuk meg. Az objektum műveletei szolgáltatásokat biztosítanak a többi objektum számára.”

„Az objektumok egy objektumosztály-definíció alapján jönnek létre. Egy objektumosztály definíciója egyszerre típusspecifikáció és egy objektumok létrehozására szolgáló sablon. Az adott osztályba tartozó objektummal kapcsolatos összes attribútum és művelet deklarációját tartalmazza.”

A következő ábra [1] egy UML-ben megadott osztálydefiníciót mutat be:

6.5. ábra - Az Alkalmazott objektumosztály



Az objektumok úgy kommunikálnak, hogy szolgáltatásokat kérnek más objektumoktól (meghívják azok módszereit). A szolgáltatás végrehajtásához szükséges információ és a szolgáltatás végrehajtásának eredményei paraméterként adódnak át. Pl.:

```
// Egy puffer objektum módszerének hívása, amely visszaadja a pufferben található következőértéket

v = Buffer.GetNextElement();

// Puffer elemszámának a beállítása

v = Buffer.SetElements(20);
```

Egy objektum „szolgáltatáskérés” üzenetet küldhet egy másiknak, akitől a szolgáltatást kéri. A fogadóobjektum elemzi az üzenetet, azonosítja a szolgáltatást és az ahhoz kapcsolódó adatokat, majd végrehajtja a kívánt szolgáltatást. Ha a szolgáltatáskérelmek ily módon vannak implementálva, az objektumok közötti kommunikáció szinkron, azaz a hívóobjektum megvárja a szolgáltatás befejeződését (soros végrehajtás). A gyakorlatban a legtöbb objektum-orientált nyelvben ez a modell az alapértelmezett.

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). Ezek a konkurens objektumok kétféleképpen implementálhatók:

1. Aktív objektumok: önmaguk képesek belsőállapotukat 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.

Az objektumosztályok egy generalizációs vagy öröklődési hierarchiába szervezhetők, amely az általános és a specifikus objektumosztályok közötti kapcsolatot jeleníti meg. Egy objektumhierarchiában a lejjebb található osztályok rendelkeznek ugyanazokkal az attribútumokkal és műveletekkel, mint szülőosztályaik, de új attribútumokkal és műveletekkel egészítheti ki azokat, valamint meg is változtathatják szülőosztályaik attribútumainak és műveleteinek némelyikét.

2.2. Objektumok élettartalma

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.

3. Felhasználói felületek tervezése

A különböző számítógépes rendszerek tervezési kérdései között a felhasználói felületek tervezési problémái szintén nagy hangsúlyt kell kapjon. Egy jól áttekinthető, gondos felhasználói felület megtervezése fontos része a teljes szoftvertervezési folyamatának. Általában a felhasználói felület az elsődleges kapcsolódási pont a felhasználó és a számítógép között, ezért fontos, hogy megfelelő színvonalon legyen kidolgozva.

Természetesen az nem csak az esztétikus kinézet a cél, hiszen az egy szubjektív fogalom, de vannak olyan alapvető tervezési elvek, melyek figyelembevételével jobb eredmény érhető el. A jó felhasználói felület terve kritikus a rendszer megbízhatósága szempontjából. Célszerű a kialakítandó felhasználói felület illeszteni a felhasználó szakértelméhez, tapasztalatához és elvárásaihoz. A felület félretervezése mindig következményeket von maga után. Például a felhasználó esetleg nem fér hozzá a rendszer bizonyos jellemzőihez, hibázik, és úgy érzi, hogy a rendszer segítségnyújtás helyett inkább gátolja őt annak a célnak az elérésében, amiért a rendszert használja.

A mai modern szoftverek világában egy felhasználói felületnek számos megjelenítési formája lehet. Beszélhetünk egyszerű ablakos vastagkliens alkalmazásokról, web-es rendszerekről, mobiltelefonok, PDA-k megjelenítési megoldásairól, és nem utolsó sorban a pixelgrafikus alkalmazásokról. Bármilyen feladat is áll a tervező előtt a döntési során figyelembe kell venni a szoftvert használó emberek fizikai és mentális képességeit.

1. Az emberek korlátozott rövid távú memóriával rendelkeznek. Ha egyidejűleg túl sok információval terheljük meg a felhasználót, elképzelhető, hogy nem tudja azokat befogadni.

2. Ha a rendszer elromlik, és figyelmeztető, valamint hibaüzeneteket ad, akkor a felhasználók általában feszültebbé válnak, ami megnöveli a hibázás esélyét.

3. Nem szabad saját képességeinkre tervezni a rendszert, és feltételezni, hogy majd minden felhasználó képes lesz megbirkózni a felülettel. Fizikai képességeink (látás, hallás, érzékelés, kézügyesség)eltérhetnek.

4. Különböző interakciós módokat részesítünk előnyben. Némelyek képekkel, mások szövegekkel szeretnek dolgozni.

3.1. Felhasználói felületek tervezési elvei

Ha a felhasználói felületek alapvető tervezési elveit szeretnénk összegyűjteni és megfogalmazni, akkor az emberi képességek alapján kell elindulnunk. Ezek olyan általános elvek, amelyek minden felhasználói felület

tervére alkalmazhatók, és ezekből speciális szervezetek vagy rendszertípusok számára részletesebb tervezési irányelvek származtathatók.

Felhasználói jártasság: a felületnek olyan kifejezéseket és fogalmakat kell használnia, amelyek megfelelnek a rendszert legtöbbet használó emberek tapasztalatainak. Nem szabad egy felületet csak azért ráerőltetni a felhasználókra, mert az kényelmesen implementálható.

Konzisztencia: a felületnek lehetőség szerint a hasonló műveleteket hasonló módon kell realizálnia. Azt jelenti, hogy a rendszer parancsainak és menüinek ugyanazzal a formátummal kell rendelkezniük, a paramétereket minden parancshoz ugyanúgy kell átadni, és hasonlónak kell lennie a parancsok formájának.

Minimális meglepetés: a rendszer soha ne okozzon meglepetést a felhasználóknak, mert nagyon ingerültté válnak, ha a rendszer nem várt módon viselkedik. A rendszert használva a felhasználók felépítik magukban a rendszer működésének gondolati modelljét. Amennyiben valamely része a szoftvernek eltérő gondolatmenettel működik, a felhasználó meglepetté és zavarttá válik.

Visszaállíthatóság: a felületnek rendelkeznie kell olyan mechanizmusokkal, amelyek lehetővé teszik a felhasználók számára a hiba után történő visszaállítást. Ezen lehetőségeknek három fajtája lehetséges:

1. Az ártalmas tevékenységek megerősítése. Ha a felhasználók potenciálisan ártalmas tevékenységet hajtanak végre, az információ megsemmisítése előtt megerősítést kell tőlük kérni.

2. Visszavonási lehetőség biztosítása. A visszavonás a rendszert visszaállítja a tevékenység bekövetkezése előtti állapotba.

3. Ellenőrző pontok készítése. Célszerű a rendszer állapotát bizonyos időközönként elmenteni, így lehetővé téve, hogy azt a legutóbbi ellenőrző ponttól újra tudjuk indítani.

Felhasználói útmutatás: a felületnek hiba bekövetkezése esetén értelmes visszacsatolást kell biztosítania és környezet-érzékeny felhasználói súgóval is rendelkeznie kell. Ezeknek be kell épülni a rendszerbe, és különböző szintű segítséget és tanácsot kell adni.

Felhasználói sokféleség: a felületnek megfelelő interakciós lehetőségekkel kell rendelkeznie a rendszer különféle felhasználói számára. A rendszernek lehetnek alkalmi és rendszeres felhasználói.

3.2. Felhasználói felületek tervezési folyamata

A tervezés során a felhasználók együttműködnek a tervezőkkel, és prototípusokat készítenek a rendszer felületéről. Az elkészült prototípusok segítségével döntenek a rendszer felhasználói felületének jellemzőiről, felépítéséről és kinézetéről, vonnak le következtetéseket a rendszer felhasználói interakciójának működésről.

A felhasználói felületek tervezését éppen ezért célszerű iteratív úton fejleszteni. A felület prototípusa különállóan is elkészíthető, párhuzamosan más szoftvertervezési tevékenységekkel, amely gyorsítja a rendszer kifejlesztését.

A felületek tervezési folyamatát alapvetően a következő három szakaszra bontva tárgyalják:

1. Felhasználók elemzése: meg kell vizsgálni, hogy a felhasználók milyen feladatokat végeznek, milyen munkakörnyezetben dolgoznak, milyen egyéb rendszereket használnak, munkájuk során.

2. Prototípus-készítés: a felhasználói felület terve alapján prototípust kell készíteni.

3. Felületek értékelése: a prototípus formálisabb értékelése. A felhasználók interfészhasználat közben szerzett aktuális tapasztalatait gyűjtjük össze.

3.2.1. 6.3.2.1 Felhasználók elemzése

A felhasználói felület tervezési folyamatának alapgondolata, hogy elemezni kell azon felhasználói tevékenységeket, amelyeket a rendszernek támogatnia kell. Meg kell értenünk a szoftver, és a felületek pontos célját.

A felhasználók elemzésére három alapvető lehetőségünk van: a feladatelemzés, az interjúztatás vagy kérdőívek kitöltetése, valamint az etnográfia. A feladatelemzés és az interjúztatás az egyénre és az egyén munkájára

összpontosít, míg az etnográfia szélesebb szemszögből vizsgálja azt, hogy az emberek hogyan kommunikálnak és hatnak egymásra, hogyan rendezik el munkakörülményeiket, és miként működnek együtt feladatok megoldásában.

A feladatelemzésnek különböző formái vannak, de legszélesebb körben a hierarchikus feladatelemzést (Hierarchical Task Analysis, HTA) használják. A HTA-t eredetileg a felhasználói kézikönyvek készítésének elősegítésére hozták létre, de használható annak meghatározására is, hogy mit kell a felhasználóknak tenniük egy bizonyos cél elérése érdekében. A HTA-ban a magas szintű feladatot részfeladatokra bontjuk, és terveket készítünk, hogy megadjuk, hogy egy adott helyzetben mi történhet. A felhasználó céljából kiindulva egy hierarchiát készítünk, amely leírja, hogy mit kell tennünk a cél érdekében. A HTA jelölésrendszerében a doboz alatti vonal azt jelöli, hogy az adott feladatot nem bontjuk részfeladatokra.

A HTA előnye a természetes nyelvi forgatókönyvekkel szemben, hogy rákényszerít bennünket arra, hogy minden feladatot átgondoljunk, és eldöntsük, hogy felbontható-e, avagy sem. A természetes nyelvi forgatókönyvek használata során könnyen kimaradhatnak fontos feladatok, emellett a forgatókönyvek hosszúvá és unalmassá válhatnak, ha sok részletet szeretnénk beleírni.

A HTA legnagyobb problémája azonban, hogy leginkább csak a szekvenciális folyamatokként leírható feladatokra alkalmazható. Ha konkurens tevékenységeket szeretnénk leírni vele, akkor a jelölésmód kényelmetlenné válik.

Az interjúztatás célja, hogy a HTA-hoz kiegészítő információkat gyűjtsünk be a felhasználóktól. A cél az, hogy megtudjuk a felhasználók valójában mit is csinálnak. Az interjúkat úgy kell megtervezni, hogy a felhasználók minden, általuk szükségesnek tartott információt közölhessenek. Ez azt jelenti, hogy nem szabad mereven ragaszkodnunk az előre elkészített kérdésekhez, hanem a kérdéseknek nyitottnak kell lenniük, és arra kell serkenteniük a felhasználókat, hogy elmondják, mit és miért tesznek.

A tervezés során nemcsak az információk begyűjtése a fontos, hanem azok valamilyen formában történő dokumentálása is. Meg kell találni a módját annak, hogy úgy írják le a felhasználói elemzéseket, hogy utána az alapján a feladatok lényegét mind a többi tervező, mind pedig maguk a felhasználók felé kommunikálni tudják. A leírásra nincs követendő egyezményes eszköz, de egy hatékony megoldást kínál az UML szekvencia diagramja.

3.2.2. 6.3.2.2 Prototípus készítése

A felhasználói felületek mindig valamilyen dinamikus folyamatot reprezentálnak, ezért a felületekkel szemben támasztott követelmények kifejezésére a szöveges leírások és a diagramok nem elég jók. Nem képesek kifejezni és leírni a folyamatot teljes egészében. Csakis a végfelhasználó bevonásával végzett prototípus-készítés az egyetlen hatékony módja a grafikus felhasználói felületének tervezésének és fejlesztésének.

A prototípus-készítés célja az, hogy egy elkészült példafelület segítségével a tervezők tapasztalatokat szerezzenek a működéssel kapcsolatosan. Nem könnyű absztrakt módon gondolkodni egy felhasználói felületről, és pontosan elmagyarázni, hogy mit szeretnénk. Ha viszont egy példán keresztül sikerül bemutatni, akkor könnyen meg tudjuk mondani, hogy mely jellemzőket szeretjük, és melyeket nem.

Ideális esetben a prototípus-készítés során a folyamatot két lépésben hajtjuk végre:

1. A folyamat elején papíralapú prototípusokat - a képernyőtervek makettjeit - készítünk, és ezeket áttekintjük a végfelhasználókkal.

2. Finomítjuk a tervet, és egyre kifinomultabb automatizált prototípusokat fejlesztünk, amelyeket tesztelésre és a tevékenységek szimulálása érdekében odaadunk a felhasználóknak.

A papíralapú prototípus-készítés előnye, hogy igen olcsó és meglehetősen hatékony. Nem kell futtatható programot készítenünk, és a terveket sem kell profi módon megrajzolni. Csupán a rendszer azon képernyőit kell papírra lerajzolni, amelyeken kapcsolatba lép a rendszerrel. Emellett forgatókönyveket is létrehozunk, amelyek azt tartalmazzák, hogy a rendszer miképpen használható.

A papíralapú prototípus kezdeti tapasztalatai után a felület tervének szoftveres prototípusát kell implementálnunk. Az elkészítésre az alábbi három lehetőségünk van:

1. Szkriptvezérelt megközelítés: vizuális elemeket (gombokat, menüket stb.) tartalmazó képernyőket hozunk létre valamilyen szerkesztő eszköz segítségével, és ezekhez szkripteket társítunk. A felhasználói interakció során a szkript végrehajtásra kerül, és megjelenik a következő képernyő, amely a korábbi tevékenységek eredményeit mutatja be.

2. Vizuális programozási nyelvek: valamilyen vizuális programozási nyelv segítségével gyorsan létrehozhatunk felületeket, amely objektumaihoz komponenseket és szkripteket társíthatunk.

3. Internetalapú prototípus-készítés: ezek a megoldások egy web böngészőn és egy nyelven alapulnak. A funkcionalitást ekkor a nyelv nyújtja (pl.: Java). Ezek a kódrészletek (appletek) az oldal böngészőbe töltésekor automatikusan végrehajtódnak.

3.2.3. 6.3.2.3 Prototípus értékelése

A felületek értékelésén azt értjük, amikor megvizsgáljuk, hogy mennyire használható az adott felület, majd ellenőrizzük, hogy megfelel-e a felhasználói követelményeknek. Az ellenőrzési folyamat ugyanolyan fontos, mint a tervezés, vagy a prototípus készítése, semmiféleképpen sem elhanyagolandó.

Az értékelés valamilyen szinten szubjektív dolog, azonban vannak olyan használhatósági jellemzők, amelyek jó alapot kínálnak az elemzésre. Ezek:

1. Tanulhatóság: mennyi idő szükséges a rendszer megtanulásához.

2. Műveleti sebesség: a rendszer válaszideje megfelel-e a felhasználók munkagyakorlatának.

3. Robusztusság: milyen a rendszer hiba tűrőképessége.

4. Visszaállíthatóság: hibák esetén milyen lehetőségek vannak a visszaállításra.

5. Adaptálhatóság: mennyire kötött a rendszer modellje.

Az értékelés során fontos, hogy milyen eszközzel végezzük a mérést és milyen formában. A felhasználói felületek értékelésére számos egyszerűbb, kevésbé drága, és a tervezés bizonyos hiányosságainak azonosítására képes technika is létezik:

1. Kérdőívek, amelyek arról gyűjtenek információt, hogy a felhasználók mit gondolnak a felületről.

2. A rendszer felhasználóinak munka közben történő megfigyelése.

3. A jellegzetes rendszerhasználat videó „pillanatfelvételei”.

4. Olyan kódrészlet beépítése a szoftverbe, amely a legtöbbször használt lehetőségekről és a leggyakoribb hibákról gyűjt információt.

A felhasználók kérdőív segítségével történő felmérése a felület értékelésének egy viszonylag olcsó módja. A kérdéseknek inkább pontosnak, mint általánosaknak kell lenniük.

A megfigyelésen alapuló értékelés egyszerűen a felhasználók figyelését jelenti, miközben azok használják a rendszert.

A videó felvétel alapú értékelés azt jelenti, hogy valamilyen képrögzítő felszerelés segítségével felvételeket készítünk a rendszer használatáról abból a célból, hogy a felhasználó tevékenységeket később elemezni tudjuk.

Azt mondhatjuk, hogy az egyik leghatékonyabb elemzési módszert a statisztika készítő kódrészlet beépítése a szoftverbe. Ez a felületek sokféle tökéletesítését teszik lehetővé. Felfedezhetők a leggyakoribb műveletek, és a felületet át lehet úgy szervezni, hogy ezeket a lehető leggyorsabban lehessen kiválasztani.

4. Ellenőrző kérdések

1. Mit értünk a szoftvertervezés folyamata alatt?

2. A tervezés miért kritikus része a szoftver fejlesztési folyamatának?

3. Soroljon fel legalább négy architekturális kérdést.
4. Miket foglalhat magában egy elkészült egy architekturális tervezési dokumentum?
5. Röviden vázolja mit értünk moduláris felbontás alatt.
6. Mi az oka annak, hogy a rétegzett modellre épülő bizonyos rendszerek strukturálása igen bonyolulttá is válhat?
7. Mit értünk vezérlési stíluson?
8. Röviden ismertessen a hívásvisszatérés modell lényegi koncepcióját.
9. Vázolja mit értünk az objektum fogalom az objektumorientált megközelítésben.
10. Mi a különbség az aktív illetve a passzív objektumtípusok között?
11. Milyen technikák álnak rendelkezésre a felhasználók elemzésére?
12. Mi a célja egy elkészítendő prototípus felhasználói felületnek?

7. fejezet - A Unified Modeling Language (UML)

Minden mérnöki szakterület rendelkezik a szakmán belül mindenki által ismert, szabványosított grafikus jelölésrendszerrel (ilyen például a gépészmérnökök számára a géprajz, a villamosmérnökök számára a kapcsolási rajz). Ennek segítségével valósítható meg a különböző munkafázisokban dolgozó szakemberek közötti egyértelmű információcsere.

Az objektum orientált fejlesztési módszerek szerint a fejlesztés során a rendszer különböző nézőpontú modelljeit kell elkészíteni. A modellek dokumentálására megfelelő technikára van szükség. Ennek a technikának szabványosnak kell lennie, mert ez teszi lehetővé a fejlesztők közötti kommunikációt, egyfajta közös nyelvet képezve. A szabványosítás egyben alapvető feltétele annak is, hogy a technikát támogató eszközöket lehessen kifejleszteni.

A szoftver mérnök számára ez a szabványosított jelölésrendszer a Unified Modeling Language (a továbbiakban UML).

1. Az UML története

Bár az 1990-es évek közepére ötvennél is több objektum orientált fejlesztési módszertan is kialakult, a gyakorlat azt mutatta, hogy ezek közül volt három olyan, amelyek:

1. széleskörűen ismertté váltak

2. számos fejlesztési projekt során bizonyították a gyakorlati alkalmazhatóságukat

3. több fejlesztőeszköz és CASE rendszer támogatja az alkalmazásukat.

A sokszor csak részletkérdésekben különböző módszertanok egymás mellett fejlődtek. Ezek a módszerek - bár sokban hasonlítanak egymáshoz - nem azonos területeken a legerősebbek. A három módszer és erősségeik:

1. Booch'93 (Booch): (Object Oriented Design) erős a tervezés fázisában, népszerű az engineering-intenzív alkalmazásoknál.

2. OMT2 (Rumbaugh) : (Object Modeling Technique) erős az analízis fázis során, népszerű az adat-intenzív alkalmazásoknál.

3. OOSE (Jacobson): (Object Oriented Software Engineering) kiváló támogatást ad a "business engineering"-hez, és igazán csak ez támogatja a követelmény analízist.

Megalkotóik eleinte „módszerháborút” vívtak egymással, ragaszkodva a saját elméletükhöz. Bár ez egy ideig az egyes módszertanok fejlesztését segítette, végül az erőforrások szétforgácsolódásához vezetett. Ezt felismerve 1994-ben Grady Booch és Jim Rumbaugh, mint a Rational Software Corporation vezető szakemberei, elhatározták egy egységes fejlesztési módszertan kidolgozását.

Minden módszertan (így természetesen a fent említettek is) alkalmaznak megfelelő jelölésrendszert a fejlesztés során az információk rögzítésére. Az egységesített fejlesztési módszertan kidolgozásának első lépcsőjeként ezért egy egységesített jelölésrendszer kidolgozását tűzték ki célként, felhasználva mindegyik módszertanból a legjobban bevált elemeket.

Ennek jelölésrendszernek a Unified Modeling Language (Egységesített modellező nyelv) nevet adták. A nyelv 0.8 verziószámú tervezetét 1995 októberében publikálták.

A munkához 1995-ben Ivar Jacobson, az OOSE módszertan kidolgozója is csatlakozott (hozzáadva a nyelvhez az általa kifejlesztett use case diagramot) és 1996. októberében már a majdnem véglegesnek szánt, 0.91 verziószámú tervezetet publikálták.

A munkához számos nagy software fejlesztő cég is csatlakozott (olyanok, mint IBM, Digital, Microsoft, Oracle, HP és még mások is), azzal a szándékkal, hogy saját szempontjaiknak is érvényt szerezhessenek, és így általuk is szabványként elfogadható és fejlesztéseikben használható jelölésrendszer szülessen.

1997. január 17.-én publikálták az UML 1.0-ás verzióját, amelyet szabványként való elfogadásra benyújtottak az OMG-hez. (Object Management Group).

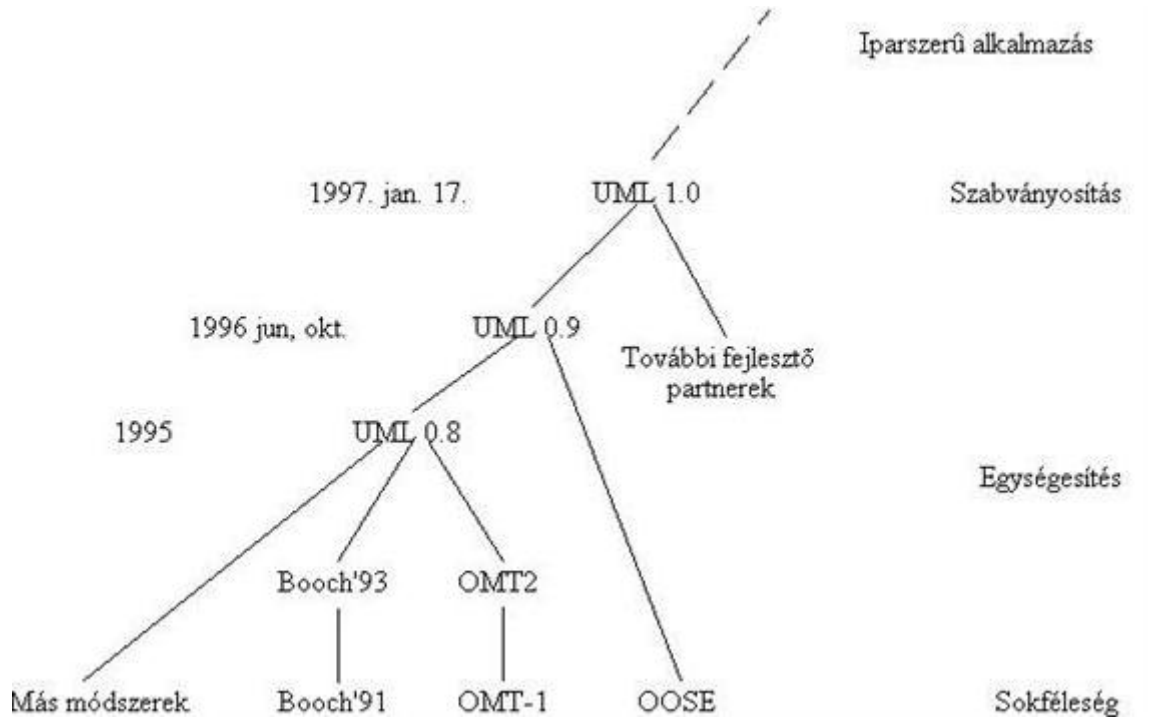
Az első szabványosított verzió az 1997 szeptemberében kiadott UML 1.1. 2005-ig egymás után jöttek ki az 1.x verziók, amelyeket egyre növekvő érdeklődés kísért, és egyre több fejlesztési projektben használták. Ezek a verziók a fejlesztések során szerzett gyakorlati tapasztalatok felhasználásával pontosították, finomították a nyelvet. A 2003-ban megjelent 1.5-ös verzió már a szakma egésze által elfogadott, érett változata lett az UML-nek.

Az informatikai rendszerek fejlesztésével szemben támasztott követelmények változásaihoz igazodva és a folyamatosan fejlődő fejlesztési technológiák igényeihez alkalmazkodva folytatódott a nyelv fejlesztése. A 2005-ben kiadott 2.0-ás verzió újabb eszközökkel gazdagodott, érezhetően kibővítve a nyelvet és megerősítve annak kifejező készségét.

Azóta ennek a bővített változatnak újabb finomításai jelentek meg. Az „utolsó verzió” kifejezés leírása mindig az elavulás veszélyét hordozza, ezért hangsúlyozottan a dátummal együtt: a jegyzet írásának idején (2011 elején) a legfrissebb verzió a 2.3. Az aktuális információk a www.uml.org címről érhetők el, ahol a nyelv hivatalos specifikációján túl (amely kétségkívül autentikus, de nem feltétlenül olvasmányos forrása az UML-ről szerezhető ismereteknek) számos hasznos ismertető és további linkek találhatók.

Az UML kialakulását, előzményeit és időskáláját mutatja az alábbi ábra:

7.1. ábra - Az UML kialakulása



2. Az UML fogalma, tervezési alapelvei

Az UML tehát egy nyelv (a szó informatikai értelmében, mint arra rövidesen még kitérünk). A definíciójához használjunk fel hiteles forrást. Az OMG Unified Modeling Language Specification, vers. 1.5 dokumentum 1.1 pontjának (Overview) első két mondata:

“The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.

The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems."

A fenti definíció szerint tehát az UML rendszerek modelljeinek vizuális dokumentálására alkalmas eszköz. A „rendszer" lehet szoftver, üzleti folyamatok rendszere, vagy bármi más is. Természetesen számunkra a szoftver alapú rendszerek modelljeinek leírása a legfontosabb. Fontos kulcsszó még a fenti meghatározásban az, hogy komplex rendszerek modellezésére is alkalmas eszközként határozza meg az UML-t.

Az OMG definíciója:

"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

A fenti definíciókból is kitűnik, hogy az UML egy eszköz (nyelv) a fejlesztés során előállítandó modell egyes elemeinek dokumentálására, de nem módszertan. Önmagában nem ad tanácsokat vagy előírásokat, hogy elemeit mikor, milyen sorrendben, mire érdemes használni. Ebből a szempontból hasonló a programozási nyelvekhez: ha valaki csupán egy nyelv szintaktikáját és szemantikáját tanulja meg, még nem fog tudni működő, és valós problémát megoldó programot írni – ehhez az adott nyelv eszközeinek lehetséges felhasználási módjait is meg kell tanulnia (algoritmusok, adatszerkezetek, programtervezési és programozás technikai fogások stb.).

Az UML tehát számtalan módon felhasználható a szoftverfejlesztési folyamat támogatására, de hogy ez hogyan történjen, a konkrét módszertanok írhatják elő. Annyit azonban megállapíthatunk, hogy szabványosságából és a szakmán belüli ismertségéből adódóan gyakorlatilag minden ma folyó fejlesztési projekt felhasználja, bármilyen fejlesztési módszertant is alkalmaz.

Az UML tervezése során az alkotók rögzítettek néhány alapvető célkitűzést. A továbbiakban vegyük sorra ezeket az alapelveket.

2.1. Kifejező vizuális modellező nyelv biztosítása

A fejlesztőknek egy olyan jól használható, kifejező modellező nyelvre van szükségük, amelynek segítségével pontosan le tudják írni a modell egyes elemeit és azok összefüggéseit. Ezek a modell leírások a fejlesztő csapat minden tagja számára ugyanazt kell jelentsék. Az UML egységbe foglal számos olyan modellezési eszközt, amelyek különböző módszertanokban megtalálhatók. A tervezés során az volt a cél, hogy az UML eszközei alkalmazhatók legyenek különböző jellegű és komplexitású rendszerek fejlesztése során is. Ennek érdekében számos elemet definiál, amelyekből egy adott fejlesztés során elég csak a projekt sajátosságainak megfelelőket alkalmazni. Így egyszerre biztosít lehetőséget a nagy bonyolultságú, nagy méretű és az egyszerűbb rendszerek fejlesztésére, és képes alkalmazkodni a különböző jellegű rendszerek (például vállalati információs rendszerek, real-time rendszerek, web alkalmazások stb.) igényeihez.

2.2. Lehetőség az alap koncepció bővítésére és specializálására

Az alkalmazásfejlesztéssel szemben támasztott mai követelmények szükségessé teszik, hogy az UML alkalmazkodni tudjon újonnan felmerülő igényekhez, újonnan kifejlesztett technológiákhoz és speciális fejlesztési területekhez. Az UML lehetővé teszi, hogy

1. egy legtöbb, szokásos jellegű fejlesztéshez elegendő legyen az alap eszköztár

2. új elképzelésekkel az alapok módosítása nélkül legyen kiegészíthető (kiterjesztési mechanizmus)

3. egy adott alkalmazásterület speciális igényei szerint testre szabható legyen

2.3. Programozási nyelvtől és módszertantól független legyen

Az UML használható a ma ismert bármelyik programozási nyelvet és módszertant használó fejlesztés során, mert azoktól független absztrakciós szinten fogalmazza meg a rendszer modelljét. Például egy megfelelő részletességű osztálydiagram implementálható Java osztályok halmazaként, vagy akár SQL utasítások sorozataként. Az UML diagramtípusait pedig minden jelenleg ismert módszertan be tudja illeszteni a saját koncepciójába.

2.4. Biztosítson formális alapot a modellező nyelv megértéséhez

Az UML egy nyelv a szó informatikai értelmében (mint például a programozási nyelvek.) Ahhoz, hogy UML-t támogató eszközök készülhessenek, precíz (lehetőleg formális) definíciók szükségesek az eszköz készítők számára. Az UML formális definíciója egy metamodell, amit osztály diagramok segítségével definiáltak. Bár ez matematikai értelemben nem teljesen pontos definíció, ezért néhány eleme szóbeli értelmezésre is szorul, a fejlesztőeszközök írói számára elegendő.

Az UML nyelv alkalmazóinak ez a metamodell túlságosan absztrakt, ezért számukra több magyarázatot tartalmazó specifikáció, illetve számos tutoriál áll rendelkezésre.

2.5. Támogatja az objektum orientált eszközök fejlesztését

Miután a szakemberek által ismert és használt, szabványos eszköz, UML diagramok készítésére számos program került kifejlesztésre. Ezek közül a legegyszerűbbek csupán diagram rajzolók, de vannak a teljes fejlesztési ciklust lefedni szándékozó fejlesztési környezetet biztosító termékek is.

2.6. Az eddigi gyakorlati tapasztalatok ("best practices") integrálása

Az UML tekinthető egy olyan koncepciónak, ami az eddigi fejlesztések tapasztalatait feldolgozva, integrálja az azok során összegyűlt bevált megoldásokat.

3. UML diagram típusok

Az UML 13 különböző diagramtípusból épül fel. A 2.0 és magasabb verziók a diagramokat két fő csoportba sorolják.

Strukturális diagramok (structural modeling diagrams): a modell statikus architektúrájának definiálására alkalmasak. Azokat az elemeket (és egymás közötti kapcsolatait és függőségeiket) modellezhetjük segítségükkel, amelyekből a rendszer felépül: osztályok, objektumok, interfészek, fizikai komponensek.

Viselkedés diagramok (behavioral modeling diagrams): a modell dinamikus aspektusainak modellezésére használatosak. A modell statikus elemeinek együttműködését, az egyes elemek viselkedését írhatjuk le segítségükkel. Mindig van idő dimenziójuk.

Strukturális diagramok:

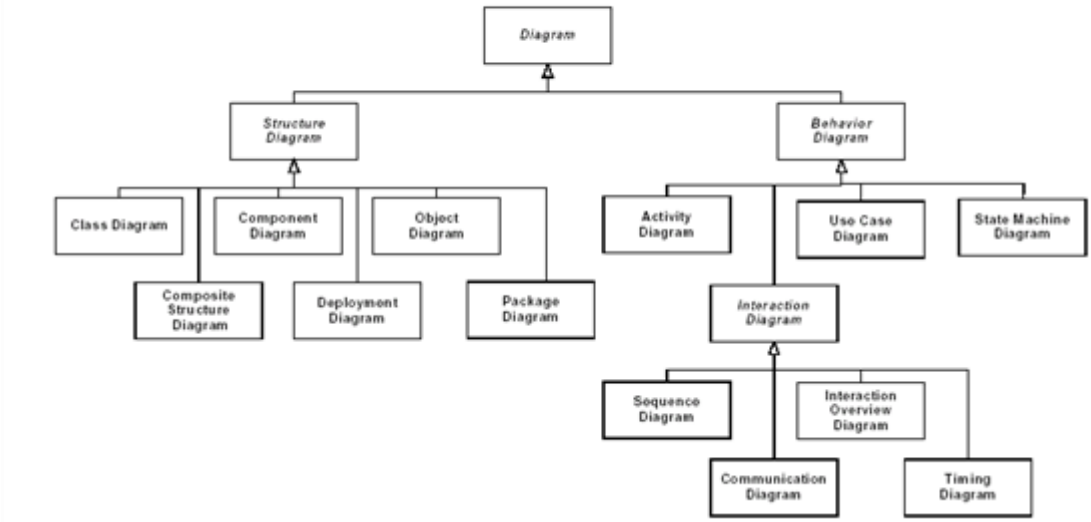
1. osztály diagram (class diagram)
2. objektum diagram (object diagram)
3. csomag diagram (package diagram)
4. összetett struktúra diagram (composit structure diagram) [„montázsdiagram”, „architektúra diagram”]
5. komponens diagram (component diagram)
6. telepítési diagram (deployment diagram) [„kihelyezési diagram”]

Viselkedés diagramok:

1. használati eset diagram (use case diagram)
2. aktivitás diagram (activity diagram)
3. állapotgép diagram (state machine diagram) [„állapot diagram” „állapot-átmenet diagram”, „állapotautomata”]
4. kölcsönhatás diagramok:

A számos diagram típusa áttekintését segíti az alábbi ábra:

7.2. ábra - Az UML digaram típusai



Megjegyzés:

Bár ma már egyre több információ található magyarul is az UML-ről, tapasztalataink szerint a magyar szaknyelv nem egységes számos ide kapcsolódó fogalom magyar megfelelőjében. Ezért az egyértelműség kedvéért a legfontosabb fogalmak angol megfelelőjét az első előfordulásuk során zárójelben mindig megadjuk. (Ez egyben segíti a jegyzet olvasóját abban is, hogy a bőségesen rendelkezésre álló angol nyelvű szakirodalmat könnyebben olvashassa.) Amennyiben ismereteink szerint az adott fogalomra több magyar megfelelő is használatos, ezeket szögletes zárójelben szintén felsoroljuk. Ezt a módszert a jegyzet további részeiben is alkalmazzuk.

A jegyzet terjedelmi korlátai nem teszik lehetővé, célkitűzése pedig nem teszi szükségessé, hogy az UML valamennyi diagram típusát teljes részletességgel ismertessük. Az összes részletet a már említett www.uml.org címről letöltheti az érdeklődő: a nem éppen didaktikai szempontok alapján felépített „OMG Unified Modeling Language (OMG UML), Superstructure” című specifikáció (jelenleg) 758 oldalas. A jegyzet íróinak a fejlesztési munkák során szerzett tapasztalatai, és a szakirodalom ajánlásai alapján határoztuk meg, mit érdemes ezen keretek között bemutatni.

A jegyzetnek nem önmagában az UML ismertetése a feladata, ezért az egyes diagramok ismertetése is több fejezetre oszlik szét, mert mindegyik diagramot a tipikus felhasználási környezetébe igyekeztünk elhelyezni. Így a 8. fejezetben kap majd helyet a viselkedés diagramok közül a használati eset diagram, a statikus nézettel foglalkozó 9. fejezetben találkozunk majd a strukturális diagramokkal, a 10. fejezetben pedig a dinamikus modellnézet leírására alkalmas további viselkedés diagramokkal.

4. Kiterjesztési mechanizmusok

Mivel az UML alapvetően vizuális nyelv, legfontosabb eszközei a diagramok. A modellek leírásához szükséges információk azonban olyan széles körűek és változatosak, hogy nem minden írható le a diagramok eszközkészletével, az UML biztosít egy kiterjesztési mechanizmust a diagramokkal ki nem fejezhető tartalmak leírására. Ez egyben lehetőséget biztosít a nyelv kibővítésére, speciális igényekhez való igazítására is, mint ahogyan azt az egyik tervezési alapelv is megköveteli.

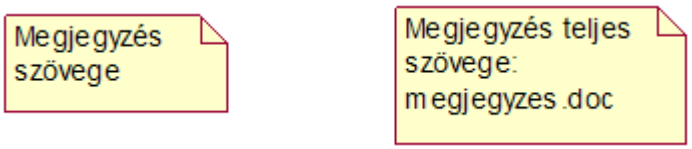
A kiterjesztési mechanizmusok olyan UML elemek, amelyek segítségével más modell elemek jelentését pontosíthatjuk, azokhoz másként nem kifejezhető többlet információkat társíthatunk, illetve új, az UML által nem szabványosított modell elemeket vezethetünk be. Ezért ezeket még a diagram típusok ismertetése előtt célszerű áttekinteni, hiszen bármilyen diagram részeként előfordulhatnak.

Az UML kiterjesztési mechanizmusa több, meglehetősen eltérő bonyolultságú eszközt foglal magában.

4.1. Megjegyzés

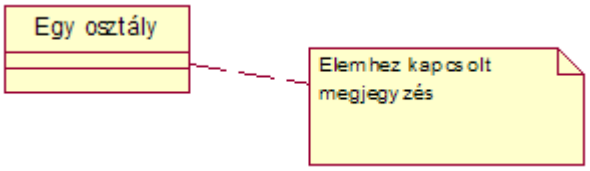
A legegyszerűbb, minden nyelvben ismert lehetőség arra, hogy tetszőleges információval bővítsük a formális eszközökkel leírt tartalmat. Tetszőleges diagramon, tetszőleges helyen és céllal elhelyezhetjük. Mivel az UML egy grafikus nyelv, egy ikon tartozik hozzá.

7.3. ábra - Megjegyzés



Kapcsolható egy tetszőleges más elemhez:

7.4. ábra - Kapcsolt megjegyzés



A megjegyzés ikonban tetszőleges szöveg, hivatkozás, más kiterjesztési mechanizmus eszköz (például kulcsszavas érték) is elhelyezhető.

4.2. Sztereotípia

A sztereotípia (stereotype) formálisan francia idézőjelek között elhelyezett tetszőleges (bár többnyire egy-két szavas) szöveg.

« megnevezés »

Segítségével megjelölhetünk, minősíthetünk tetszőleges modell elemet. A minősített név előtt vagy fölött kell megadni. Ikon is rendelhető hozzá, ezáltal szimbolikus vagy vizuális sztereotípiák is létrehozhatók.

Az UML specifikációjának mellékletében mintegy nyolc oldalnyi előre definiált szöveges sztereotípia található. Ezeket - a félreértések elkerülése végett – csak a definiált jelentésükben szabad használni. A modell készítője által használt saját sztereotípiák jelentését pedig – ha az nem egyértelmű –dokumentálni kell.

A későbbiekben látunk majd példákat előre definiált sztereotípiák használatára.

4.3. Megszorítás (Constraint)

Formája kapcsos zárójelek közé zárt szöveg.

{megszorítás leírása}

A leírás lehet szabad szöveges, de létezik formális leírás is. Az eredetileg az IBM által kifejlesztett OCL (Object Constraints Language) speciális leíró nyelv továbbfejlesztett változata ma már az UML szabvány része.

Megadható a minősített elem után vagy alatt, vagy kapcsolt megjegyzésben.

4.4. Kulcsszavak értékek

Formája kapcsos zárójelbe zárt név-érték párok felsorolása, vesszővel elválasztva. Érték önállóan is szerepelhet, ha egyértelmű a jelentése. Formálisan ez is egy speciális megszorítás, így szintén a minősített elem után vagy alatt, vagy kapcsolt megjegyzésben adható meg. Például:

{author="Kiss Pista", version=0.9.9, date=2011.01.01}

4.5. Profilok

Sztereotípiák és kulcsszavas értékek halmazát profilba foglalhatjuk össze. Számos előre definiált profil létezik az UML-hez, de egy fejlesztési projekt vagy egy fejlesztő szervezet saját igényei szerint újabbak is kialakíthatók. Ezzel az UML szabványos eszközkészlete bővíthető.

Előre definiált profilok léteznek fejlesztési platformok (például .NET, EJB) programozási nyelvek vagy például a SQL specifikus eszközök UML-be integrálására.

5. UML eszközök

Számos program támogatja az UML ábrák vagy az UML modellek előállítását. A fejlettebb eszközök a vizuális modellezést a teljes fejlesztési fázis részeként tekintik, és a követelmény analízistől a különböző nézőpontú modellek UML diagramokkal való felépítésén át a modellekből történő kódgenerálásig a teljes fejlesztési folyamatot igyekeznek átfogni. Számos ilyen rendszer az UML diagramokon kívül egyéb, a fejlesztés során hasznos diagramokat is támogat.

Az UML-t támogató programok között szabad szoftverek és kereskedelmi termékek egyaránt megtalálhatók. A népszerű programfejlesztési környezetek legtöbbje szintén rendelkezik beépített vagy beépíthető UML támogatással.

Hasznos szolgáltatásai ezeknek az eszközöknek az alábbiak:

1. Kódgenerálás (többnyire több nyelven is) az osztálydiagramból (forward engineering).
2. Osztálydefiníciókból osztálydiagram készítése (reverse engineering)
3. Osztálydiagram és osztálydefiníciók (implementáció) összekötése (round-trip engineering). Ebben az esetben a modellen végrehajtott változtatások automatikusan megjelennek az implementációban, és fordítva.
4. Dokumentáció generálás a diagramokból
5. UML diagramok szabványos XML formátumú exportja (XMI, XML Metadata Interchange), amely lehetővé teszi modellek átvitelét egyik modellező eszközből egy másikba.

Néhány UML-t támogató eszköz (messze a teljesség igénye nélkül):

1. Microsoft Visio: csak diagramszerkesztő, osztálydiagram, állapotdiagram, szekvencia diagram és telepítési diagram szerkesztése
2. Visual Paradigm for UML: platform független, kereskedelmi termék. Egyetemi oktatási licenc igényelhető. A jegyzet UML ábrái ezzel készültek.
3. Rational Rose: platform független, kereskedelmi termék. Az egyik első, és legtöbb szolgáltatással rendelkző UML eszköz.
4. Borland Together: platfoprm, független, kereskedelmi temék.
5. StarUML Windows platform, pár éve nem fejlesztik tovább. Ingyenes, néhány diagram típust nem támogat.
6. Poseidon UML: platform független, kereskedelmi termék. Korlátozott funkcionalitású ingyenes verziója van.

8. fejezet - A használati eset modell

A használati eset fogalma és a használati eset diagram már az UML megalkotása előtt megjelent Jacobson munkáiban, az általa kifejlesztett OOSE fejlesztési módszertan részeként. A használati eset diagram célja leírni a modellezendő rendszer és a környezete kapcsolatait.

A használati eset modell a rendszer felhasználói nézetét szemlélteti. A modell elemei:

1. egy vagy több használati eset diagram,

2. a diagramokon megjelenő modell elemek megnevezése, rövid, összefoglaló leírása, amely a fejlesztés korai fázisában kialakul,

3. az egyes modell elemek részletes specifikációja, amely a fejlesztés későbbi fázisaiban bővíti, pontosítja a modellt.

A használati eset modell a feltárt követelmények elemzése alapján készülhet el. Jelölésrendszere elég egyszerű és szemléletes ahhoz, hogy a megrendelő is megértse, ezért alkalmas a megrendelő és a fejlesztő közötti kommunikáció pontosítására.

A használati eset diagramok alapvelő elemei az alábbiak:

1. használati eset (use case): egy felhasználó által látható / igényelhető, a fejlesztendő rendszer által megvalósítandó funkció vagy funkció csoport

2. aktor (actor): a rendszerrel kapcsolatba kerülő személyek, külső rendszerek, akik/amelyek a rendszer szolgáltatásait igénybe veszik

3. Kapcsolatok az aktorok és használati esetek között.

4. Kapcsolatok a használati esetek között

5. Kapcsolatok az aktorok között.

1. Az aktor

A fejlesztendő rendszer tekinthető egy funkció halmaznak. Ezek a funkciók a felhasználók munkáját segítik, számukra valamilyen hasznos szolgáltatást nyújtanak. Ebből a szempontból azonban nem az a lényeges, hogy ki a felhasználó, hanem az, hogy milyen szerepet játszik a rendszer használata során. (Nem az a fontos például a Neptun rendszerben, hogy Kiss Pistának hívják azt, aki egy vizsgára akar jelentkezni, hanem hogy ő hallgató.)

Az aktor tehát a felhasználó egy lehetséges szerepkörét jelenti, amelyet betöltve lép kapcsolatba a rendszerrel, hogy annak információt szolgáltasson, vagy egy szolgáltatását igénybe vegye. Aktor nem csak személy lehet, hanem valamilyen külső rendszer, eszköz is. (Például egy bérszámfejtő rendszernek adatokat kell szolgáltatni az adóhatóság felé a levont adóelőlegről, így ennek a rendszernek egy aktora az adóhatóság rendszere, amely ezt a jelentést fogadja.)

A felhasználó és az aktor fogalmak között több-több kapcsolat van.

1. Több felhasználó - egy aktor: sok tényleges felhasználó léphet kapcsolatba a rendszerrel ugyanolyan szerepkörben (például minden egyetemre beiratkozott személy használhatja a Neptun rendszert hallgatóként)

2. Egy felhasználó - több aktor: ugyanaz a személy több szerepkörben is használhatja a rendszert (például egy doktorandusz lehet tanszéki alkalmazott, aki hallgató szerepkörben használja a Neptunt, amikor felvesz egy neki meghirdetett tárgyat, ugyanakkor oktatói szerepkörben például beírhat egy vizsgajegyet.)

Az aktor szimbóluma egy pálcikaember. A szimbólum alá kell írni a szerepkör megnevezését.

8.1. ábra - Aktor

2. A használati eset

A használati eset a rendszer olyan funkciója vagy funkció halmaza, amely a felhasználó (pontosabban az aktor) céljának eléréséhez járul hozzá.

A használati eset jele egy ellipszis. Az ellipszisbe, vagy az alá kell írni a megnevezését.

8.2. ábra - Használati eset

A felhasználó egy adott céljának eléréséhez lehet, hogy több használati eset végrehajtása szükséges. Például, ha egy web áruházban vásárolni akarunk, ahhoz az alábbi funkciókat kell igénybe venni:

1. bejelentkezés
2. keresés az áruk között
3. áru kosárba helyezése
4. fizetési és szállítási adatok megadása

Ezek a modellben egy-egy használati esetként jelenhetnek meg.

A használati eset modell nem tartalmaz idő dimenziót, csak szükséges funkcionalitásokat. Az előző példában felsorolt használati esetek a modellben egymás mellé helyezve jelennek meg. Az, hogy egy konkrét vásárlás során ezeket milyen sorrendben kell igényelni és végrehajtani, majd a rendszer dinamikus nézetének megalkotása során kell eldönteni a fejlesztőknek.

3. Kapcsolatok

A használati eset diagram elemei között kapcsolatokat jelölhetünk. A diagram jellegéből adódóan nem lehet olyan eleme, amely legalább egy másik elemmel nincs kapcsolatban.

3.1. Kapcsolat aktor és use case között

Aktor és use case között jelzett kapcsolat azt jelenti, hogy az aktor használja a use case-t.

Jele egy vonal a két elem között. Egyes UML szerkesztők nyilat használnak, ami lehetővé teszi, hogy hangsúlyozzuk, hogy az aktor kezdeményező (ekkor a nyíl a használati eset felé mutat) vagy információt fogadó (a nyíl az aktor felé mutat) viszonyban van a használati esettel.

Példa:

8.3. ábra - Aktor és használati eset kapcsolata

Az ilyen kapcsolatot, amikor két modell elem együttműködését jelezzük, nem nevesítve az együttműködés módját, az UML ábrákban asszociációnak nevezzük.

Bár ritkán, de szükség lehet az asszociáció számosságát is jelölni, amit a kapcsolat valamelyik végére írhatunk, és azt jelöli, hogy az adott kapcsolatban az adott elem milyen multiplicitással vesz részt. A lehetséges jelölések:

1. n..m vagy n-m ami az [n,m] intervallumot jelenti

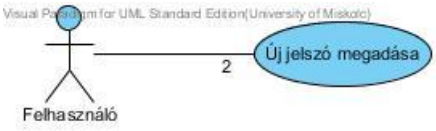
2. n,m, ...,k: a lehetséges értékek felsorolása

3. n,m stb lehet pozitív szám, 0 vagy *, aminek jelentése „bármennyi”

4. a * magában a 0..* -ot jelenti.

Például, ha ki akarjuk hangsúlyozni, hogy jelszó változtatáskor az új jelszót kétszer kell megadni:

8.4. ábra - Használati eset és aktor kapcsolata számossággal



Hasonlóan, például egy „chat” funkció esetén a „Felhasználó” aktor oldalán megjelölhetjük a 2..* számosságot (hiszen egy kommunikációhoz legalább két fél szükséges.)

3.2. Kapcsolat a használati esetek között

A használati esetek között három féle kapcsolat lehetséges:

1. include (tartalmazás)

2. extend (kibővítés)

3. általánosítás / pontosítás (öröklődés)

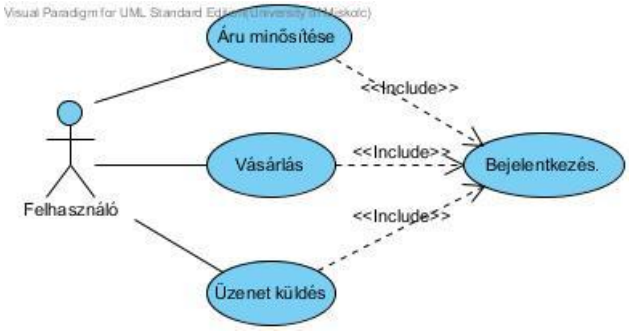
3.2.1. „ include” kapcsolat

A és B használati eset között tartalmazás (include) kapcsolat áll fenn, ha az A használati eset végrehajtásakor a B mindig, feltétel nélkül végrehajtódik. Az A használati esetnek részét képezi a B, esetleg a B ismétlődésével hajtható végre. A tartalmazott használati eset lehet közvetett kapcsolatban egy aktorral, a tartalmazó funkción keresztül, de közvetlen kapcsolatban is állhat aktorral.

Akkor alkalmazzuk ezt a lehetőséget, ha ki akarjuk hangsúlyozni, hogy az A használati eset milyen résztevékenységekből áll össze. Egy használati eset több más használati esetnek is lehet része, ilyenkor a közös funkciók kiemelése egyszerűsíti, redundancia mentessé teszi a modellt.

Például egy web áruházban több olyan tevékenység is lehet, ami bejelentkezéshez kötött. Az alábbi ábra azt szemlélteti, hogy ilyenek az áru minősítése, a vásárlás és az üzenet küldés más felhasználóknak.

8.5. ábra - Használati esetek „include” kapcsolata



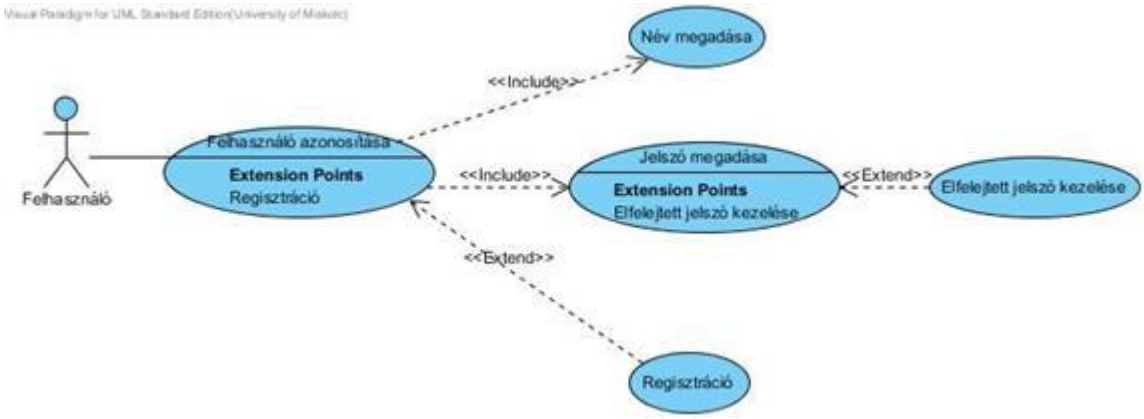
A tartalmazás kapcsolat jele a tartalmazótól a tartalmazott felé mutató, szaggatott vonallal rajzolt nyíl, amelyet az «Include» sztereotípiával minősítünk.

3.2.2. „ extern” kapcsolat

A bővítés kapcsolat az jelenti, hogy egy használati eset bizonyos esetekben (valamilyen feltétel fennállása esetén) egy másik funkció végrehajtását igényli. A bővítő használati eset kiegészíti az alap funkciót, vagy valamilyen kivételes esetet kezel. Az ilyen használati eset önmagában – önállóan – nem fordulhat elő, és aktorral közvetlenül nem lehet kapcsolatban.

Példa: számos web-es szolgáltatás igénybevételének feltétele a felhasználó azonosítása. Az azonosítás része a név és jelszó megadása. (Ezeket a funkciókat az azonosítás funkció tartalmazza.) Ha a felhasználó még nem ismert a rendszer számára, regisztrálnia kell magát. Ha elfelejtette a jelszavát, kérheti azt egy e-mailben. Ezek kiegészítő funkciók. Az alábbi ábra ezt modellezi.

8.6. ábra - Használati esetek „extend” kapcsolata



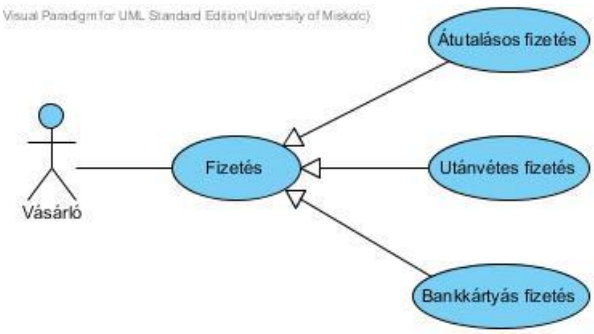
Az extend kapcsolat jele a kiegészítő használati eset felől az alap funkció felé mutató szaggatott vonallal rajzolt nyíl, amelyet az «Extend» sztereotípiával minősítünk.

3.2.3. Általánosítás kapcsolat

Használati esetek között jelölhető az objektum orientált programozásból már ismert általánosítás – pontosítás kapcsolat (öröklődés, generalization). Ez a használati esetek közötti is-a kapcsolatot jelenti. Ha A használati eset leszármazottja B, akkor azt jelenti, hogy B egy speciális esete A-nak (azaz tudja mindazt, amit A, de van néhány speciális, csak rá jellemző tulajdonsága).

Például egy web áruház különböző fizetési módokat ismerhet:

8.7. ábra - Használati esetek általánosítás kapcsolata



3.3. Kapcsolat az aktorok között

Az aktorok között csak az általánosítás – pontosítás kapcsolat értelmezett. Minden más kapcsolat – ha a valóságban létezik is - a rendszeren kívüli, ezért a rendszer szempontjából közömbös. Ha a B aktort az A

leszármazottjának jelöljük, az a szokásos kapcsolatot jelenti: a B bármikor felveheti az A szerepkörét. Például egy rendszer adminisztrátora mindig lehet a rendszer egyszerű felhasználója is, azaz közöttük az alábbi kapcsolat áll fenn:

8.8. ábra - Aktorok általánosítás kapcsolata

4. Használati eset modell készítése

A használati eset modell a rendszer funkcionális követelményeinek leírásának elemzése segítségével készíthető el. Első lépésként célszerű az aktorokat azonosítani, mert az általában könnyebb, és még bonyolult rendszerek esetén sem kell túl sok aktorral számolnunk. Az aktorok köre már a fejlesztés elején is pontosan meghatározható.

Következő lépés a használati esetek azonosítása. Egy bonyolult rendszer esetén nagyon sok használati esettel kell foglalkozni, ezért ebben is az inkrementális megközelítés javasolt.

4.1. Aktorok azonosítása

Az aktorokra a feladat leírásában, a funkciólistákban többnyire valamilyen főnévvel hivatkozunk. Fontos azonban, hogy az aktor

1. a rendszeren kívül létező entitás

2. a rendszerrel valamilyen funkcionális kapcsolatba kerül.

Egy tanulmányi nyilvántartó rendszer leírásában (mint amilyen például a Neptun) találhatunk például egy alábbi mondatot:

„A portásfülke előtt elhelyezett terminálon bejelentkezett hallgató megnézheti, hogy egy általa felvett tárgynak mikor vannak a vizsgái.”

Nyilvánvaló, hogy a portásfülke, és a terminál, bár a rendszeren kívül álló dolgok, nem aktorai a rendszernek, mert nem lépnek vele kapcsolatba, és nem igénylik annak szolgáltatásait, a „hallgató” azonban egy lehetséges aktor.

4.2. Használati esetek azonosítása

Az aktorok azonosítása után a használati esetek azonosítása következik. Használati esetekre valamilyen igei szerkezet utalhat a feladat leírásában, azonban nem olyan közvetlenül, mint az aktorok esetén. A fenti példa mondatban három funkció is el van rejtve:

1. a hallgató lekéri az általa felvett tárgyak listáját,

2. a hallgató kijelöl egy tárgyat ezek közül,

3. a hallgató lekéri a kijelölt tárgy vizsgaidőpontjait.

Az UML nem mondja meg, hogy egy használati eset milyen összetett funkciót modellezhet. A jelölés rendszer ismertetése során a példák egyszerű funkcionalitásokat tartalmaztak, de ha egy bonyolult rendszert ilyen egyszerű funkciókból szeretnénk felépíteni, akár több ezer használati esetet tartalmazó modellt kaphatnánk.

A tapasztalatok szerint egy áttekinthető használati eset diagram legfeljebb tíz-húsz használati esetet tartalmazhat. A használati eset modellt tehát számos használati eset diagramból kell felépíteni. Ezt úgy tudjuk elérni, hogy

1. a rendszert részrendszerekre bontjuk, és azokat külön-külön modellezzük,

2. előbb nagyobb funkcionális egységeket modellezünk egy használati esetként, majd ezeket részletezve jutunk el a finomabb felbontásig.

4.3. Egy összetettebb példa

Ebben a fejezetben egy olyan példát mutatunk, amelyben majdnem minden jelölési elem szerepet kap, és illusztrálja, hogy milyen elemzése lépéseken keresztül juthatunk el egy használati eset modell megalkotásáig.

A modellezendő rendszer (pénztári rendszer) leírása:

„A rendszer egy pár alkalmazottat foglalkoztató, néhány pénztárral működő kis önkiszolgáló élelmiszerbolt pénztári rendszere. A vevő egy kosárban összegyűjti a megvásárolni kívánt árukat. A pénztáros az árukat a vonalkódok beolvasásával azonosítja, és így készíti el a blokkot, ami alapján a vevő fizet. A megvásárolt áruk mennyiségével a rendszer csökkenti a raktárkészletet. Bizonyos áruk csak betétdíjas göngyöleggel együtt adhatók el, és a göngyöleg készletét is nyilván kell tartani. (Például a sör…).

A boltvezető bármikor helyettesíthet egy pénztárost, ha annak valami miatt félbe kell szakítania a munkáját. A boltvezetőnek joga van egy törzsvásárló esetén hitelbe is odaadni az árut, ezt a pénztáros nem teheti meg.

Ha egy áru vonalkódja sérült, ezért nem olvasható be, a pénztáros az árut a megnevezése alapján keresi vissza az adatbázisból.”

4.3.1. Aktorok azonosítása

A fenti leírásból szereplőként kiemelhetők az alábbi szerepkörök:

1. vevő

2. pénztáros

3. boltvezető

Ezek közül a vevő – bár szereplője a folyamatnak – nem aktor, mert a rendszerrel nincs közvetlen kapcsolata. Kapcsolata van a pénztárossal, de ez rendszeren kívüli kapcsolat, tehát a modellben nem kell szerepelnie. Így marad aktorként a pénztáros és a boltvezető.

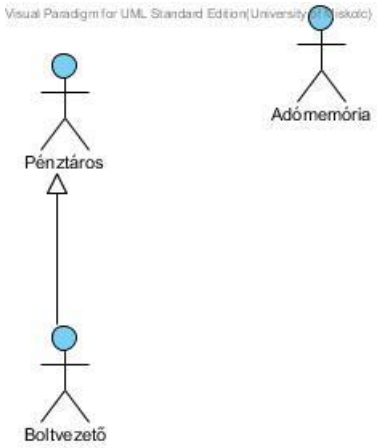
A két aktor között egy általánosítási kapcsolat van: a boltvezető mindig lehet pénztáros szerepkörben, tehát a pénztáros leszármazottjaként modellezhető. (Vegyük észre, hogy bár ez egy hierarchikus kapcsolat, de a beosztási hierarchiával most éppen ellentétes irányú.)

Ez az egyszerű példa is rámutat a szakterületi követelmények fontosságára, és egyben problémát okozó szerepére. Aki ilyen rendszert akar fejleszteni, annak ugyanis tisztában kell lennie azzal, hogy a pénztárgép nem egyszerű számológép: a vonatkozó rendeletek szerint ugyanis kell rendelkeznie egy speciális memória modullal, az úgynevezett adómemóriával, amibe minden blokk adatát be kell írnia, és ennek a memóriának a tartalma nem változtatható meg, csak kiolvasható.

A szakterületi követelmények értelmében tehát van egy harmadik aktor is, az adómemória. Ez nem felhasználó, hanem egy eszköz.

A használati modell aktorai tehát:

8.9. ábra - A pénztári rendszer aktorai



4.3.2. Használati esetek azonosítása

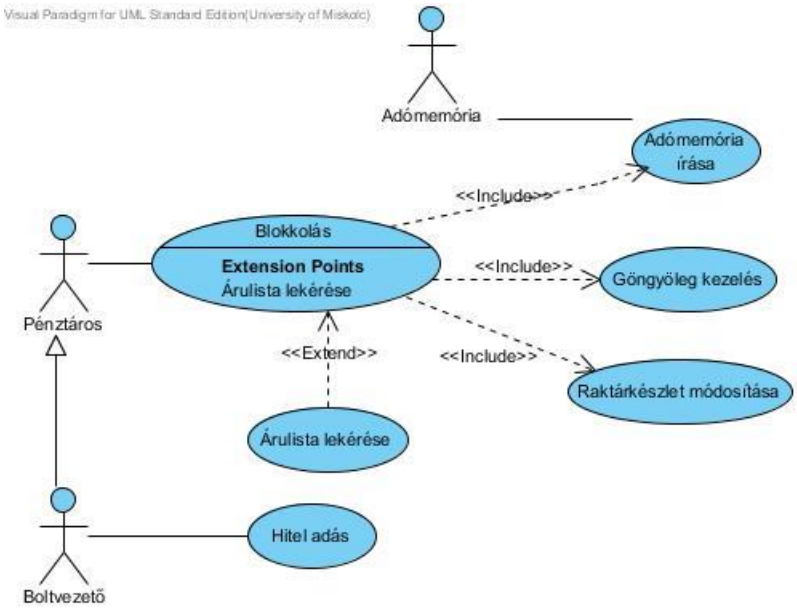
A leírásból kiolvasható, hogy a központi használati eset a blokkolás. Ehhez a pénztáros aktor kapcsolódik. A blokkolásnak szerves része a göngyöleg kezelés, raktárkészlet módosítás és az adómemóriába írás. Ezek tehát „include” kapcsolatban vannak a blokkolással, mert feltétel nélkül, minden esetben végrehajtandók a blokkolás során. Az adómemóriába íráshoz kapcsolódik az adómemória aktor.

A blokkolás különleges esete az, ha nem olvasható a vonalkód. Ilyenkor az áru név szerinti visszakeresése funkció (egyben használati eset) kiegészíti az alap funkcionalitást, tehát a blokkolással „extend” kapcsolatban van. (Csak bizonyos feltételes teljesülése esetén – ha nem olvasható a vonalkód – szükséges.)

A boltvezetőnek van egy olyan funkciója, ami csak rá jellemző: adhat hitelt, ez az utolsó felfedezett használati eset a leírás elemzéséből.

A fenti elemzés alapján az alábbi használati eset diagram építhető fel:

8.10. ábra - A pénztári rendszer használati eset diagramja



4.4. A használati eset modell dokumentációja

A diagram rajzolása során mind az aktoroknak, mind a használati eseteknek nevet kell adni. Az elnevezések nem csak azonosító szerepet töltenek be, (hiszen erre a rajzolásra használt programok által alapértelmezésben adott „Use case 1”, „Use case 2” stb. stílusú azonosítók is alkalmasak lennének), hanem segít a diagram tartalmának megértésében.

Minden további modell típusra is általánosan érvényes az, hogy az elnevezési kényszer egyben annak ellenőrzése, hogy az adott funkciót, fogalmat, kapcsolatot – általánosságban minden modell elem jelentését pontosan megértettük-e. A tapasztalatok alapján elmondható, hogy ha egy modell elemre nem tudunk rövid, frappáns, kifejező elnevezést találni, annak mindig az az oka, hogy nem tisztáztuk pontosan az adott elem jelentését és a modellen belül játszott szerepét.

Minden vizuális modellező eszköznek van olyan funkciója, aminek segítségével az egyes modell elemekhez rövid, tömör, néhány mondatos leírást rendelhetünk. A használati eset modell dokumentációjának ez a második eleme. A tömör leírás segít abban, hogy a modellt könnyebben megértse a fejlesztésben résztvevők mindegyike.

A fejlesztés korai időszakában a leírásokkal kiegészített diagramok elegendőek a használati eset modell definiálására. A későbbi fázisokban ezt ki kell egészíteni a használati esetek által modellezett funkció végrehajtásának részletes leírásával, hiszen ezek alapján lehet megtervezni és implementálni a megvalósításért felelős szoftver elemeket.

Egy használati eset konkrét végrehajtási folyamatának leírását forgatókönyvnek nevezzük. Egy használati esethez számos forgatókönyv kapcsolódhat.

Minden használati esethez általában tartoznak előfeltételek, úgynevezett prekondíciók, amelyeknek teljesülése szükséges ahhoz, hogy bármelyik forgatókönyv alapján a funkció végrehajtható legyen. A végrehajtás után előálló állapotot nevezzük postkondíciónak (utóhatásnak).

A forgatókönyveket aktor interakció – rendszer válasz párok sorozataként specifikálhatjuk. Bonyolultabb használati esetek forgatókönyvét ezen felül aktivitás vagy szekvencia diagrammal is szemléltethetjük.

Gyakran hasznos az egyes használati esetek fontosságát. prioritását („rank”) is megadni, mert ez segíthet a fejlesztési munka ütemezésében. Általában az alacsony – közepes – magas prioritás skála elegendő.

A jegyzet UML ábráit a Visual Paradigm vizuális modellező szoftver oktatási verziójával készítettük. Ez a szoftver is lehetőséget ad arra, hogy az egyes modell elemekhez további információkat csatoljunk, és ezekből az információkból html, pdf vagy Microsoft Word formátumú dokumentációt generáljunk.

Az összefoglaló dokumentáció egy részlete a 8-15. ábra diagramjához:

Summary

Name	Documentation
Pénztáros	A pénztárgépet kezelő személy. Felelős az eladott áruk regisztrálásáért és a pénz kezeléséért.
Blokkolás	Az eladott áruk regisztrálása, a blokk elkészítése és a fizetendő összeg kijelzése.
Boltvezető	Az üzlet vezetője. Bármikor helyettesítheti a pénztárost.

A „Blokkolás” használati eset részletes dokumentációja:

Use Case Descriptions

Full		
Use Case ID	UC8.10	
Super Use Case		
Primary Actor	Pénztáros	

<table>
<tr><td>Secondary Actor(s)</td><td></td></tr>
<tr><td>Brief Description</td><td>Blokkolás</td></tr>
<tr><td>Preconditions</td><td>A pénztárgép megnyitva
A pénztáros bejelentkezve.
Az előző blokk lezárva</td></tr>
<tr><td>Flow of Events</td><td>
<table>
<tr><td></td><td>Actor Input</td><td>System Response</td></tr>
<tr><td>1</td><td>Új vásárló</td><td></td></tr>
<tr><td>2</td><td></td><td>Új blokk niytása
Fizetendő összeg 0.</td></tr>
<tr><td>3</td><td>Vonalkód beolvasás</td><td></td></tr>
<tr><td>4</td><td>Darabszám megadás</td><td></td></tr>
<tr><td>5</td><td></td><td>Újabb blokk tétel kiírása
Fizetendő összeg növelése</td></tr>
<tr><td>6</td><td>Blokk lezárása</td><td></td></tr>
<tr><td>7</td><td></td><td>Végső fizetendő összeg kiírása</td></tr>
<tr><td>8</td><td>A vevő által átadott összeg beírása</td><td></td></tr>
<tr><td>9</td><td></td><td>Visszajáró pénz kijelzése
Kasszafiók kinyitása</td></tr>
</table>
</td></tr>
<tr><td>Post-conditions</td><td>Elkészült és kinyomtatásra került a blokk.</td></tr>
<tr><td>Alternative flows and exceptions</td><td>Hibás darabszám lett megadva - stornózni kell!
Áru vonalkódjának ismételt beolvasása - stornózni kell!</td></tr>
<tr><td>Non-behavior requirements</td><td></td></tr>
</table>

Assumptions		
Issue		
Source		
Author	Ficsor Lajos	
Date	2011.02.21. 22:49:26	

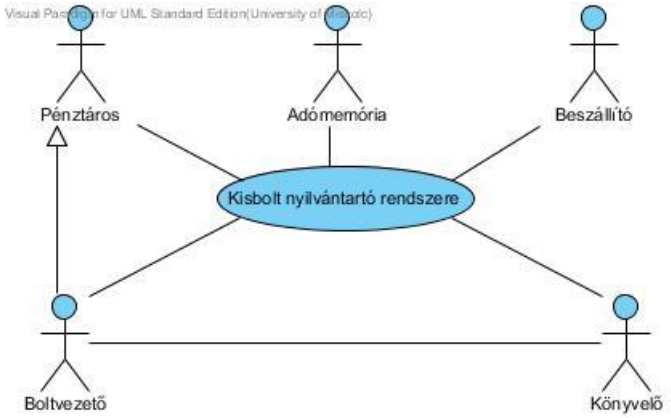
5. A használati eset modell helye a fejlesztési folyamatban

A használati eset modell elkészítése a fejlesztés korai szakaszában, a funkcionális követelmények feltárása során elkezdődhet. Az analízis fázis során az aktorok és a használati esetek rövid leírása egészíti ki a diagramokat.

Gyakran hasznos lehet első lépésként az úgynevezett kontextus diagram (context diagramm) elkészítése, amelyben a teljes rendszer egy használati esetként modellezzük. Ez a megközelítés segíthet feltérképezni a modellezendő rendszer és a külvilág kapcsolatait, az aktorok körét. Az alapszabálytól való eltérésként ilyen diagramok esetén megengedett az aktorok közötti asszociációk jelölése is, ha az segíti az aktorok közötti viszonyok megértését.

A 8-15. ábra diagramja egy kisbolti rendszer funkcionalitásainak csak egy részét (a pénztári modult) fedi le. A teljes rendszer kontextus diagramja további aktorokat fedhet fel, például az alábbiak szerint:

8.11. ábra - A kisbolti rendszer kontextus diagramja

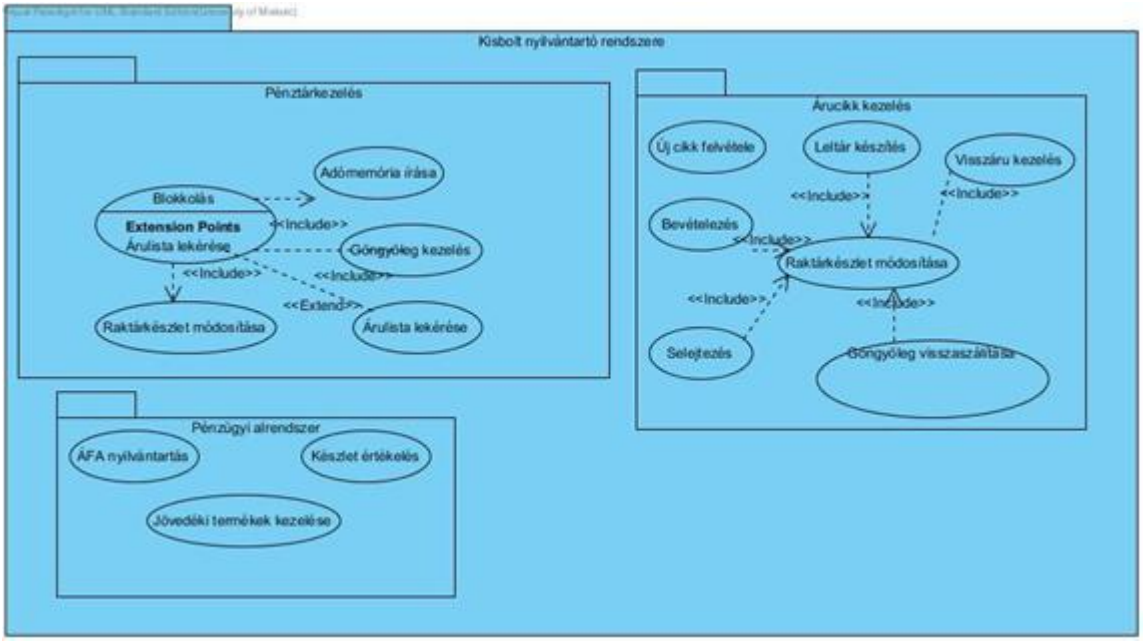


A fenti ábrán látható asszociáció a boltvezető és könyvelő között nyilvánvalóan rendszeren kívüli kapcsolat. Felhívja azonban a figyelmet arra, hogy a két aktornak együtt kell működnie bizonyos üzleti folyamatok végrehajtása során. Az együttműködést a rendszernek biztosítania kell, például a könyvelő kérésére a boltvezetőnek bizonyos adatokat kell szolgáltatnia, és ezeket az adatokat a nyilvántartó rendszernek elő kell tudnia állítani.

Ha a legfontosabbnak ítélt használati eseteket összegyűjtöttük, érdemes az összetartozó funkcionalitásokat alrendszerekbe rendezni. Az UML alrendszer vagy csomag diagramja nyújt erre lehetőséget. Ezt a modell megjelenést használati eset leltárnak nevezzük.

A kisbolti rendszer használati eset leltárának egy induló változata lehet például az alábbi:

8.12. ábra - A kisbolti rendszer funkció leltár diagramja



A fenti funkcióleltár elemzése során az alrendszerek közötti kapcsolatokat tárhatjuk fel, vagy akár új alrendszer(ek) szükségességére is rájöhetünk. A fenti ábrán például észrevehetjük, hogy a „Raktárkészlet módosítása” használati eset két alrendszerben is előfordul. Ez arra utal, hogy ebben az esetben célszerű meggondolni, hogy egy új alrendszert („Készlet kezelés”) alakítsunk ki az ismétlődő funkcionalitások összefogására.

Az alrendszerek kialakítása az analízis fázisból a tervezés fázisba való átmenetet jelenti.

Szintén a tervezési fázis felé haladunk akkor, amikor a használati esetek részleteit dolgozzunk ki. Ez az előző pontban vázolt részletesebb dokumentációk kidolgozását, a forgatókönyvek kialakítását jelenti. Ha szükséges, ezeket a forgatókönyveket viselkedés diagramok segítségével (legtöbbször aktivitás és/vagy szekvencia diagramok alkalmazásával) pontosíthatjuk.

Az analízis fázis során kialakított használati eset modell a rendszer felhasználói nézetét reprezentálja. A forgatókönyvekkel kiegészített modell már a dinamikus nézet részét képezi. A használati eset modell tehát a fejlesztés során folyamatosan bővül, egyre pontosabbá válik.

A használati eset modell megalkotása és részletezése tehát elsősorban az analízis és a tervezési fázis eszköze.

Ha a használati esetekhez prioritást is rendelünk, az a fejlesztési munka ütemezéséhez adhat hasznos információkat.

6. Ellenőrző kérdések

1. Mi a különbség a használati eset modell és a használati eset diagram között?
2. Melyik fejlesztési fázis(ok)hoz kapcsolódik a használati eset modell kidolgozása?
3. Melyek a használati eset diagram elemei?
4. Definiálja az aktor fogalmát!
5. Definiálja a használati eset fogalmát!
6. Mi a különbség a felhasználó és az aktor fogalma között?
7. Mit jelent egy aktor és egy használati eset közötti kapcsolat (asszociáció)?
8. Mit jelent két használati eset között az „include” kapcsolat, és hogyan jelöljük?

9. Mit jelent két használati eset között az „extern” kapcsolat, és hogyan jelöljük?
10. Milyen kapcsolat lehetséges két aktor között?
11. Mi a jelentősége a modell elemek elnevezésének és rövid leírásának?
12. Sorolja fel a használati eset részletes dokumentációjának legfontosabb elemeit!
13. Mi a kontextus diagram?
14. Mi a folyamat leltár?

9. fejezet - Strukturális diagramok

A követelmények összegyűjtése után a modellezés második fontos lépése a rendszer statikus nézetének megalkotása. Ennek során felderítjük, hogy a rendszer milyen részegységekből, architekturális elemekből áll. Az UML strukturális diagramjai a statikus nézet dokumentálására adnak eszközöket. Ez a fejezet a legfontosabb, leggyakrabban használt diagram típusokat tartalmazza.

1. Az osztálydiagram

Az osztálydiagram az UML egyik legtöbbet használt diagram típusa.

Ebben az alpontban összefoglaljuk az osztálydiagramok elemeire vonatkozó szabályokat, de nem foglalkozunk az egyes elemek felhasználási módjaival. Az ismertetés nem teljes. Az egyszerűsítés kedvéért és a terjedelmi korlátok miatt néhány – a tapasztalatok szerint ritkábban használt – elemet nem tárgyalunk. Az osztálydiagramnak a modellezés egyes fázisaiban betöltött szerepét a következő fejezetekben tárgyaljuk.

Az osztálydiagram osztályokat és azok kapcsolatait ábrázolja. Az „osztály” fogalma az objektum orientált programozásból már ismerős, ezért fontos hangsúlyozni, hogy az UML osztály fogalma nem teljesen azonos a programozási nyelvekből megismerttel, attól általánosabb értelemben használjuk.

Az UML terminológiájában az osztály a rendszer valamely statikus, strukturális összetevője. Az analízis modell felállítása során az osztály a problématér (alkalmazási szakterület) fogalmait reprezentálja. Ugyanakkor – a programozási nyelvekhez szemléletéhez hasonlóan - tekinthető absztrakt típusként is, amelynek vannak adatokkal leírható tulajdonságai (attribútumai) és viselkedési mintái (operációi). Ez az értelmezése a tervezési és az implementációs modell elkészítése során válik hangsúlyosabbá.

Az UML osztály által modellezett rendszer elemek kiterjedése is eltérő a programozási nyelvek osztályaitól: a modell részletezettségének szintjétől függően akár egy egész alrendszer is lehet, mert határait nem (feltétlenül) implementációs szempontok határozzák meg.

AZ UML osztályait különböző absztrakciós szinteken is megfogalmazhatjuk. Az egyes fejlesztési fázisok különböző megközelítést tesznek szükségessé, ennek megfelelően különböztethetjük meg az alábbi absztrakciós szinteket.

1. Analízis, elemzés. Ezen a szinten az osztályok az alkalmazási szakterület fogalmait reprezentálják. Az elsődleges feladat, hogy ezeket a fogalmakat és kapcsolataikat értsük meg, ezért számos részlet még hiányozhat. Egy elemzési osztálydiagram elemei közvetlenül még nem lennének implementálhatók.

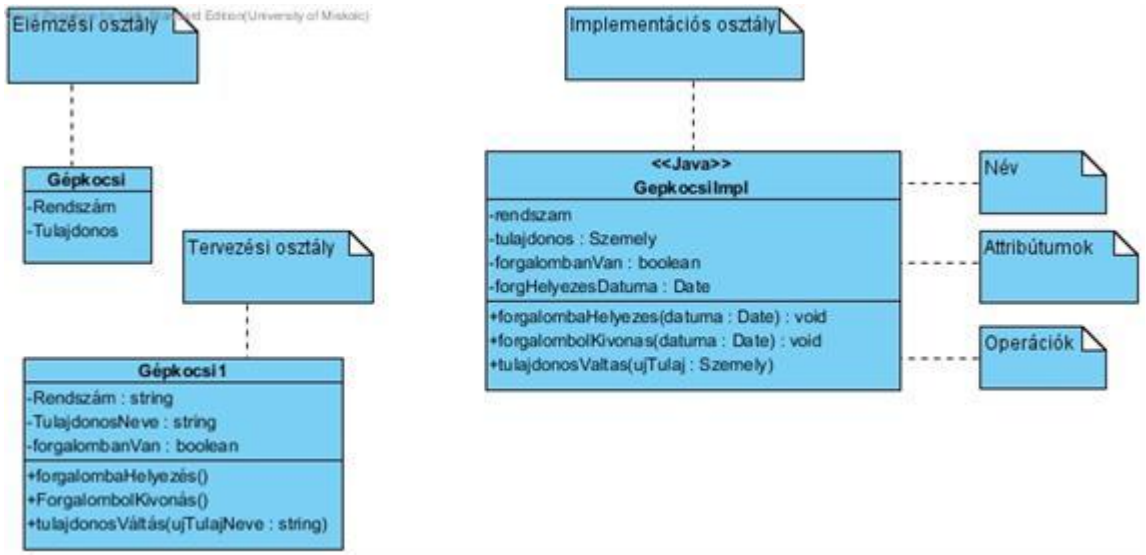
2. Tervezés. Mivel a fázis célja az analízis során megismert struktúra technikai megvalósíthatóságának a biztosítása, az elemzési osztályok a tervezés során feltárt további információkkal és implementációs részletekkel bővülnek. Ezek mellett a modellben megjelennek olyan osztályok is, amelyek nem a szakterület fogalmait modellezik, hanem technikai elemei a megoldásnak (a problématér osztályai mellett megjelennek a megoldási tér osztályai is).

3. Megvalósítás, implementáció. Ezen szinten az osztályok minden olyan részletet tartalmaznak, amelyeknek segítségével egy adott programozási eszköz nyelvi elemei leképezhetők. Ehhez már ismerni kell a célnyelvet, mert nem minden szabályos UML osztály valósítható meg bármely implementációs környezetben (például másként kell megtervezni egy olyan osztályt, amelyet Java osztályra képezünk le, mint amelyet egy adatbázis táblára).

1.1. Az osztály szimbóluma

Az osztály jele alapesetben egy függőlegesen három részre osztott téglalap. A legfelső részbe írandó az osztály neve, a középsőbe az attribútumok, az alsóba az operációk specifikációja. A fejlesztés korai fázisaiban még nincs feltétlenül minden rész kitöltve, és a kitöltés részletessége is változhat. Az egyetlen kötelező kellék a megnevezés. Ha csak a nevével hivatkozunk egy osztályra, a téglalap felosztása is elmaradhat.

9.1. ábra - Osztály szimbóluma



Az osztály jelölése további rekeszekkel is bővíthető, amelyben speciális attribútum vagy operáció fajták, esetleg más diagramok, vagy azok elemei tüntethetők fel, ezzel is kifejezve, hogy azok az osztályhoz tartoznak. Alkalmazhatjuk ez a lehetőséget arra, hogy például az operációkat meghatározó használati eseteket, vagy egyes operációk végrehajtását leíró aktivitás vagy szekvencia diagramot helyezzünk el ezekben a rekeszekben.

Ha túl sok, esetleg a tartalma miatt túl nagy rekeszekkel egészítjük ki az osztályt, az a diagramot nehezen áttekinthetővé teszi, ezért használatában célszerű mértéket tartani. Minden vizuális modellező eszköz lehetővé teszi az egyes modell elemek közötti kapcsolatok létrehozását, az összefüggések feltüntetésére használjuk inkább ezt a lehetőséget.

1.1.1. Attribútumok

Az osztály adat jellegű tulajdonságainak felsorolása. Az attribútum jelentése az egyes absztrakciós szinteken az alábbiak szerint írható le.

1. Analízis: az osztálynak van ilyen elnevezésű adata.
2. Tervezési szint: az osztálynak van adott típusú adata, amelyen meghatározott operációk hajthatók végre (például beállítható, lekérdezhető).
3. Implementációs szint: az osztály adott típusú, elérési módú adata. Pontos megjelenése attól függ, hogy az osztályt hogyan implementáljuk. Például:

Az attribútum jelölése:

láthatóság név : típus = alapérték

Ahol a láthatóságot az alábbi karakterek valamelyike jelzi:

+ public

protected

~ csomagszintű

- private

Az attribútum valamennyi tulajdonságát általában csak az implementációs szintű osztálydiagram tartalmazza.

1.1.2. Operációk

Az osztály példányain végezhető műveletek felsorolása. Az operáció jelentése az egyes absztrakciós szinteken az alábbiak szerint írható le.

1. Analízis: az osztály objektumainak egy lényeges viselkedési eleme.

2. Tervezési szint: az osztály esetén annak publikus módszerei, adatbázis tábla esetén a legfontosabb lekérdezések

3. Implementációs szint: az osztály adott szignatúrájú és elérési módú metódusa. Pontos megjelenése attól függ, hogy az osztályt hogyan implementáljuk.

Az operáció jelölése:

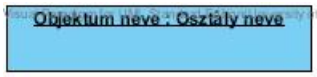
láthatóság név(param) : típus{comment}

1.2. Az objektum szimbóluma

Bizonyos esetekben szükséges lehet adott osztály objektumának vagy objektumainak és a közöttük levő kapcsolatoknak az ábrázolására is. Az objektum jele maximum két részre osztott téglalap (hiszen operációkat nem kell tudni feltüntetni).

Egy konkrét objektum jelölése:

9.2. ábra - Objektum jelölése



Egy osztály tetszőleges objektuma:

9.3. ábra - Egy osztály tetszőleges objektuma

Megadhatók konkrét attribútum értékek is:

9.4. ábra - Objektum adott attribútum értékekkel

Figyelem: az aláhúzás az objektum és osztály névnél, és a kettőspont az osztály neve előtt a jelölés része!

Ez az objektum jelölés az UML nyelven belül egységes, tehát minden olyan diagram esetén, amelyeknek van objektum eleme, szintén ezt kell alkalmazni.

1.3. Osztályok közötti kapcsolatok

Az osztály diagramban jelölhetjük az osztályok közötti kapcsolatokat. A kapcsolatokhoz számos jellemző kapcsolható.

Az általános kapcsolat jele az osztályok között rajzolt egyszerű vonal, ezt asszociációnak nevezzük. Jelentése: „valamilyen kapcsolat van a két osztály között”.

Az UML néhány speciális kapcsolat fajtát is nevesít, amelyeknek specifikált jelentése van, és egyedi jelölés tartozik hozzá.

1.3.1. Asszociáció

Az asszociáció jele az egyszerű vonal. Az asszociációhoz számos tulajdonság kapcsolható. Minden asszociációnak (mint modell elemnek) van elnevezése. Szokás szerint ezt a vonal közepénél a vonal alá írjuk. Az asszociációt jelző vonalak végére, a vonal fölé írható az, hogy az adott osztály a kapcsolatban milyen szerepkörben vesz részt. A szerepkör név elhagyható, ha a kapcsolat nevéből egyértelműen következik. Az asszociáció iránya is jelölhető a vonal végére helyezett nyílvéggel („navigálhatóság”). Ha nincs a vonalon nyílhegy, az kétirányú kapcsolatot jelent. A nyíl (vagy annak hiánya) azt jelenti, hogy a másik oldalon álló osztály ismeri, használhatja az osztályt.

A kapcsolatok további lehetséges jellemzői:

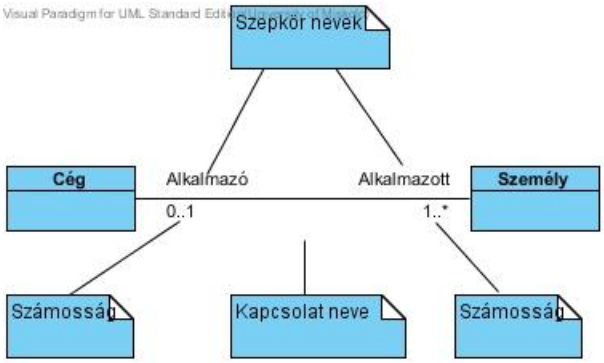
1. A szerepkörök számossága. A számosságot ugyanúgy jelöljük, mint a használati diagram esetén.

2. Az asszociáció minősítője is jelölhető.

3. Az asszociációhoz a tulajdonságait leíró osztály is rendelhető.

Egy egyszerű példa:

9.5. ábra - Asszociáció és alap tulajdonságai

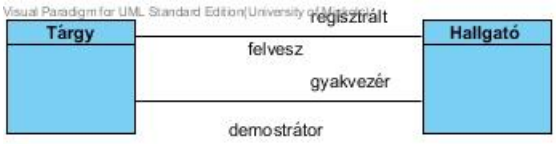


A fenti ábra azt fejezi ki, hogy a Cég és a Személy osztály között „Alkalmazás” kapcsolat van. A kapcsolatban a Cég szerepköre az alkalmazó, a Személyé az alkalmazott. (Ebben az esetben a szerepkör nevek értelemszerűek, tehát elhagyhatók lennének, csak a jelölésrendszer szemléltetése miatt tüntettük fel.)

Az alkalmazó szerepkör számossága 0..1, ami azt jelenti, hogy egy Személynek 0 vagy 1 alkalmazója lehet. A másik oldali számosság szerint egy Cégnek legalább egy, de egyébként tetszőleges számú alkalmazottja lehet. A kapcsolat kétirányú, hiszen a Cégnek és az alkalmazott Személynek kölcsönösen ismerniük kell egymást.

Két osztály közötti asszociációhoz tartozhat több szerepkör is. Ilyenkor minden szerephez egy vonal tarozik, így az egyes szerepkörök és azok számossága elkülöníthető módon jelölhető. Példa:

9.6. ábra - Több szerepkör jelölése



A fenti ábra azt rögzíti, hogy hallgató és tárgy között kétféle kapcsolat lehet. Felvehet egy tárgyat, amit hallgat, de dolgozhat egy tanszéken demonstrátorként is, így egy másik tárgynak lehet gyakorlatvezetője.

Ez az osztálydiagram egyben példa arra is, hogy egy diagram nem feltétlenül tud kifejezni minden szükséges információt. A fenti diagram például nem tartalmazza azt a nyilvánvaló korlátozást, hogy egy hallgató nem lehet egyszerre egy tárgynak a regisztrált hallgatója, ugyanakkor demonstrátorként gyakorlatvezetője. Ezt az a korlátozást csak UML-en kívüli eszközökkel tudjuk kifejezni.

Előírható, hogy egy adott szerepkörben az objektumoknak kötött sorrendben kell részt venniük. A sorrendiség az {ordered} megszorítással írható elő. Példa:

9.7. ábra - Sorrendiségi szerepkör jelölése

Ha ki akarjuk hangsúlyozni, hogy a kapcsolat valamelyik irányban nem navigálható (azaz azt az osztályt a másik nem láthatja), a vonal nem látható osztály felőli végére egy x jelet teszünk.

9.8. ábra - Nem navigálgató asszociáció

A fenti ábra azt fejezi ki, hogy a jelszóból mindig képezhető a tárolt jelszó, de a tárolt alapból nem állítható vissza az eredeti jelszó.

A többes szerepkör esetén megadható egy minősítő, ami a számosság csökkentésére alkalmas adat. Ha az adat egyedivé teszi a kiválasztást, a számosság egyre csökkenhet.

Jelölése:

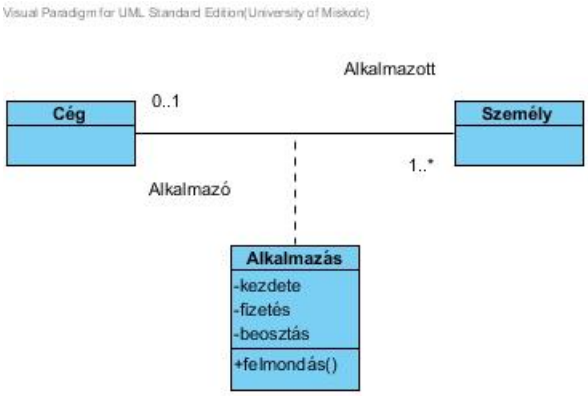
9.9. ábra - Szerepkör minősítője

Ebben a példában a minősítő egyértelműen meghatároz egy tárgyat.

A kapcsolat lehet elég összetett ahhoz, hogy önálló adatokkal és funkciókkal rendelkezhet. Ebben az esetben a kapcsolathoz egy, a kapcsolatot kezelő osztály rendelhetünk. Ebben az osztályban foglalhatjuk össze azokat az adatokat és viselkedésformákat, amelyek magára a kapcsolatra jellemzők. Az ilyen osztályt az asszociációt jelző vonalhoz szaggatott vonallal kapcsoljuk.

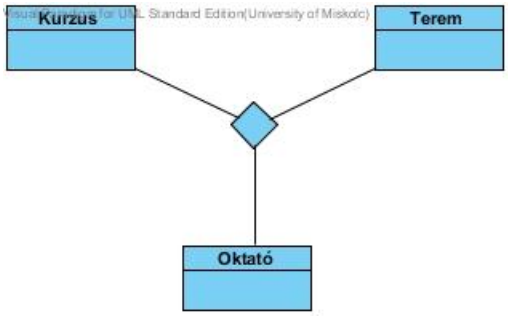
Példa:

9.10. ábra - Asszociációs osztály



Az eddigi példákban az asszociáció mindig két osztályt kötött össze (bináris asszociációk). Vannak azonban olyan esetek, amikor három osztály vesz részt egy kapcsolatban egyenrangú félként (ternáris asszociáció). Jelölése:

9.11. ábra - Ternáris asszociáció



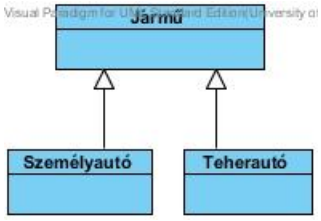
Az ilyen asszociációk nehezen implementálhatók, ezért elsősorban elemzési osztálydiagramokban fordulnak elő. Legkésőbb a megvalósítási szintű diagram készítésekor ezeket át kell alakítani implementálható formára.

1.3.2. Általánosítás

A programozási nyelvekből is ismert öröklődési mechanizmus, amelyet az UML általánosításként ismer. Jele egy kitöltetlen, zárt nyílvégű nyíl (amint azt már a használati esetek esetén is alkalmaztuk), amely a speciálisabb (leszármazott) elemtől az általánosabb (ős) osztály felé mutat. Egy ősnek tetszőleges számú leszármazottja lehet, és egy osztály tetszőleges számú ős leszármazottja lehet (többszörös öröklődés). (Az osztályok közötti többszörös öröklődést nem minden programozási nyelv támogatja.)

Jele:

9.12. ábra - Általánosítás jelölése



1.3.3. Tartalmazás

A tartalmazásnak (egész-rész viszonynak) két alapvető formáját különböztethetjük meg:

1. kompozíció: a tartalmazott önmagában nem létezhet, csak valaminek a részeként,

2. aggregáció: a rész hozzátartozik valamihez (esetleg csak időlegesen), de önállóan is létező entitás.

A kétféle viszony nem mindig könnyen különböztethető meg egymástól. Az alábbi néhány szempont segíthet ebben.

Kompozíció. Példa: tartalmazó - képernyő ablak, rész - gördítő sáv.

1. A rész soha nem jöhet létre a tartalmazó előtt, és biztosan megszűnik a tartalmazóval együtt. A tartalmazó életciklusán belül létrejöhet és meg is szűnhet.

2. A kompozíciós egység soha nem lehet egy időben két tartalmazó része.

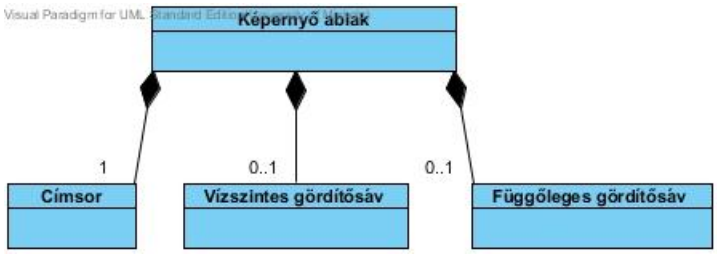
Aggregáció. Példa: Egy autó motorja és kerekei. Ezek alkatrészként önállóan is léteznek (cserélhetők, megvásárolhatók).

1. A rész előbb is létrejöhet, mint a tartalmazó, és életciklusa során válhat részévé a tartalmazónak, de el is válthat attól. (Pl. téli-nyári gumicsere). A résznek legkésőbb a tartalmazottá válás pillanatában kell létrejönnie, és nem szűnhet meg addig, amíg tartalmazottként funkcionál. A tartalmazó megszűnése nem jelenti egyben a tartalmazott megszűnését is.

2. Az aggregációs egység egyszerre több tartalmazónak is lehet része. (Az autós példa erre éppen nem jó minta, de ha mondjuk egy céget az alkalmazottak összességeként modellezzük, akkor egy személy egy szerre több cégnek is lehet része.)

A kompozíciót a tartalmazó felőli végén fekete rombusszal végződő vonal jelöli.

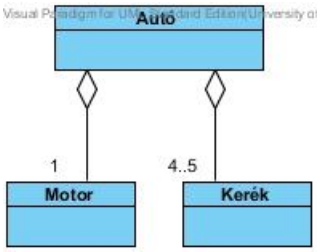
9.13. ábra - Kompozíció jelölése



A fenti ábra azt fejezi ki, hogy egy képernyő ablaknak része pontosan egy címsor, és része lehet (lásd a 0..1 számosságot) a gördítő sáv. A gördítő sáv az ablak életciklusa alatt számtalanszor létrejöhet és megszűnhet, ahogyan azt a valóságban is tapasztaljuk.

Az aggregációt a fentihez hasonlóan, de kitöltés nélküli rombusszal jelöljük.

9.14. ábra - Aggregáció jelölése



1.4. Parametrizált osztály

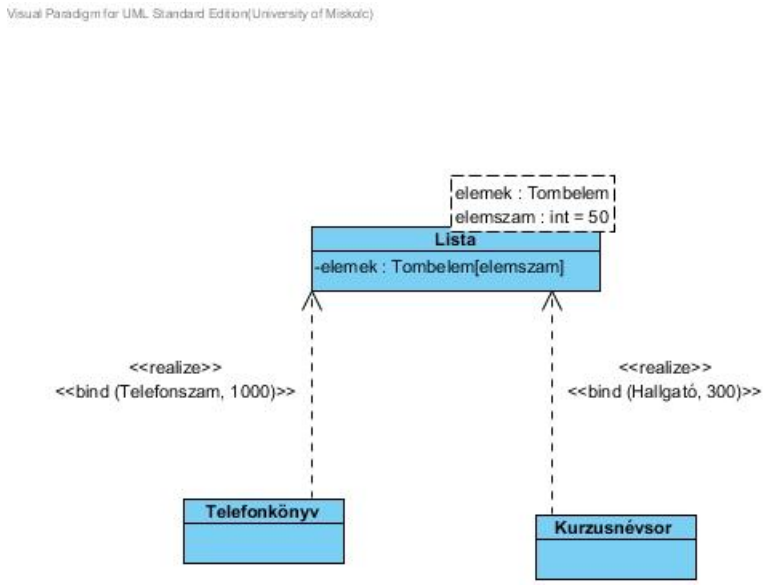
Másik magyar elnevezése: sablon osztály.

Számos programozási nyelv támogatja a polimorfizmusnak az a módját, amikor egy osztálydefiníció típus (és esetlen érték) paramétereket tartalmazhat. Az ilyen osztályból nem lehet példányosítani, hanem a paraméterek megadásával konkrét osztálydefiníciókat hozhatunk létre (ez a konkretizálás vagy realizáció művelete). A parametrizált osztály tehát osztály létrehozására szolgáló sablon.

A parametrizált osztály UML jelölése az osztály téglalapját kiegészíti egy szaggatott vonallal rajzolt téglalappal a paraméterek jelölésére. A konkretizálást a sablon osztály felé mutató szaggatott vonallal rajzolt nyíl jelöli. A nyilat a «realize» sztereotípia minősíti. A konkretizáláshoz használt paramétereket a «bind» sztereotípiával adhatjuk meg.

Példa:

9.15. ábra - Parametrizált osztály és konkretizálása



A fenti ábrán a Lista sablonosztály paramétere a Tombelem típus, és az n egész szám. A típus helyére Telefonszam típust helyettesítve a Telefonkönyv osztályt, Hallgató típust helyettesítve a kurzusnévsor osztályt kapjuk.

A parametrizált osztály elsősorban a tervezési és a megvalósítási modellben alkalmazható.

1.5. Absztakt osztály

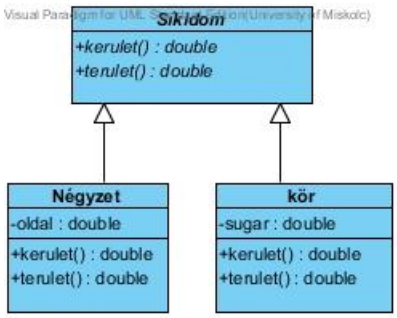
Az absztrakt osztály olyan osztály, amely tartalmaz olyan operációkat, amelyeket specifikálunk, de nem definiáljuk a megvalósítás módját. Ezért az ilyen osztály nem példányosítható, de lehet ős osztály. A leszármazott osztály fogja a hiányzó operáció definíciókat meghatározni.

Az absztrakt osztály nevét dőlt szedéssel írjuk, vagy fölé az «abstract» sztereotípiát írjuk.

A hiányzó definíciójú (tehát szintén absztrakt) operációkat is dőlt betűs szedéssel különböztetjük meg a definiáltaktól.

Példa:

9.16. ábra - Absztrakt osztály és leszármazottai



A fenti ábrán a Sikidom absztrakt osztály, mert bár tudjuk, hogy van kerülete és területe, de nem tudjuk megmondani, hogy azt általánosságban hogyan tudjuk kiszámítani. Ezért a két operáció is absztrakt. A leszármazottak konkrét síkidomok, a megfelelő adatokkal, így a kerület és terület számító operációknak konkrét definíciót tudnak adni.

Az absztrakt osztály a tervezési és az implementációs modellben alkalmazható.

1.6. Interfész

Az interfész hasonlít az absztrakt osztályokhoz, de nem tartalmazhat attribútumokat, sőt konkrét operációkat (műveleteket) sem, csak viselkedés mintákat (absztrakt operációkat). Ilyen elemre az analízis modell során nincs szükségünk, mert jellegzetesen technikai tulajdonságokat írhatunk le segítségükkel.

Az interfész által előírt viselkedésmintákat az adott interfészt megvalósító (realizáló) osztály definiálja. Ennek az osztálynak az interfész által előírt valamennyi operációt definiálnia kell.

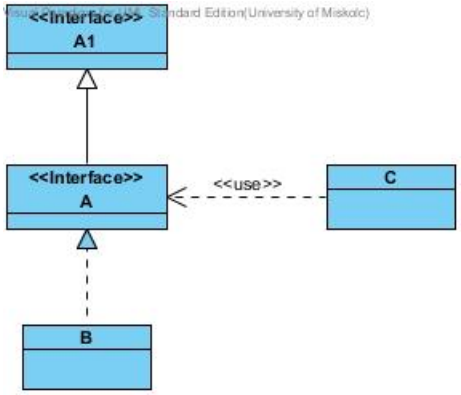
Más osztályok használhatják az adott interfészt, ami a használó és a megvalósító osztály közötti közvetett kapcsolatot jelent.

Interfészek között jelölhető az általánosítás kapcsolat ugyanolyan szabályokkal, mint az osztályok között.

Az interfészt ugyanolyan téglalappal jelöljük, mint az osztályt, csak a neve felé az «interface» sztereotípiát írjuk.

További jelöléseket láthatunk az alábbi ábrán:

9.17. ábra - Interfész, interfészek közötti öröklődés, interfészt implementáló és használó osztály jelölése



A fenti ábrán az A interfészt a B osztály implementálja, a C osztály pedig használja. Az A interfész az A1 leszármazottja.

Egy osztály tetszőleges számú interfészt implementálhat egy időben. Ilyenkor minden interfész által előírt operációt meg kell valósítania.

Egy osztály lehet egyszerre leszármazottja tetszőleges számú osztálynak, és implementátora tetszőleges számú interfésznek.

Az interfész a tervezési és a megvalósítási modellben alkalmazható.

2. A csomag diagram

A csomag diagram alapvetően más modell elemek csoportosítására használható, és mint ilyen, szintén minden fejlesztési fázisban használatos. A használati eset modell elkészítése során a használati eset leltár esetén már említésre került, ebben a fejezetben összefoglaljuk a részletes szabályokat.

A csomag jele az UML-ben egy „füles” téglalap. A fül felirata lehet a csomag neve, de írható a téglalapba is. A csomagnév fölött sztereotípiát adhatunk meg, alatta megszorítást.

A csomag tartalma lehet bármilyen szövegesen vagy diagrammal megadott modell elem, illetve azok halmaza. A korábban már említett használati eset leltár például használati eseteket tartalmazó csomagokból áll.

A csomagok újabb csomagot is tartalmazhatnak. Egy modell elem csak egy csomaghoz tartozhat. Így a csomagok fa struktúrájú hierarchiát alkotnak, hasonlóan, mint a filerendszerek elemei, vagy a Java csomagok.

A «system» sztereotípiával jelzett csomag a legfelső szintű, az egész alkalmazást jelképező csomag.

A csomagok közötti függőségek szaggatott, nyitott hegyű nyíllal ábrázolhatjuk. A kapcsolat jellegét sztereotípiával pontosíthatjuk.

Az UML csomag egyben egy névteret is képvisel, ezért egy csomagon belül nem lehet két azonos nevű modell elem. Ha egy másik csomagban levő modell elemre akarunk hivatkozni, azt a minősített nevével tehetjük meg. A minősített név tartalmazza azon csomagok neveit, amelyeken keresztül eljuthatunk az adott modell elemig. A csomagneveket a :: választja el egymástól.

A csomag elemeihez public(+) és private (-) láthatóságot is rendelhetünk, hasonlóan az osztályok elemeihez.

A minősített névvel való hivatkozás nehézkes lehet a hosszúsága miatt, és minden ilyen hivatkozásra hatással lehet a csomagok átszervezése. Ezért helyette egyszerűbb és célszerűbb a csomagok közötti hivatkozásokat a függőséget jelölő (szaggatott vonallal rajzolt) nyíllal és sztereotípiákkal jelölni (importálás). Az importált elem az importáló csomag számára látható lesz. A használható sztereotípiák:

«import»: nyilvános import, az importáló csomag tovább exportálhatja

«access»: privát import, csak az importáló csomag láthatja.

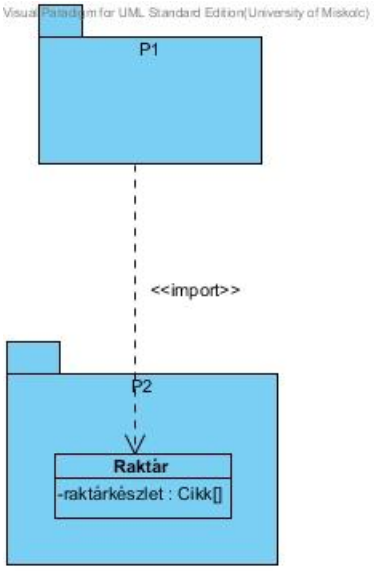
Jelölése:

1. Az importáló csomagtól közvetlenül az importált elemre mutató szaggatott nyíl

2. Az importáló csomagtól az importált elemet tartalmazó csomagra mutató nyíl, a sztereotípia után írva az importálandó elem neve

3. A két csomag között rajzolt, elemnév nélküli nyíl a teljes csomag valamennyi elemének importálását jelenti.

9.18. ábra - Elem importálása



3. Komponens diagram

A komponens az UML-ben egy olyan univerzális egység, amely valamilyen szolgáltatás halmazt képvisel, azokat egy egységbe zárja. A szolgáltatásokat más komponensek interfészeken keresztül érhetik el. Egy komponens kicserélhető egy másikkal, ha ugyanolyan interfésszel rendelkezik, és ugyanazokat a szolgáltatásokat nyújtja.

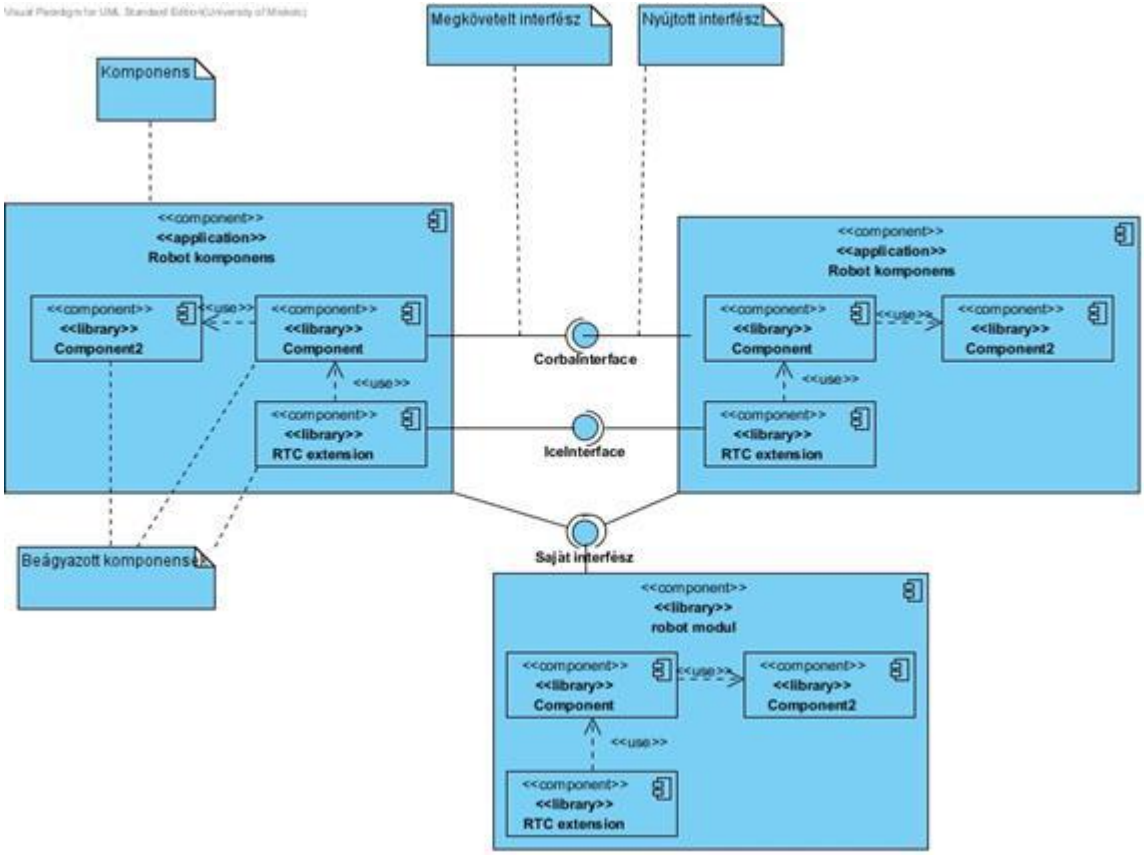
Egy komponens tartalmazhat más komponenseket.

A komponensek fogalma jól használható a fejlesztés minden fázisában. Így jelenthet egy implementációs értelemben vett komponenst (például egy Enterprise JavaBean-t), de akár egy komplett alrendszert is (például könyvelés).

A komponens jele egy téglalap a «component» sztereotípiával, vagy egy ikonnal megjelölve. A komponens a külvilággal csatlakozókon (port) keresztül teremt kapcsolatot. Minden csatlakozóhoz egy vagy több interfész kapcsolható. A külvilág felé szolgáltatásokat nyújtó interfészt (nyújtott interfész) egy körrel, a komponens által másoktól igényeltet (megkövetelt interfész) egy félkörrel jelöljük. A komponensek közötti kapcsolatokat az interfészek összeillesztésével ábrázolhatjuk. Egy nyújtott interfész csak egy megkövetelt interfésszel állhat kapcsolatban. (Ezt jól szemléltetik a jelölések is.)

A komponens diagram jelöléseit szemlélteti az alábbi ábra, amely az esettanulmány komponens diagramjának egy részlete.

9.19. ábra - Komponens diagram



4. Telepítési diagram

A telepítési diagram feladata a működő szoftver rendszer alkotóelemeinek az azokat működtető hardver-szoftver elemek összerendelése, és a működtető elemek közötti kapcsolatok ábrázolása.

A szoftver elemei lehetnek:

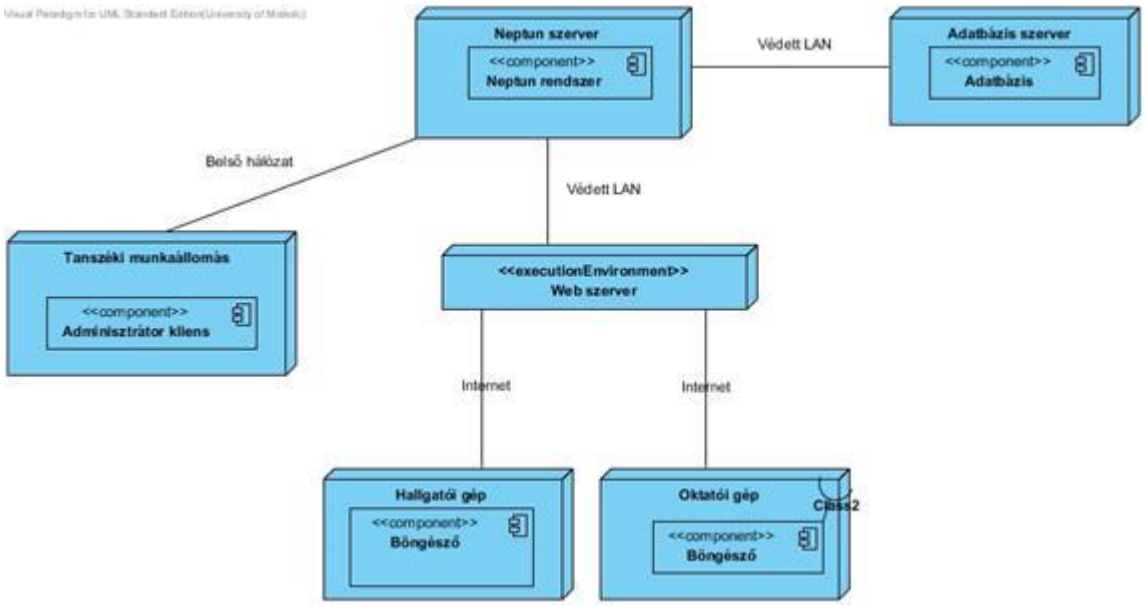
1. végrehajtható program modulok
2. beállítások, konfigurációs file-ok
3. adatok

A működtető elemek lehetnek:

1. számítógépek, hálózati csomópontok
2. végrehajtási környezetek (virtuális gép, alkalmazás szerver stb.)

Példaként rajzoljuk meg egy tanulmányi nyilvántartó rendszer egy lehetséges telepítési diagramját.

9.20. ábra - Telepítési diagram



A fenti ábra szerint egy lehetséges rendszer felépítés, hogy a Neptun szerver és az adatbázis szerver egy szigorúan védett belső hálózatba vannak kötve. A tanszéki adminisztrátor egy telepített kliens segítségével, az intézményi belső hálózaton keresztül éri el közvetlenül a Neptun szervert. A hallgatók és az oktatók egy Web szerveren át, internetes elérést kapnak a rendszerhez.

10. fejezet - Viselkedés diagramok

A rendszer statikus nézetének megalkotása után a dinamikus nézet elemzése következik. A dinamikus nézetet az UML viselkedés diagramjai segítségével írhatjuk le. Ebben a fejezetben összefoglaljuk a legfontosabb viselkedés diagramokat.

1. Szekvencia diagram

A szekvencia diagram feladata objektumok egymás közti üzenetváltásainak ábrázolása egy időtengely mentén elhelyezve.

A diagram elemei:

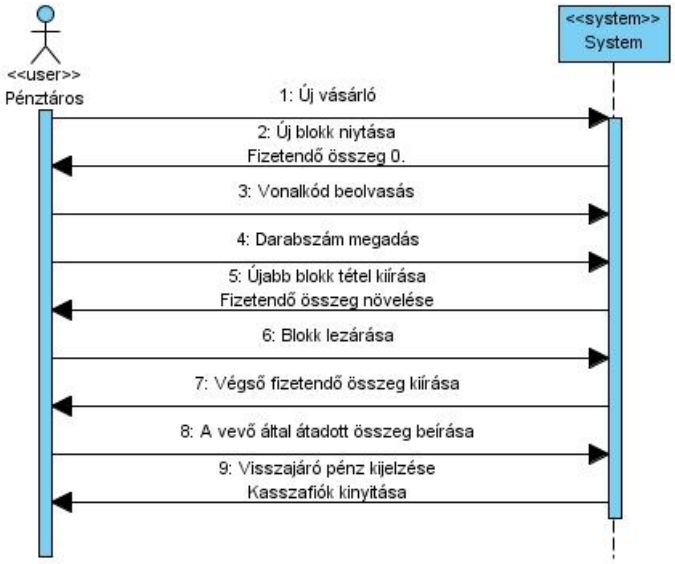
1. Objektumok, amelyektől egy szaggatott vonallal jelzett életvonal indul, amely felülről lefelé az idő múlását jelképezi. Az életvonalon jelölhetők az objektum aktivitási szakaszai. Az aktivitási szakaszt a szaggatott vonal helyett elnyújtott téglalap jelzi.

2. Üzenetváltások. Az egyes objektumok közötti üzenetváltásokat (interakciókat) a küldő életvonalától a fogadóéig rajzolt nyíllal jelöljük. A nyílra az üzenet elnevezését írjuk, de feltüntethetők az üzenet paraméterei (az üzenet elnevezése után zárójelben, mint az operációk paraméterei), illetve az üzenethez kapcsolódó feltételek is. Az üzenet továbbítás ideje általában nullának tekinthető, ezért a nyilak vízszintesek. A nyilak formája az üzenetváltás módjára utal.

3. A diagram bal szélén megjegyzések, megszorítások és időbeliségre utaló jelölések helyezhetők el.

Egy egyszerű példa a szekvencia diagramra a „Blokkolás” használati eset (lásd 8. fejezet) forgatókönyvének részletezése szekvencia diagram formájában. (A szekvencia diagramot a Blokkolás használati eset részletes leírásából a Visual Paradigm modellező eszköz generálta automatikusan.)

10.1. ábra - A Blokkolás használati eset forgatókönyve



1.1. Objektumok

Az „objektum” kifejezés a szekvencia diagram szempontjából a programozási nyelvek objektum fogalmához képest általánosabb értelmű is lehet: valójában egy rendszer valamilyen működésre képes részét, vagy – mint a fenti példa is mutatja – akár az egész rendszert jelenheti. Ha valóban implementációs szintű objektumról van szó, akkor a szokásos objektum jelölést használjuk a téglalapban.

Speciális „objektumként” jelenhetnek meg a rendszer aktorai, hiszen egy üzenetváltás gyakran az aktor és a rendszer (vagy annak valamely objektuma) közötti interakcióval kezdődik, vagy az üzenet egy aktornak szól.

1.2. Üzenetek

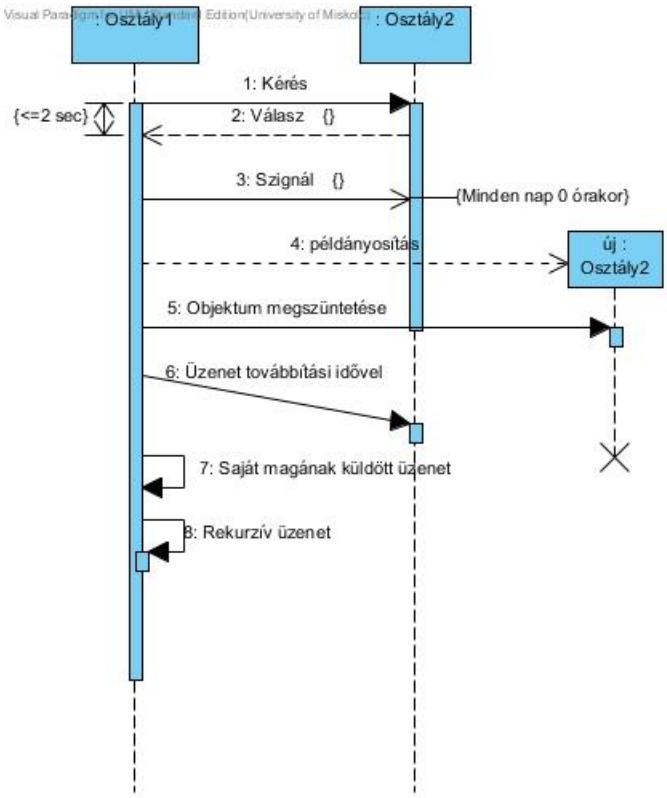
Bár a fenti példa ábrán csak egy féle üzenet szerepel, lehetőségünk van több fajta üzenet ábrázolására.

Az üzenetek fajtái:

1. Szinkron üzenet (kérés): a küldő elküldi az üzenetet, majd vár a válaszra. A fogadó aktiválódik (ha nem volt az).
2. Válasz üzenet: mindig az előző üzenetre vonatkozik. A választ küldő deaktiválódik, a fogadó pedig aktiválódik.
3. Aszinkron üzenet (szignál): a küldő elküldi az üzenetet, de folytatja a munkáját, nem vár válaszra.
4. Objektum létrehozása: az üzenet hatására létrejön egy új objektum. A nyíl ilyenkor az objektum fejre mutat.
5. Objektum megszüntetése: az üzenet hatására egy objektum megszűnik. Az életvonal végét egy X zárja le.
6. Üzenet nem nulla továbbítási idővel: ha ki akarjuk hangsúlyozni, hogy egy üzenet továbbítása időt vesz igénybe, ferde nyilat használunk.
7. Saját magának küldött üzenet: egy objektum küldhet saját magának üzenetet.
8. Rekurzív üzenet: az objektum saját magának küld üzenetet. Ennek kiszolgálására az objektum fő tevékenysége felfüggesztődik a kiszolgálás idejére. („Beágyazott aktivitási szakasz”)

Az alábbi ábra az egyes üzenetfajták jelöléseit foglalja össze:

10.2. ábra - Üzenet fajták jelölése



1.3. Üzenetek időbelisége

Az objektumok életvonala egy felülről lefelé mutató időtengelyt is képvisel. Az időbeliség azonban alapesetben csak sorrendiséget jelent: amelyik üzenet nyila lejjebb található, az követi a felé rajzolt üzenetet. A lefelé lejtő nyíllal jelölt üzenet továbbítása „valamennyi időt” vesz igénybe.

Az életvonal elvileg valódi időskálával rendelkező időtengellyé is alakítható, ahol a nyilak távolsága az egyes üzenetek között eltelt időt is jelezheti. Ha azonban az üzenetek pontos időbeli elhelyezkedése lényeges (egy valós idejű rendszerben például ez nagyon fontos lehet), célszerűbb az UML 2.0-tól bevezetett idődiagramot használni. (A jegyzet terjedelmi korlátai miatt az idődiagram ismertetésével nem foglalkozunk.)

A szekvencia diagramon is ábrázolhatók azonban megszorításokkal az egyszerű sorrendiségtől pontosabb időkorlátok, mint például két üzenet közötti maximális megengedett időtartam, vagy egy üzenet időpontja. A fenti ábrán például jelöltük, hogy az 1. üzenetre adott válasznak maximum 2 másodpercen belül meg kell érkeznie, illetve hogy a 3. üzenetet minden nap 0 órakor el kell küldeni.

1.4. Interakciós operátorok

Az UML 2.0-ás verziótól kezdve a nyelv bevezette a „komplex interakció” fogalmát, azaz az üzeneteket (interakciókat) kombinálhatjuk, és a köztük levő viszonyokat operátorokkal fejezhetjük ki.

Egy komplex interakció jele egy téglalap. A téglalap bal felső sarkában a rá vonatkozó operátort tüntetjük fel, vízszintesen pedig „átfogja” az interakciókban érintett objektumok életvonalait.

AZ UML számos interakciós operátort definiál, ebben a jegyzetben csak néhány, gyakran használatos operátort nézünk meg. A többiről jó leírást találunk a Störle irodalomban.

1.4.1. A ref operátor

Gyakran előfordul, hogy egy interakció sorozat több esetben is része a folyamatoknak. Ilyenkor ezeket célszerű egy külön szekvencia diagramban ábrázolni, és mindenütt, ahol szüksége van rá, a ref operátorral hivatkozhatunk rá. A ref operátor tehát egyfajta interakció makró, de paraméterezhető, és visszatérési érték is rendelhető hozzá.

1.4.2. A loop operátor

A téglalapba foglalt interakció sorozat ismétlődését jelenti. Egyedi interakciókat és valamilyen operátorral azonosított komplex interakciókat is tartalmazhat, tehát a komplex interakciók egymásba ágyazhatók.

1.4.3. A strict operátor

Azt írja elő, hogy a befoglalt interakciók (az operandusai) szigorú sorrendben követik egymást. Az operandusait szaggatott vonal választja el egymástól. Mivel egy egyszerű interakciókra ez az alapértelmezés, akkor van értelme használni, ha az operandusai között van legalább egy operátorral ellátott összetett interakció.

A szigorú sorrend azt jelenti, hogy egy operandus végrehajtása csak akkor kezdődhet el, ha az előző végrehajtása teljesen befejeződött.

1.4.4. Feltételes végrehajtás operátorai

Ezek az operátorokhoz egy feltétel is kapcsolódik, és az operandusaik a feltétel teljesülésétől függően futnak le.

opt: az operandus opcionálisan felléphet

alt: a feltételnek megfelelő eset választódik ki a lehetséges alternatívák közül

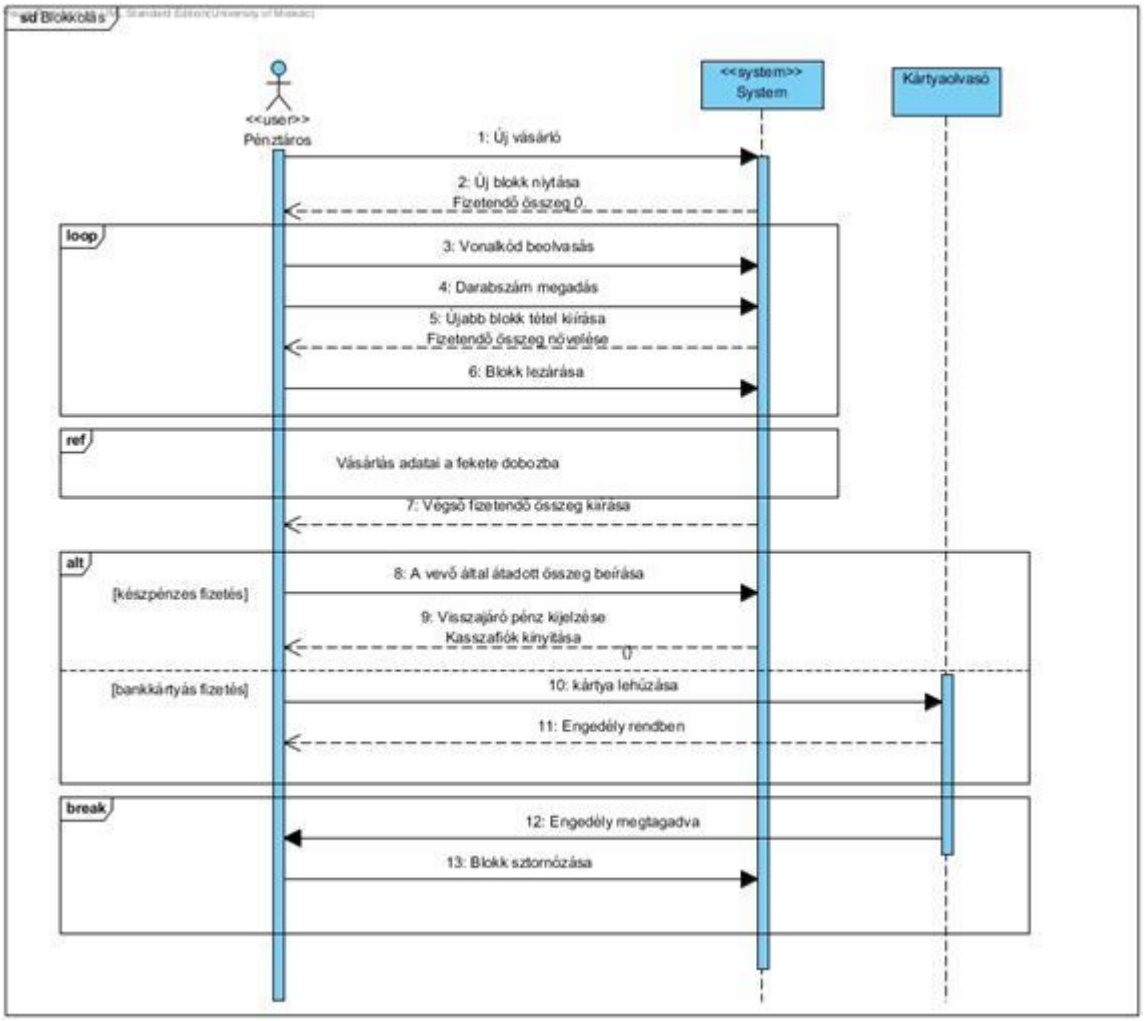
break: a tartalmazó interakciót megszakítja. Valamilyen kivételes körülmény bekövetkezését kezelhetjük segítségével.

1.5. Példa: a Blokkolás forgatókönyvének pontosítása

Az eddig elmondottak ismeretében dolgozzuk át a fejezet első ábráján található generált szekvencia diagramot. A diagram megfelel egy analízis modell részeként, de a tervezési fázis számára már túlságosan elnagyolt, pontosítani, bővíteni kell.

Egy lehetséges továbbfejlesztés például az alábbi:

10.3. ábra - Kiegészített szekvencia diagram



A kiinduló diagramhoz képest az alábbi változtatásokat tettük:

1. A válasz üzenetek az ábrában csak számozással voltak kiemelve, ehelyett használjuk a szaggatott vonallal rajzolt nyilakat.

2. Az eredeti ábrából nem derül ki, hogy az egy tétel blokkolásához szükséges üzenetváltásoknak ismétlődniük kell. Ezt tényt a most a loop operátorral jelöltük.

3. A blokk lezárása után a blokk adatait a fekete doboz számára gyűjteni kell. Ennek a folyamatnak a részleteit egy másik („Vásárlás adatai a fekete dobozba” nevű, a jelen jegyzetben nem részletezett) szekvencia diagramban ábrázoljuk, a ref operátor erre csak hivatkozik. A hivatkozott diagramnak természetesen a modellben léteznie kell. (Ezt az ábrát csak úgy lehetett elkészíteni, hogy felvettünk egy üres szekvencia diagramot a kívánt névvel.)

4. Kiegészítettük a kiindulási diagramot azzal, hogy készpénzes és bankkártyás fizetést is megengedünk. Ehhez fel kellett venni egy újabb objektumot, a Kártyaolvasót. A két fizetési mód közötti választási lehetőséget az alt operátorral jeleztük. Az alternatív futási lehetőségeket a megfelelő feltétellel („guard”) jelöltük. A két lehetőség közül pontosan az egyik történik meg a valóságban.

5. Kiegészítettük a kiindulási diagramot azzal a kivételes esettel, ha a vásárló kártyája nem alkalmas a fizetésre (a bank nem adja meg az engedélyt, például fedezet hiány miatt). A break operátorral megjelölt folyamat csak ebben az esetben hajtódik végre, és a befoglaló interakciót (most a teljes Blokkolás folyamatot) megszakítja.

6. Figyeljük meg, hogy az egyes operátorokhoz tartozó téglalapok nem azonos szélességűek: mindegyik csak azokat az életvonalakat fogja át, amely objektumoknak szerepük van az operandusaik végrehajtásában.

2. Kommunikációs diagram

A kommunikációs diagram, a szekvencia diagramhoz hasonlóan, objektumok közötti üzenetváltásokat képes ábrázolni, azonban ebben az esetben nem az üzenetváltások időbeliségére helyezzük a hangsúlyt, hanem az üzenetváltásokban részvevő objektumok közötti kapcsolatokra. A szekvencia diagramon az objektumok kapcsolata csak közvetve jelenik meg, azáltal, hogy mely elemek között van interakció. A kommunikációs diagramon az együttműködő objektumok az osztálydiagramoknál már megismert, asszociációkat jelző vonalakkal (nyilakkal) vannak összekötve. Az üzenetek időbeli sorrendjét az üzeneteket számozása jelzi.

A kommunikációs diagramon is szerepelhet aktor, mint az üzenet forrása vagy címzettje.

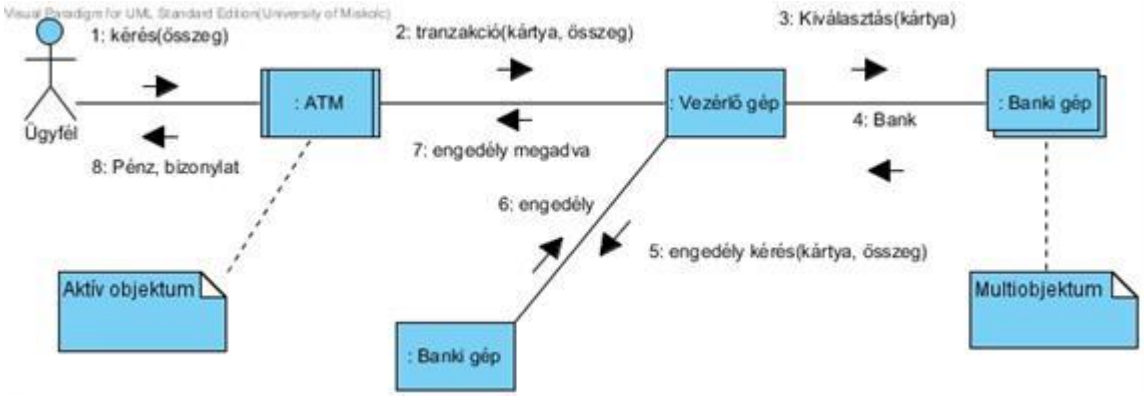
Jelölni lehet, ha egy objektum aktív (önállóan képes üzenetet küldeni), illetve ha egy üzenet nem egy objektumhoz, hanem azonos osztályhoz tartozó objektumok egy halmazához irányul (multiobjektum).

Példaként rajzoljuk meg egy bank automatából való készpénzfelvétel kommunikációs diagramját. Feltételezzük, hogy az ügyfél a kártya beolvasásával és a PIN lód megadásával már azonosította magát, és a lehetséges funkciók közül kiválasztotta a készpénzfelvételt.

A kézpénzfelvétel forgatókönyve:

Az ügyfél megadja a felvenni kívánt összeget. A összeget és a kártya adatait az ATM továbbítja az automatákat vezérlő gép felé. A vezérlő gép a kártya adatok alapján kiválasztja a megfelelő bankot, és annak a gépére továbbítja az adatokat. A banki gép ellenőrzi a tranzakciót, és megadja az engedélyt, ami a vezérlő gépen keresztül eljut az ATM-hez. Ennek hatására az automata kiadja a pénz és a bizonylatot az ügyfélnek.

10.4. ábra - Készpénzfelvétel kommunikációs diagramja



Az ábrán az ATM aktív objektumként van jelölve. Bár ez most inkább a jelölés szemléltetését szolgálja, indokolható azzal, hogy hangsúlyozzuk azt a tényt, hogy az ábrán nem szereplő előző műveletek (ügyfél azonosítás) már megtörténtek, és ezek hatására az ATM aktív állapotba került.

A vezérlő gép a 3. üzenetet egy multiobjektumnak küldi. Ezzel tudjuk kifejezni, hogy a kártya adatok alapján először a megfelelő bankot kell a vezérlő gépnek kiválasztania, és ennek megtörténte alapján tudja az engedélykérést a bank felé továbbítani.

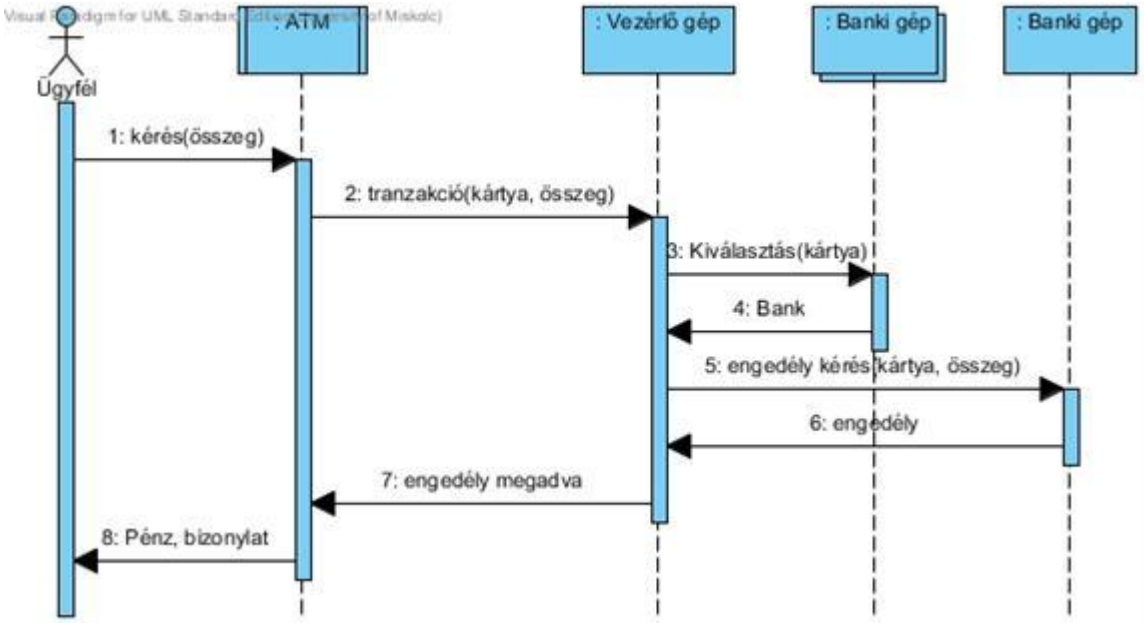
Az üzenetek számozása az időrendjüket határozza meg. Az egyidejűséget a sorszám után betűkkel jelölhetjük. Például a 2a: és 2b: üzenetek egyidejűek, és az 1: után, de a 3: előtt kerülnek elküldésre.

2.1. A szekvencia és a kommunikációs diagram összehasonlítása

Mindkét diagram típus interakciók ábrázolására szolgál, de a kommunikációs diagram esetén jobban ábrázolható az üzenetek objektumok közötti terjedésének módja, és az objektumok ehhez szükséges függőségi rendszere. Minden kommunikációs diagram átkonvertálható szekvencia diagramba. Fordítva ez csak a komplex interakciókat nem tartalmazó szekvencia diagramokra igaz.

Az előbbi kommunikációs diagram például az alábbi szekvencia diagrammal egyenértékű:

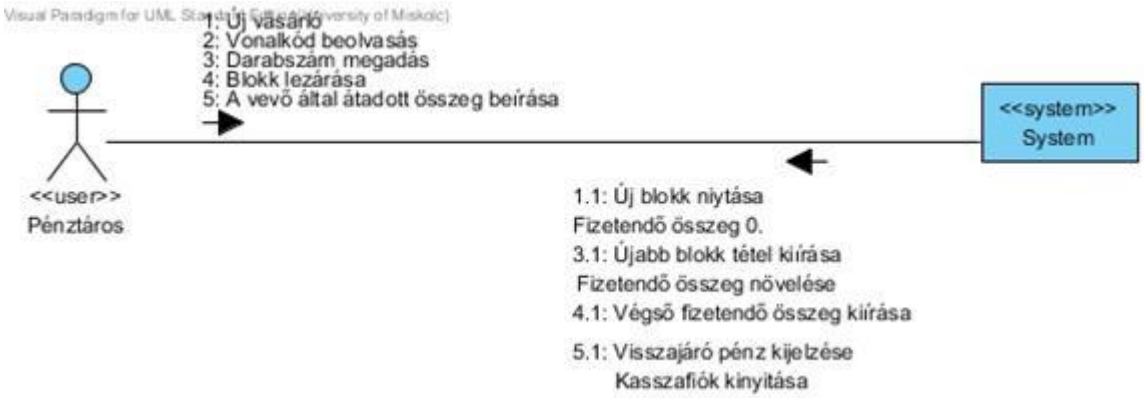
10.5. ábra - A készpénzfelvétel szekvencia diagram formájában



Ebben az esetben mindkét diagram fajta szemléletes módon ábrázolja a folyamatot.

A blokkolás egyszerűsített szekvencia diagramjának kommunikációs diagramra konvertált formája:

10.6. ábra - A blokkolás forgatókönyve kommunikációs diagramon



Láthatóan ebben az esetben ez a forma sokkal áttekinthetetlenebb. Ennek oka, hogy két modell elem között túl sok üzenetváltás történik, ilyenkor a szekvencia diagram szemléletesebb. Fordítva is igaz lehet: ha sok objektum szerepel a folyamatban, de azok között viszonylag kevés interakció zajlik, a kommunikációs diagram lehet célszerűbb.

A blokkolás kibővített szekvencia diagramja nem konvertálható át, mert komplex interakciókat tartalmaz.

3. Állapotgép diagram

Ez a diagramfajta nem az UML találmánya, eredetileg David Harell dolgozta ki, amelyet az UML beillesztett a saját rendszerébe, az ehhez szükséges módosításokkal és kiegészítésekkel.

Az állapotgép diagram egy osztály objektumainak az életciklusuk alatt felvehető lehetséges állapotait és az állapotok közötti lehetséges átmeneteket ábrázolja.

3.1. Az állapot fogalma

Az objektum orientált szemlélet szerint egy objektum állapotát az attribútumainak a pillanatnyi értékhalmaza határozza meg (konkrét állapot). A modellezés során egy bizonyos szempontból a konkrét állapotok meghatározott halmazai egyenértékűnek tekinthetők, egy állapot jelleget határoznak meg. Ezt absztrakt állapotnak nevezzük. Például egy hallgatónak abból a szempontból, hogy teljesítette-e a Szoftver technológia tárgyat, két absztrakt állapota lehet:

1. teljesítette: aláírása és legalább elégséges vizsgajegye van (ez négy lehetséges konkrét állapotot fog össze),

2. nem teljesítette: nincs aláírása, vagy van aláírása, de nincs vizsgajegye, vagy elégtelen vizsgajegye van (ami három lehetséges konkrét állapot).

Ez a két absztrakt állapot elegendő ahhoz, hogy eldönthessük, egy ráépülő tárgyat felvehet-e. (Bár természetesen más szempontból nem lényegtelen, hogy a vizsgajegye elégséges vagy jeles.)

Az állapotgép diagram esetén általában absztrakt állapotokkal foglalkozunk.

3.2. Az állapot átmenet és jelölése

Az állapotok közötti átmenetet valamilyen esemény bekövetkezése okozza. Az átmenetek atomi egységek, nem szakíthatók félbe, azaz időponthoz kötöttek. Az átmenetet nyitott hegyű nyíllal jelezzük. Az állapotok adatai lehetnek:

1. kiváltó ok (trigger) az esemény megnevezése

2. feltétel (guard) szögletes zárójelpárral határolt, az átmenet bekövetkezésének feltételét adja meg,

3. hatás (effekt) az októl / jellel elválasztva adható meg, és az átmenet miatt végrehajtandó tevékenységet specifikálhatja.

A fentiek mindegyike opcionális. Ha nincs megadva kiváltó ok, akkor a kiváltó esemény az előző állapot befejeződése.

3.3. Az állapot jelölése

Az állapotnak az átmenettel szemben van időtartama, tehát egy adott állapotba az objektum belép, majd valamennyi idő után abból kilép, tehát az állapot időtartamhoz kötött. Egy adott absztrakt állapotban tartózkodó objektum ebben az időszakban több konkrét állapotot is felvehet. (Például a „tárgyat nem teljesítette” absztrakt állapotban állapoton belül marad a hallgató az aláírás megtagadva – aláírás pótolva – elégtelen vizsgajegy konkrét állapotokat felvéve.)

Minden állapothoz legalább egy átmenet vezet, és onnan legalább egy átmenet vezet egy másik állapotba. Lehetséges olyan esemény, amelynek hatására ugyanabba az állapotba tér vissza az objektum. (Például az elégtelen ismételt vizsga a „tárgyat nem teljesítette” állapotból ugyanabba az állapotba viszi vissza a hallható objektumot.)

A diagramon két speciális állapot jelenhet meg:

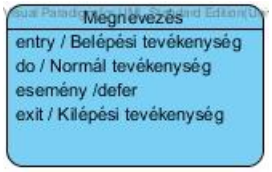
1. kezdőállapot: ebbe kerül az objektum, amikor létrejön. Ebbe az állapotba nem vezethet átmenet. A kezdőállapotot kitöltött körrel jelezzük.

2. végállapot: az objektum megszűnését jelzi. Ebből az állapotból nem indulhat ki átmenet. Egy körbe rajzolt ponttal jelöljük.

Bár a valóságban minden objektumnak van életciklusa (kezdő és végállapota), ez a tény a modellezés szempontjából lehet lényegtelen, tehát a fenti speciális állapotok hiányozhatnak a diagramról.

Az állapot jelölése egy kerekített sarkú téglalap, amelynek felső részében az állapot megnevezése található (ez kötelező) majd a vonal alatt a további elemei, melyek opcionálisak. A jelölés:

10.7. ábra - Állapot jelölése



Az opcionális elemek:

1. entry / Annak a tevékenységnek a leírása, ami az állapotba való belépés esetén elvégzendő. Ez a tevékenység mindig végrehajtódik, attól függetlenül, hogy milyen átmenet miatt lépett be az objektum az állapotba.

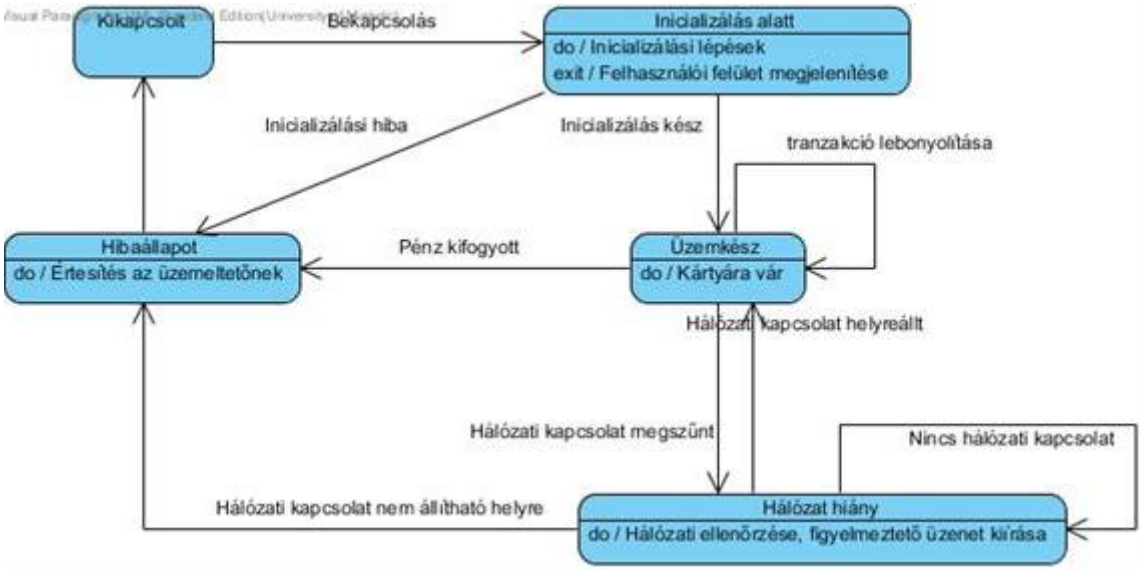
2. do/ Annak a tevékenységnek a leírása, amely az adott állapothoz tartozik. Ha több ilyen is van, azokat a végrehajtásuk sorrendjében soroljuk fel. Ezt a tevékenységet egy esemény félbeszakíthatja. Az esemény lehet:

3. esemény / Belső esemény bekövetkezéséhez kapcsolt tevékenység leírása.

4. exit / Annak a tevékenységnek a leírása, amelyet bármely átmenet bekövetkezése esetén még az állapotból való kilépés előtt végre kell hajtani.

Példaként, ha már a kommunikációs diagrammal kapcsolatban szóba került, rajzoljuk meg egy ATM állapotgép diagramját. (A valóságban ennél biztosan több állapota van, és nem biztos, hogy minden állapot átmenet az ábrának megfelelő, de a diagram jelöléseinek szemléltetésére ez is elegendő.)

10.8. ábra - ATM állapotdiagramja



Figyeljük meg, hogy ezen az ábrán nincs kezdő és végállapot feltüntetve, azaz az ATM objektumot „örökéletűként” modelleztük. A kezdő állapot a telepítést, a végállapot a leszerelést jelentené ebben az esetben, de ez a működés modellezése szempontjából nem lényeges.

3.4. Strukturált állapotgép diagram

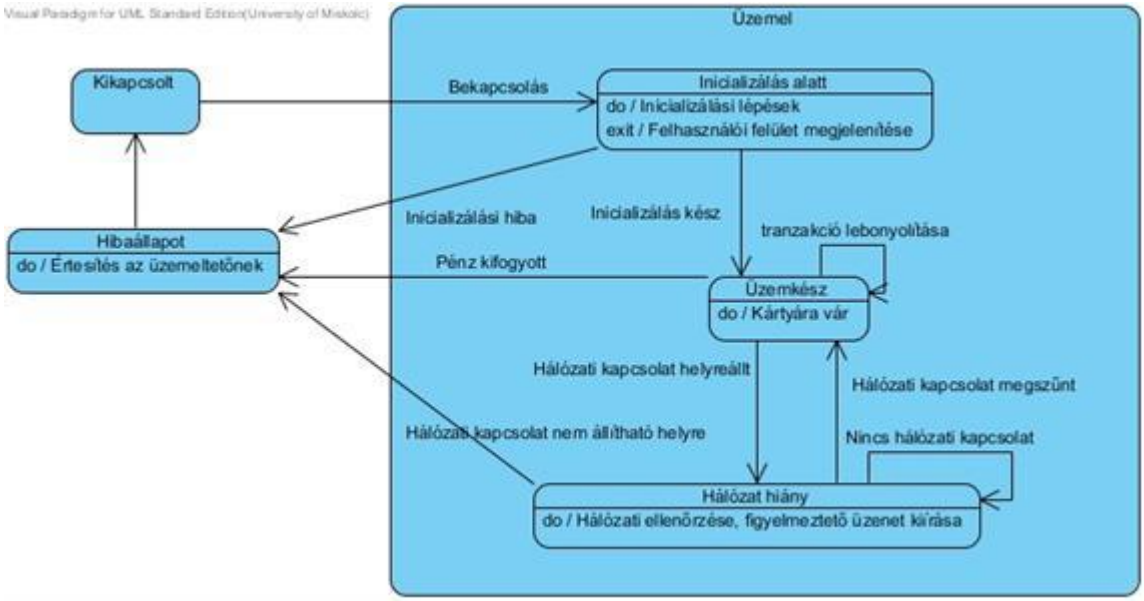
Bonyolultabb viselkedésű objektum esetén az állapotok és a köztük lehetséges átmenetek száma olyan nagy lehet, hogy a diagram áttekinthetetlenné válna.

Az áttekinthetőség érdekében ilyenkor az összetartozó állapotokat szuperállapotokba foglalhatjuk. A szuperállapotok tartalmazhatnak alállapotokat és azok közötti átmeneteket. A szuperállapotokat külön ábrába is foglalhatjuk.

A fenti ábrában például ilyen, külön részletezhető állapot az „Üzemkész”, hiszen az ATM viselkedésének pontos leírásához ezt egy külön diagramban kellene definiálni.

A strukturált diagram bemutatására rajzoljuk át a fenti ábrát úgy, hogy három fő állapotot különböztetünk meg: Kikapcsolt, Üzemel, Hibaállapot. Az Üzemel állapot foglalja magában az összes többi állapotot.

10.9. ábra - Struktúrált állapotgép diagram

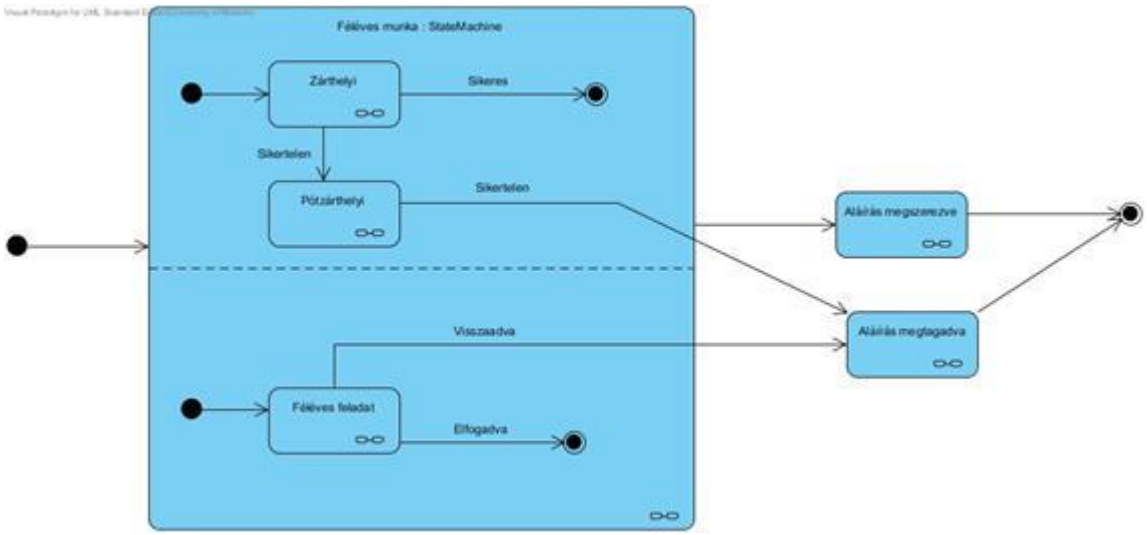


3.5. Konkurencia

Egy bonyolult viselkedésű objektum esetén előfordulhat, hogy az állapotait leíró attribútumai önállóan, egymással párhuzamosan változó részhalmazokra bonthatók. Ezt a párhuzamosságot úgy jelölhetjük, hogy az objektumnak egy szuperállapotot feleltetünk meg, amelyet vízszintesen szaggatott vonallal több részre osztunk. Minden részben egy önálló állapotgép diagram rajzolható, amelyek önálló kezdeti és vég állapotokkal is rendelkezhetnek, és amelyek konkurens módon, egymással párhuzamosan működnek. Az ilyen összetett szuperállapotból kivezető állapot átmenet az összes többi konkurens működést félbeszakítja. Az összetett állapot automatikus átmente akkor következik be, ha valamennyi rész állapotváltozás eléri a vég állapotot.

Példa: egy tárgy aláírásának megszerzéséhez egy félévközi zárthelyit kell megírni (amely pótolható) és egy féléves feladatot kell beadni Az aláírás feltétele mindkettő sikeres teljesítése. Az ennek megfelelő állapot diagram:

10.10. ábra - Konkurencia az állapotgép diagramban



A Féléves munka állapotból az Aláírás megszerezve állapotba csak mindkét (konkurens módon végrehajtott) állapot átmenet sorozat végállapotának elérése során lehet eljutni. Bármelyikből induló, a határon átlépő átmenet a másik folyamat megszakítását is jelenti, ezzel fejezhetjük ki, hogy a két feltételnek együttesen kell teljesülnie.

4. Aktivitás diagram

Az aktivitás diagram feladata időben lezajló változások, folyamatok ábrázolása a végrehajtandó tevékenységek és azok sorrendjének megadásával. Elődjei között megtalálható például a munkafolyamat (workflow) diagram és a folyamatábra, bár az UML-ben definiált diagram leginkább a Petri-hálókra támaszkodik.

Az aktivitás diagramokat gyakran használjuk a használati esetekben leírt forgatókönyvek működésének leírására, vagy akár egy operáció implementálási módjának definiálására, de alkalmas egy alrendszer vagy az egész rendszer működésének a szemléltetésére is.

A tevékenység diagram az állapotgép diagram egyfajta duálisaként is felfogható: míg az állapotgép esetén az állapotok állnak a középpontban, és az átmenetek alárendeltek, itt az átmeneteket megvalósító tevékenységeké a fő szerep. Ezért nem véletlen a két diagram típus jelöléseinek hasonlósága – bár ez néha problémát is okoz, amikor az egyes jelölések tartalma nem teljesen azonos.

4.1. Az aktivitás diagram alapelemei

A diagram alapelemei:

1. tevékenységek (jelölése lekerekített sarkú téglalap)

2. átmenetek (jelölése nyíl),

3. döntési pont (jelölése rombusz,)

4. kezdő- és végállapot (jelölése azonos az állapotgép diagraméval).

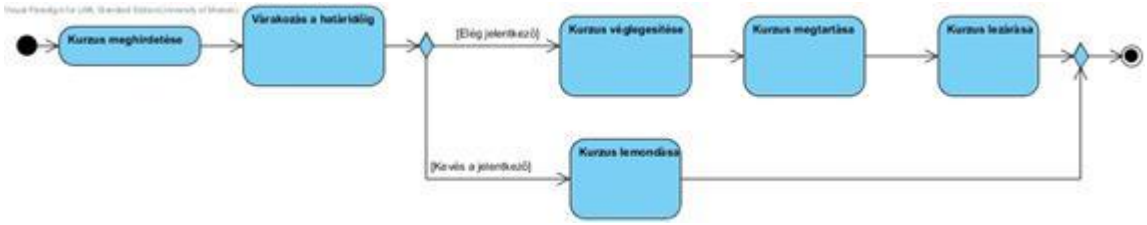
A tevékenység (aktivitás) valamilyen végrehajtandó műveletsorozat. Ez lehet néhány utasítással megoldható, de akár nagyon bonyolult, összetett tevékenység is. A tevékenységek részleteit újabb aktivitás diagrammal is meg lehet adni, ezáltal a tevékenységek egymásba ágyazhatók.

Az egymás után végrehajtandó, egymástól függő tevékenységeket nyíllal kötjük össze. Ha az A tevékenységtől nyíl vezet B-be, az azt jelenti, hogy az A tevékenység teljes befejeződése után kell elkezdeni a B végrehajtását.

A nyilakkal tevékenységek szekvenciáját lehet kijelölni. Döntési pont közbeiktatásával alternatív végrehajtási utakat hozhatunk létre. A döntési pontba legalább egy nyíl vezet a döntést megelőző tevékenység(ek)től, és legalább két nyíl az alternatív tevékenységekhez. A döntési pontból kiinduló nyilakhoz szögletes zárójelben meg kell adni azt a feltételt, amelynek teljesülése esetén az adott irányban folytatódik a végrehajtás. Az alternatív ágak összefutásánál szintén egy rombuszt helyezhetünk el.

Az eddigi jelöléseket szemlélteti az alábbi diagram, amely egy egyetemi kurzus meghirdetéséhez kapcsolódó tevékenységeket foglalja össze, a meghirdető tanszék szempontjából.

10.11. ábra - Egyetemi kurzus meghirdetésének aktivitás diagramja



4.2. Konkurens tevékenységek

Gyakran előfordul, hogy bizonyos tevékenységcsoportok egymástól függetlenül, vagy párhuzamosan végezhetők. Ezért a végrehajtási szálakat szétválaszthatjuk. A szétválasztás jele egy vastag vonal, legalább egy nyíl vezet a megelőző tevékenység(ek)től, és legalább két nyíl a párhuzamos tevékenységekhez. A szétválasztáshoz logikai kifejezés (őrszem, guard) kapcsolható, a szokásos szögletes zárójelpárral jelölve. Az alapértelmezés szerinti őrszem: [és].

A párhuzamos tevékenységsorozat végét ugyanilyen vonallal jelölhetjük (összeolvasztás). Az összeolvasztás vonala egy szinkronizációs pontot jelez: az azt követő tevékenység csak akkor kezdődhet, ha mindegyik párhuzamos tevékenység sorozat véget ért.

A párhuzamos ágak szinkronizációs vonal nélkül is összefuthatnak. Ennek az a jelentése, hogy az első befejeződő szál megszakítja a többi szálat, és a vezérlés halad tovább.

Az aktivitás digrammon megjelenő végrehajtási szálak nem azonosak a programok „szál” fogalmával, mert kétféle jelentéssel bírnak:

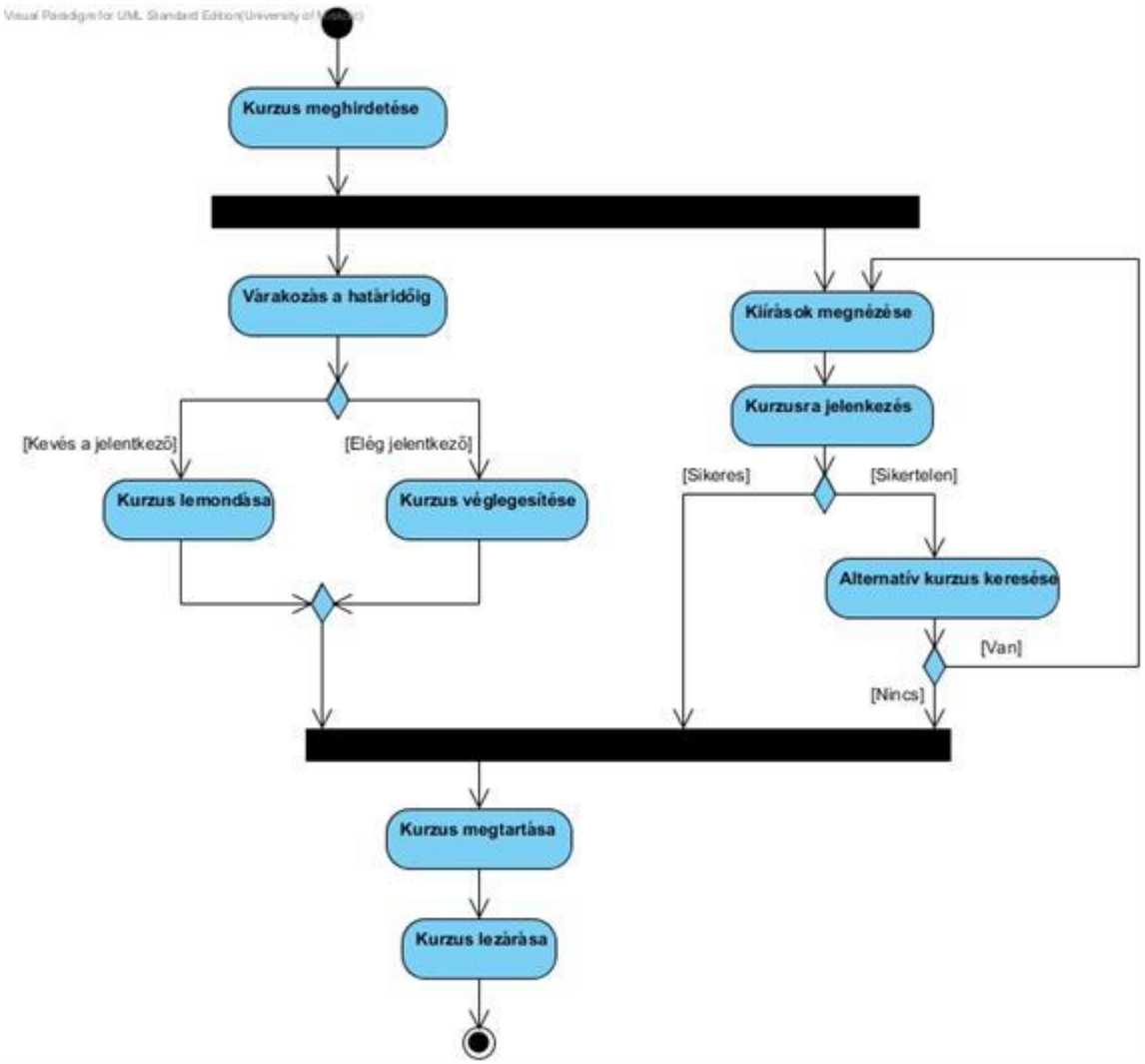
1. Az egyes szálakba foglalt tevékenységek egymástól függetlenek, végrehajtásuk sorrendje lényegtelen, vagy akár párhuzamosan is végrehajtódhatnak. (Nem lényegi konkurencia.)

2. Az egyes szálakba foglalt tevékenységeket párhuzamosan kell végrehajtani. (Lényegi konkurencia.)

Az UML nem ad eszközt a kétféle konkurencia megkülönböztetésére, tehát ezt mindig a tevékenységek jellegének vizsgálatával kell eldönteni.

Példaként visszatérve a kurzus kezelésre, a tanszék mellett a hallgatóknak is van feladatuk. Ezt fejezi ki a következő ábra.

10.12. ábra - Aktivitás diagram párhuzamos tevékenységekkel



A két szikronizációs pont között a tanszék és a hallgatók tevékenysége párhuzamosan kell fusson. A kurzus megtartására csak az után kerülhet sor, ha mind a tanszék, mind a hallgatók befejezték a kurzus választáshoz tartozó tevékenységeiket.

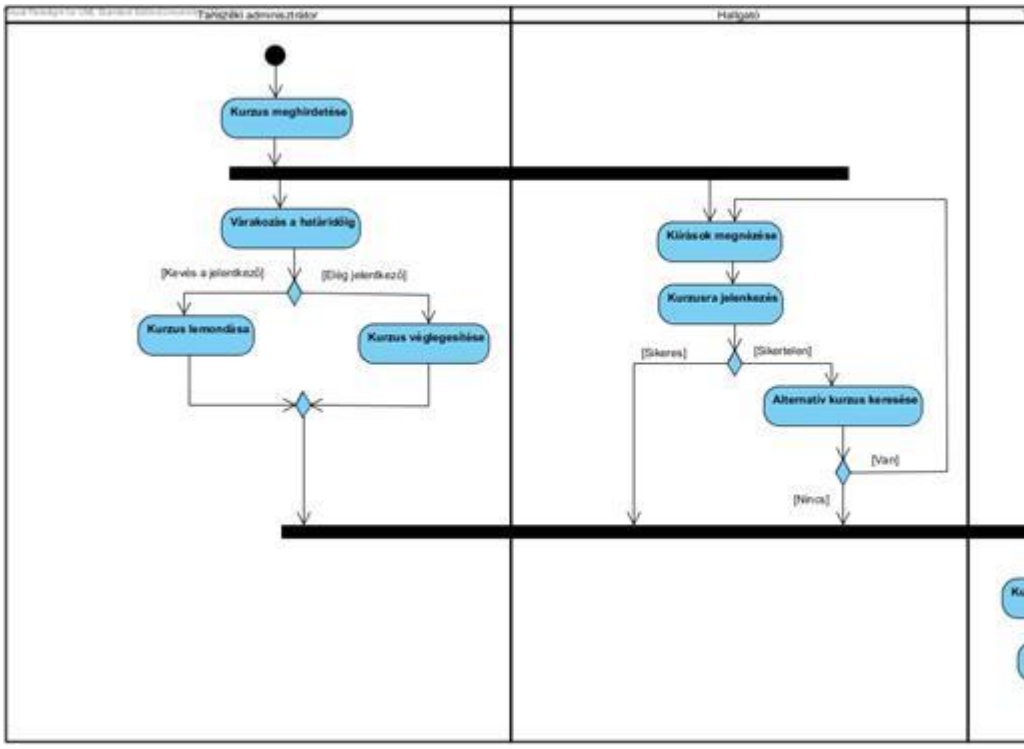
4.3. Sávos aktivitás diagram

Az aktivitás diagramban jelölhetjük az egyes tevékenységek végrehajtóit is. Ehhez a diagramot sávokra osztjuk. Minden sáv tetejére egy végrehajtó nevét írjuk. Az egyes tevékenységek attól függően kerülnek valamelyik sávba, hogy ki a felelős a végrehajtásukért.

A tevékenységek végrehajtói lehetnek aktorok, objektumok vagy akár nagyobb architekturális egységek.

A fenti példában a végrehajtók aktorok: a tanszéki adminisztrátor, a hallgató és a tárgy előadója. A felelősöket is feltüntető sávos diagram az alábbi ábrán található.

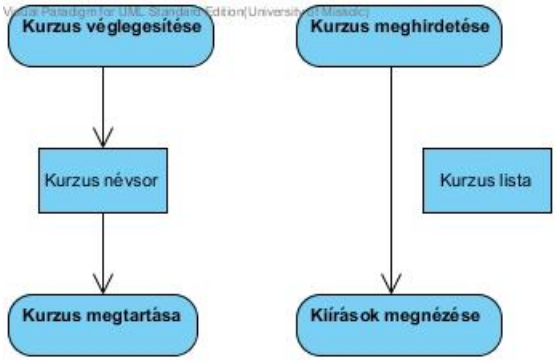
10.13. ábra - Sávos aktivitás diagram



4.4. Adatfolyam és adattárolás

Az aktivitás diagramokon jelölhetjük a tevékenységek között áramló adatelemeket is. Az adatelemet téglalap jelöli, és a tevékenységeket összekötő nyilakhoz tartozik. Két lehetséges jelölési módját mutatja az alábbi ábra.

10.14. ábra - Adatfolyam jelölése



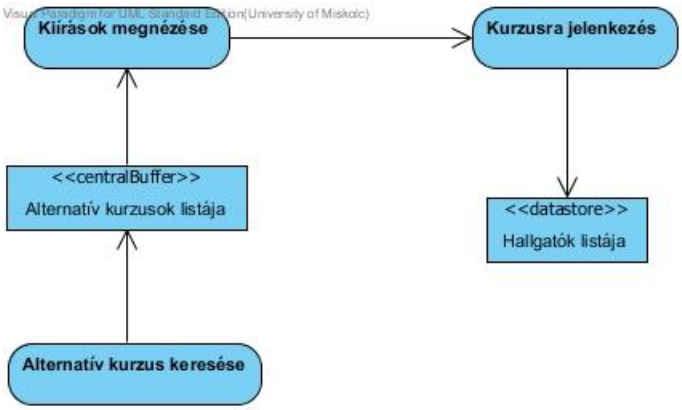
A diagramon jelölhetjük azt is, ha az adatokat tárolni kell. Kétféle tárolási módot jelölhetünk:

1. Tranziens tárolás: adatpuffer (CentralBUfferNodes). Az adatelemeket időlegesen helyezhetjük ide. Az adatelemekre való hivatkozás törli azokat a pufferből.

2. Perzisztens tárolás: adattár (DataStores). A bejövő adat tárolásra kerül, a lekérdezett adatok másolatként kerülnek visszaadásra. Ha egy már eltárolt objektumot helyezünk be újra, az előzőleg tárolt objektum felülíródik.

Az alábbi ábra a kétféle adattárolás jelölését mutatja. Az alternatív kurzusok listája egy tranziens tárolóba kerül. Ha valamelyikre jelentkezik a hallgató, a jelentkezését perzisztens módon kell eltárolni.

10.15. ábra - Adattárolás jelölése



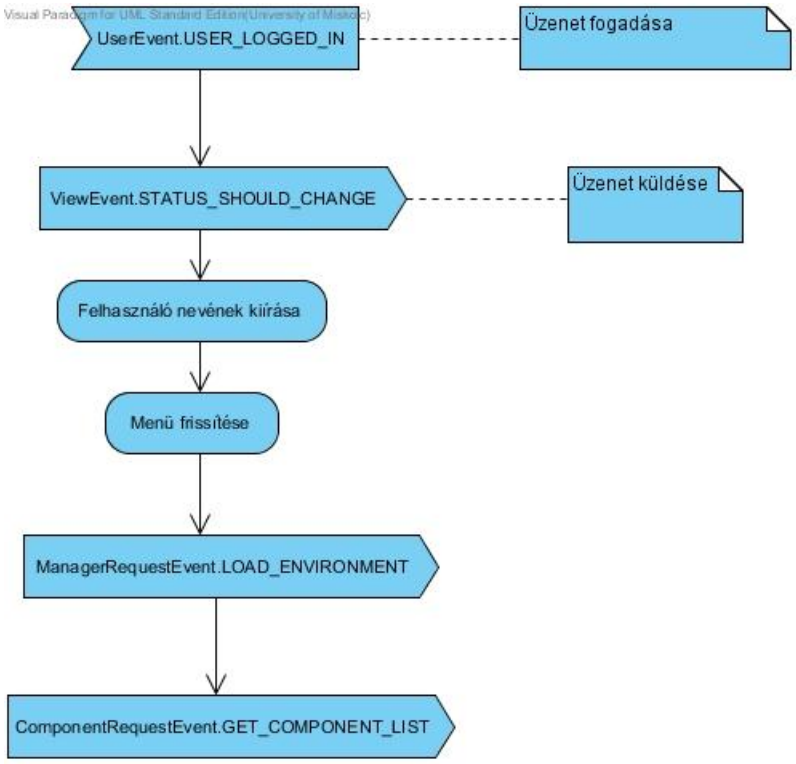
4.5. Szignál küldése és fogadása

Az aktivitás diagram tevékenységei összetett tevékenységek, feltételezésünk szerint időt vesz igénybe a végrehajtásuk. Szükséges lehet azonban, hogy a tevékenység végrehajtása során más tevékenységnek jelzést küldjünk. Az ilyen jelzéseket akcióknak, vagy szignálnak nevezzük. A tevékenységekkel ellentétben ezek pillanatnyi történések, végrehajtási idő nélkülinek tekinthetők.

Az aktivitás diagramon lehetőség van jelölni egy jelzés küldését és a jelzés fogadását. A jelzéshez objektum folyam is tartozhat, mert a szignállal adatokat is küldhetünk

A jelölésre mutat példát az esettanulmányból kiemelt diagram, amely egy felhasználói bejelentkezési folyamatot ábrázol. A bejelentkezés tényét egy szignál jelzi, és további szignálokat kell küldeni azoknak a moduloknak, amelyeknek a bejelentkezésről tudomást kell szerezniük.

10.16. ábra - Szignál küldése és fogadása

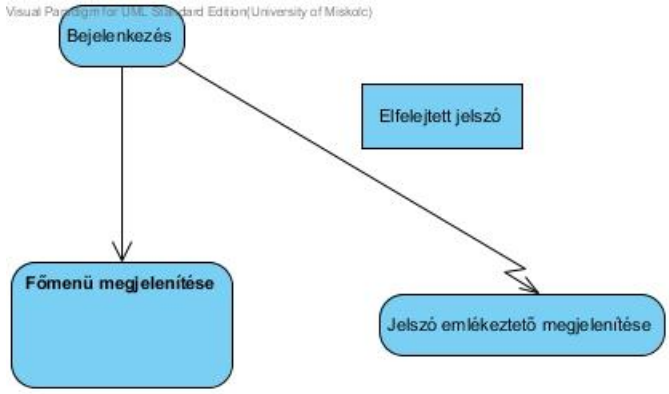


4.6. Kivételek

Gyakran előfordul, hogy a tevékenységeket bizonyos feltételek bekövetkezése esetén meg kell szakítani. Ezt jelölhetjük egy kivétellel. A kivételeket a tevékenységen kívül kell lekezelni. A kivétel valójában egy speciális

szignál. Jelölése egy szimbolikus villám. A kivételhez gyakran objektum folyam is tartozik, amely a kivétel keletkezésének okát és körülményeit írja le. Egyszerű példa: felhasználó bejelentkezése. Ha a felhasználó a jelszó megadása helyett az „Elfelejtettem a jelszavamat” gombot nyomja meg, ez kivételt vált ki, aminek lekezelése ebben az esetben egy jelszó emlékeztető szöveg megjelenítése. Ha nem keletkezik kivétel (azaz a bejelentkezés sikeres), a főmenü jelenik meg.

10.17. ábra - Kivétel jelölése



11. fejezet - Az analízis modell

A fejlesztés első fázisában összegyűjtöttük a rendszerrel szemben támasztott funkcionális és nem funkcionális követelményeket. A funkcionális követelmények rendszerezésére használhatjuk a használati eset modellt. A használati eset modell a rendszer felhasználói (funkcionális) nézetének is tekinthető.

Egy számítógépes rendszer elkészítése során a következő lépés a feltárt követelmények elemzése alapján az analízis modell elkészítése, amely a rendszer statikus és dinamikus nézetét is rögzíti. Az analízis fázisban csak a szakterületi követelményekkel foglalkozunk, és a célterület fogalmaival dolgozunk. Nem foglalkozunk az egyes elemek megvalósítási módjával: Ez majd a következő munkafázis, a tervezés feladata lesz.

A továbbiak könnyebb megértése kedvéért újra felidézzük a használati eset modell példáját, egy kisbolti rendszer leírását. Ez az (egyszerűsített) szakterületi modell segít abban, hogy az analízis modell megalkotásához szükséges lépéseket szemléltessük.

A modellezendő rendszer (pénztári rendszer) leírása:

„A rendszer egy pár alkalmazottat foglalkoztató, néhány pénztárral működő kis önkiszolgáló élelmiszerbolt pénztári rendszere. A vevő egy kosárban összegyűjti a megvásárolni kívánt árukat. A pénztáros az árukat a vonalkódok beolvasásával azonosítja, és így készíti el a blokkot, ami alapján a vevő fizet. A megvásárolt áruk mennyiségével a rendszer csökkenti a raktárkészletet. Bizonyos áruk csak betétdíjas göngyöleggel együtt adhatók el, és a göngyöleg készletét is nyilván kell tartani. (Például a sör…).

A boltvezető bármikor helyettesíthet egy pénztárost, ha annak valami miatt félbe kell szakítania a munkáját. A boltvezetőnek joga van egy törzsvásárló esetén hitelbe is odaadni az árut, ezt a pénztáros nem teheti meg.

Ha egy áru vonalkódja sérült, ezért nem olvasható be, a pénztáros az árut a megnevezése alapján keresi vissza az adatbázisból.”

1. Elemzési osztálydiagram készítése

A rendszer modelljének először elkészítendő nézete a statikus (strukturális) nézet. A rendszer statikus nézete a rendszert alkotó elemek és azok kapcsolatainak felderítése során alakítható ki.

A statikus modell elkészítése szükséges ahhoz, hogy egy következő lépésben a rendszer dinamikus nézőpontú modelljét megalkossuk. A statikus modellben szereplő elemek együttműködése határozza meg a rendszer dinamikus viselkedését. A dinamikus modell előállítása során a statikus nézet is változik, kiegészül.

A statikus modell elkészítésének ismertetése során erősen támaszkodunk az OMT módszertan ajánlásaira.

A statikus nézet rögzítésére alkalmas UML eszközök:

1. osztály diagram
2. alrendszer / csomag diagram
3. összetett struktúra diagram

A modell kezdeti statikus nézetét az analízis fázisban készítjük el. Megjelenési formája egy vagy több elemzési osztálydiagram. Az összetartozó osztályokat alrendszerekbe, csomagokba rendezhetjük.

Az OMT módszertan az analízis fázis első lépéseként az általa „objektum modell”-nek nevezett statikus modell elkészítését írja elő. Kiindulásként feltételezi, hogy rendelkezésre áll az elkészítendő rendszer leírása. Alapja a funkcionális követelmények leírásának nyelvtani szempontú elemzése.

Az OMT kidolgozása idején még nem voltak ismertek azok az eszközök, amelyeket ma a szakterületi modell dokumentálására használunk, ezért a módszertan a rendszer általános, szöveges leírását tekintette az analízis kiindulópontjának. A jelenleg szokásos fejlesztési munkafolyamat is a követelmények összegyűjtésével és a használati eset modell elkészítésével kezdődik. A funkciólista és a használati esetek dokumentációi alkalmasak az OMT által ajánlott, nyelvtani alapú elemzések elvégzésére.

1.1. Osztályok azonosítása

Az analízis modell első lépése az osztályok azonosítása. A rendszer leírásában általában főnevek utalnak lehetséges osztályokra, abból a feltételezésből kiindulva, hogy a főnevek a rendszer részét képező fogalmak, tárgyak, események, szerepek, személyek, helyek, egyéb strukturális elemek megnevezései. Természetesen nem minden főnév jelent osztályként modellezendő absztrakciót, csak amely elég hangsúlyos szerepet játszik, és nem egyszerűen adat, hanem viselkedéssel is felruházható.

1.2. Megfelelő osztályok kiválasztása

Az OMT ajánlásaiban minden elemi lépést az adott lépés eredményeinek az ellenőrzése követ. Az előző pontban leírtak végrehajtásával lehetséges osztályok listáját állítottuk elő. A lista minden egyes elemét újra megvizsgálva, jelentését ismételten végiggondolva finomíthatjuk a listát. Az elsődleges nyelvtani elemzés során feltárt osztályok közül törölnünk kell a nem megfelelőket.

Törlendők:

1. Redundáns osztályok

A természetes nyelv gyakran használ szinonímákat, így előfordulhat, hogy az osztály listán ugyanaz az elem több néven is szerepel. Ezek közül a legkifejezőbbet kell meghagyni, a többit törölni.

1. Felesleges osztályok

A feladat leírásában gyakran olyan fogalmak is szerepelnek, amelyek az egyes funkciók leírását magyarázzák, de nem tartoznak a modellezendő rendszerhez.

1. Pontatlan osztályok

A természetes nyelv nem mindig fogalmaz pontosan. Ha például egy osztály-jelöltnek nem tudunk kifejező megnevezést adni, még nem értettük meg az adott modell elem pontos jelentését. Az ilyen pontatlan osztályokat elemezve azok vagy valamelyik elhagyható kategóriába sorolandók át, vagy pontosítani kell a jelentésüket.

1. Példányok

Gyakran előfordulhatnak a szövegben olyan főnevek, amelyek csak valamelyik osztály példányát / példányait jelentik.

1. Attribútumok

A rendszer adatszerű elemei, amelyek nem igénylik, hogy osztályként modellezzük azokat. Ha a rendszer lényegi elemeit képezik, majd valamelyik osztály részeként kerülnek be a modellbe.

Lehetnek olyan adatok, amelyek kezelése, a rajtuk végzendő műveletek nem triviálisak, ezért gondolhatunk az osztályként való modellezésére. (Tipikusan ilyen például a dátum, ami gyakori attribútum, és a dátum műveletek nem egyszerűen implementálhatók.) Elemzési osztályként azonban nem érdemes felvenni a listára, ha csak valamilyen strukturális elem attribútumaként szerepel. Például egy számlán több dátumot is meg kell adni (kiállítás, teljesítés, esedékesség dátuma). A számla lehet osztály, a dátumok csak adatok. Az implementációs szintű modellben az ilyen adatok kezelésére szolgáló, adattípust jelentő osztályok majd újra megjelenhetnek, de az elemzési modellt nem érdemes bonyolítani ezekkel az osztályokkal.

1. Műveletek

A főnevek jelenthetnek műveleteket is. A műveletek operációként, vagy objektumok együttműködéseként majd a dinamikus modellbe kerülnek be.

1. Szerepkörök

Az osztályok kapcsolatainak jellemzőjeként kerülhetnek majd be a modellbe.

1. Implementációs elemek

Az analízis modellben csak a szakterületi követelményeket vizsgáljuk, azok megvalósítási módjait nem. Az implementációs elemeket jelentő osztályok csak bonyolítják a modellt, de nem teszik érthetőbbé, pontosabbá, ezért elhagyandók.

1.3. Osztályok leírása

Valójában már az előző pontban leírt ellenőrző tevékenységgel párhuzamosan, azt kiegészítve el kell készíteni minden osztályhoz egy rövid, néhány bekezdésnyi definíciót. A rövid leírások elkészítése a későbbiekben felhasználandó információkat rögzít, egyben segíti az egyes osztályok jelentésének jobb megértését is. A definíciók megfogalmazása közben is felfedezhetünk még pontatlan vagy felesleges osztályokat.

1.4. Példa: kezdeti osztálydiagram

A fenti lépések végrehajtása egy nevekkel azonosított osztályhalmazt eredményez, amelyeket a definíciójuk egészít ki. A fejezet elején leírt példarendszer esetén a kezdeti osztálylista lehet az alábbi:

rendszer, alkalmazott, pénztár, pénztáros, élelmiszerbolt, vevő, kosár, áru, vonalkód, blokk, raktárkészlet, göngyöleg, sör, boltvezető, hitel, megnevezés (árué), adatbázis

A fentiek közül a rendszer, alkalmazott, pénztár, élelmiszerbolt, vevő, kosár főnevek a rendszeren kívüli, csak a leírás érthetőségét szolgáló fogalmak.

A vonalkód, raktárkészlet, megnevezés főnevek csak attribútumok.

A sör csak az áru egy lehetséges példánya.

A hitel egy művelet, így az sem osztály.

Az adatbázis implementációs elem.

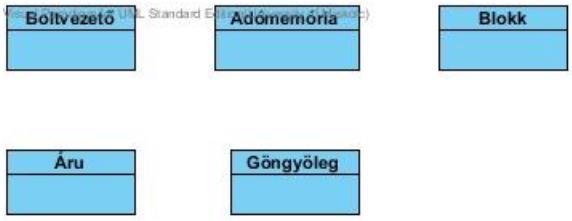
A pénztáros, áru, blokk, göngyöleg, boltvezető főnevek a rendszer részét képező, fontos absztrakcióknak tűnnek, tehát potenciális osztályok.

Bár a leírásban nem szerepel, de a használati eset modell elkészítése során felfedeztük, hogy a rendszernek része még egy „Adómemória” is. Ez is szereplője a rendszernek, és osztályként modellezhető.

Érdekes még külön elgondolkodni a használati eset modellben aktorként, tehát a rendszeren kívüli szereplőkként megjelölt pénztáros, boltvezető, adómemória elemeken, hiszen a statikus modellben csak a rendszer részeit képező elemeket kell számba venni. Az aktorok csak akkor jelennek meg a modellben, ha attribútumaik és műveleteik annak részét képezik. A példa rendszer leírása alapján a pénztáros valóban csak felhasználóként szerepel, mert nincs a rendszerben nyilvántartásba veendő adata, így kimarad a kezdeti osztálydiagramból. A boltvezető és az adómemória azonban konkrét operációkkal szerepel a rendszerben (hitel adás, illetve bevétel és ÁFA gyűjtése), ezért elemzési osztályként célszerű a modellezésük.

A fentiek alapján a kezdeti osztálydiagram:

11.1. ábra - Kezdeti osztálydiagram



1.5. Asszociációk azonosítása.

A következő feladat az eddig feltárt osztályok közötti kapcsolatok azonosítása. A feladat leírásában igék vagy igei kifejezések utalhatnak rá. Például:

1. fizikai elhelyezkedés (része, alkotja ...)

2. tárgyas igékkel kapcsolatos cselekvések (átveszi, alkalmazza, kezeli ...)

3. kommunikáció (üzeni, továbbadja, közli ...)

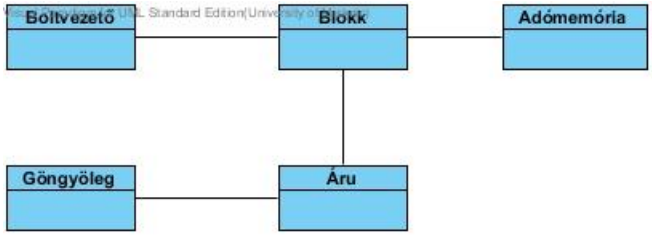
4. birtoklás (van neki, hozzá tartozik ...)

5. előírt feltételeknek való megfelelés (tag, alkalmazásban áll ...)

Ebben a fázisban elegendő egyszerű vonallal összekötni a kapcsolatban álló osztályokat.

A kisboltos példa leírásában az alábbi kapcsolatokat fedezhetjük fel:

11.2. ábra - Osztályok kapcsolatokkal



1.6. Megfelelő asszociációk kiválasztása.

Következő lépésként a kapcsolatok ellenőrzése következik. Az ellenőrzés során az alábbiakra célszerű figyelni:

1. Lényegtelen asszociációk

Bár létező, de a rendszer működése szempontjából irreleváns kapcsolatok

1. Implementációs asszociációk

A modell egyszerűsítése érdekében elhagyandók.

1. Származtatott asszociációk

Ha az A osztály kapcsolatban van B-vel, az pedig C-vel, akkor értelemszerűen A és C is kapcsoltban vannak. Ha A és C más ok miatt nincsenek kapcsolatban egymással, az ilyen származtatott asszociációk feltüntetése csak bonyolultabbá és zavarosabbá teszi a modellt, tehát a kapcsolatuk törlendő.

1. Hiányzó asszociációk

A kapcsolatok ismételt áttekintése során még hiányzó asszociációkra derülhet fény.

A kisboltos példa esetén egy hiányzó kapcsolatot tudunk felfedezni az ellenőrzés során. Mivel a bolt napi zárásakor szükséges elszámolásnak része a napi bevétel ellenőrzése, ami az adómemória kiolvasását jelenti, és ez a boltvezető feladata, a boltvezető az adómemóriával is kapcsolatban kell álljon.

1.7. Ternáris (illetve többszörös) asszociációk átalakítása

A többszörös asszociációk arra utalnak, hogy nem pontosan tisztázott az adott osztályok közötti kapcsolat. A bináris asszociációkká alakítás új osztály bevezetésével lehetséges. A kisboltos példában nincs többszörös asszociáció.

1.8. Asszociációk szemantikájának ellenőrzése

Ebben a munkafázisban az egyes kapcsolatokhoz hozzá kell rendelni minden olyan tulajdonságot, amely a feladat leírásából kiolvasható.

1. Megfelelő elnevezés

Minden kapcsolatot el kell látni elnevezéssel. Ha nem tudunk kifejező megnevezést találni az adott kapcsolathoz, az az asszociáció szerepének átgondolását igényli.

1. Szerepkör nevek megadása

2. Számosság meghatározása

3. Asszociációk minősítőjének kiválasztása

A kapcsolatok többes oldalán ki kell választani azt a szempontot, ami alapján a számosság csökkenthető

1. Tartalmazás kapcsolatok feltárása

2. Hiányzó asszociációk feltárása

Ebben a munkafázisban is felfedezhetünk hiányzó asszociációkat.

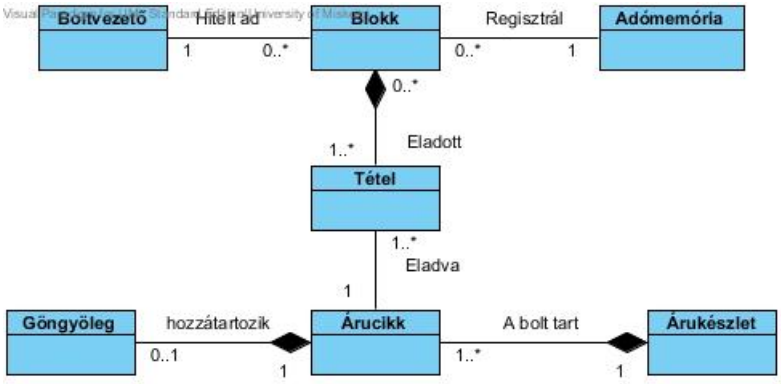
Ha a kapcsolatokat nem tudjuk egyértelműen azonosítani, az jelentheti azt, hogy az osztályok nem elég pontosak. Ezért a kapcsolatok elemzése és pontosítása az osztályokat módosíthatja, illetve új osztályok bevezetését is eredményezheti.

A példa rendszer esetén a blokk és az áru kapcsolata tűnik problémásnak. Mivel a blokkon szerepelnek az eladott áru (bizonyos) adatai, tartalmazás jellegű kapcsolatra gondolhatunk. Az áru azonban nem szűnik meg létezni attól, hogy a blokkal a vásárló kimegy a boltból, hiszen marad még (általában) belőle a polcon / raktárban. A problémát az okozza, hogy érezhetően az „Áru” osztály több fogalmat mos össze: van eladott áru, és eladható áru. Ráadásul a fenti gondolatmenetben feltűnt egy újabb elem: a raktár.

A problémát úgy oldhatjuk meg, hogy felvesszük az „Árukészlet” osztályt, az „Áru” osztály fogalmát pedig pontosítjuk. Vezessük be az „Árucikk” fogalmát. A raktárban található áruk összessége az „Árukészlet”, amely „Árucikk”-ekből áll (ez tartalmazás, sőt kompozíciós kapcsolat), a blokkon megjelenő tételek pedig az adott vevőnek éppen eladott áruk, amiket nevezzünk „Tétel”-nek. Az „Árucikk” és a „Tétel” között természetesen kapcsolat van. Egy „Tétel” eladása például csökkenti az „Árucikk” készletét.

A fenti meggondolásokkal is kiegészített osztály diagram tehát az alábbi:

11.3. ábra - Osztálydiagram pontosított kapcsolatokkal



1.9. Attribútumok azonosítása

A következő feladat a már eddig feltárt osztályok attribútumainak azonosítása. Attribútumokra utalhatnak a leírás elemzése során a melléknevek, birtokos szerkezetek. Attribútumok az osztály objektumainak állapotát meghatározó adatok. Az attribútumok nevének egy osztályon belül egyedinek kell lennie.

Az elemzési osztálydiagramban az attribútumok az osztály lényeges adatait reprezentálják. Általában a típusát is meg tudjuk mondani, de ez akár hiányozhat is.

1.10. Megfelelő attribútumok kiválasztása

Az attribútumok körének meghatározása után azok ellenőrzése következik.

1. Származtatott attribútum elhagyása vagy megjelölése

Az attribútumuk egymástól független adatok. A más attribútumok értékéből kiszámítható értékeket általában elhagyjuk, de ha az osztály megértése szempontjából fontos, megtarthatjuk, de ilyenkor a neve előtt a / (per) jellel megjelöljük. A tervezési szintű modellben a származtatott adatokat azok kiszámítására alkalmas operációkkal helyettesítjük.

1. Ha egy adat önálló létezéssel rendelkezik, az objektum

Az ilyen attribútumot kiemeljük az osztályból, és külön osztályként modellezzük (amely természetesen kapcsolatban lesz az eredeti osztállyal).

1. Az objektum azonosító implementációs elem

Minden objektum önmagában egyedi, tehát egy olyan attribútum, ami csak azért kell, hogy a többitől megkülönböztesse, felesleges. Nem tévesztendő ez az eset össze azzal, amikor valamely adat a külvilág számára teszi azonosíthatóvá az objektumot (például az autó rendszáma). Az ilyen attribútum szerepelhet minősítőként is, lásd a következő pontot

1. A minősítő, ha az értéke azonosítás jellegű, elhagyható és asszociációhoz rendelhető

Ha az attribútum szerepe az, hogy az asszociáció többes oldalán a számosságot csökkentse, akkor azt modellezhetjük a kapcsolat minősítőjeként. (például a bolti példában a vonalkód lehet ilyen adat.)

1. Az attribútum osztályhoz vagy asszociációhoz kapcsolódik?

Találhatunk olyan attribútumokat, amelyek egy kapcsolat két végén szereplő osztályok bármelyikéhez kapcsolhatók, de egyikhez sem illenek igazán. Ilyenkor célszerű megvizsgálni, hogy nem magához a kapcsolathoz tartoznak-e. Ha igen, akkor a kapcsolathoz csatolt új osztályt kell létrehozni, és a kérdéses attribútumot ebbe az asszociációs osztályba telepíteni. Erre látunk példát az osztály diagram ismertetése során a Cég – Személy osztályok kapcsolatánál.

1. Implementációs részleteket jelentő attribútum (belső érték) elhagyandó

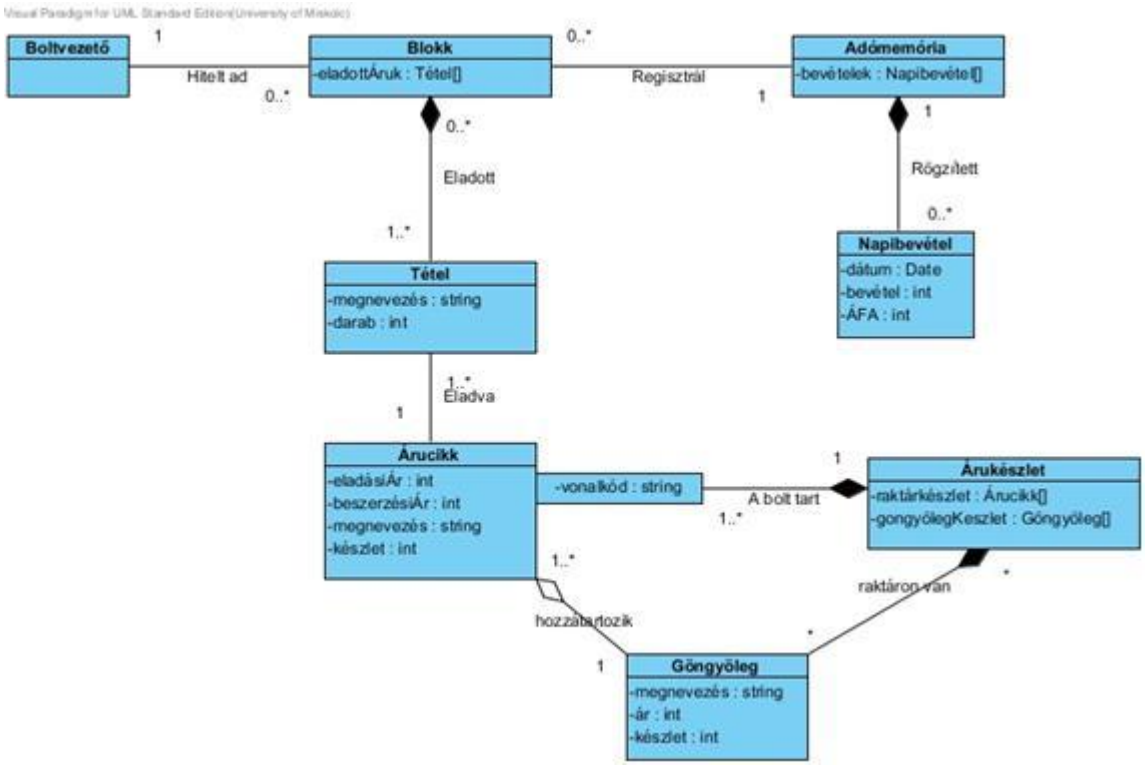
2. Az attribútumok kohéziójának vizsgálata

Mivel egy osztály valamennyi attribútuma az osztály által modellezett fogalom tulajdonsága, közöttük valamilyen logikai kapcsolatnak kell lennie. Ha azt tapasztaljuk, hogy egy vagy több attribútum a többihez csak lazán kapcsolódik, az annak a jele lehet, hogy az adott osztály több fogalmat próbál egyszerre modellezni. Ilyenkor célszerű az osztályt több osztályra bontani, ezzel is pontosítva a modellt.

Az attribútumok vizsgálata, az egyes adatok megfelelő osztályhoz rendelése tehát a fentiek szerint bővíti az osztályok és kapcsolataik specifikációját, de az elemzés során az osztálydiagram is változhat, új osztályokkal és új kapcsolatokkal bővülhet, esetleg pontatlannak bizonyult osztályok eltűnnek.

A kisboltos példa osztálydiagramja kiegészítve az attribútumokkal (ez is csak példa, nem teljes, egy valódi fejlesztés során még további attribútumokkal kellene pontosítani):

11.4. ábra - Osztálydiagram attribútumokkal



A fenti ábra elkészítése az alábbi meggondolásokat tükrözi:

1. Az Árucikk adatai közül a vonalkód az Árukészlet osztállyal való kapcsolatában minősítőként szerepel, mert ez alapján lehet egy ilyen típusú objektumot kiválasztani.

2. Az adómemóriába naponta le kell menteni a napi bevételt és annak az ÁFA tartalmát (a valós szabályok ettől valamivel bonyolultabbak, de most a példa kedvéért egyszerűsítünk). Ezek az adatok összetett szerkezetűek, és műveletek is értelmezhetők rajtuk, ezért célszerűen egy új osztályként jelennek meg a diagramon.

3. A boltvezető osztálynak nincs feltüntetve attribútuma. Ez szándékos: a boltvezető egy egyedi objektum a rendszerben, és semmilyen adata nem szükséges a rendszer működéséhez. Külön azonosítóra sincs szüksége, mert az objektum egyedisége automatikusan azonosítja.

4. Gondolhatunk rá, hogy a Blokknak természetesen van végösszege, de ez származtatott (számított) adat, tehát nem kell feltüntetni attribútumként.

További változást figyelhetünk meg a Göngyöleg osztály diagramon belüli helyzetében. Ez egy példa arra, hogy a modell pontosítása során korábbi döntéseinket is felül kell bírálni adott esetben. Az osztálydiagram módosítására a következő gondolatmenet vezethetett:

1. A Göngyöleg osztály szerepét eddig nem jól értelmeztük, amit az attribútumainak elemzése során fedezhetünk fel: ennek is van készlete, amit egy göngyöleges árucikk eladása csökkent, az üvegvisszavétel pedig növel.

2. Eszerint a Göngyöleg az Árukészletnek ugyanolyan része, mint az Árucikk, és ezt a kapcsolatot is fel kell tüntetni.

3. Az osztály pontosabb jelentésének átgondolása során még arra is rá kell jönnünk, hogy az így „önállósított” Göngyöleg fogalom és az árucikk kapcsolatának multiplicitása sem volt helyes, mert több árucikkhez tartozhat ugyanaz a göngyöleg típus (gondoljunk az ugyanolyan üvegbe palackozott különböző sörmárkákra …).

4. Az elemzés során egy új funkció is felmerült: az üvegvisszaváltás. Ennek következtében a funkcionális modellt is bővíteni kell. Ez a kis példa tehát arra is rámutat, hogy egy adott fejlesztési fázis során megalkotott modellt gyakran a későbbi fázisokban megszerzett új ismeretek felhasználásával módosítanunk kell.

A diagram kiegészítése mellett természetesen az osztályok leírását bővítjük az attribútumok definíciójával, és az osztályok leírását is módosítjuk szükség szerint (lásd Göngyöleg), tehát az osztályok dokumentációja is pontosabbá válik.

1.11. Általánosítás

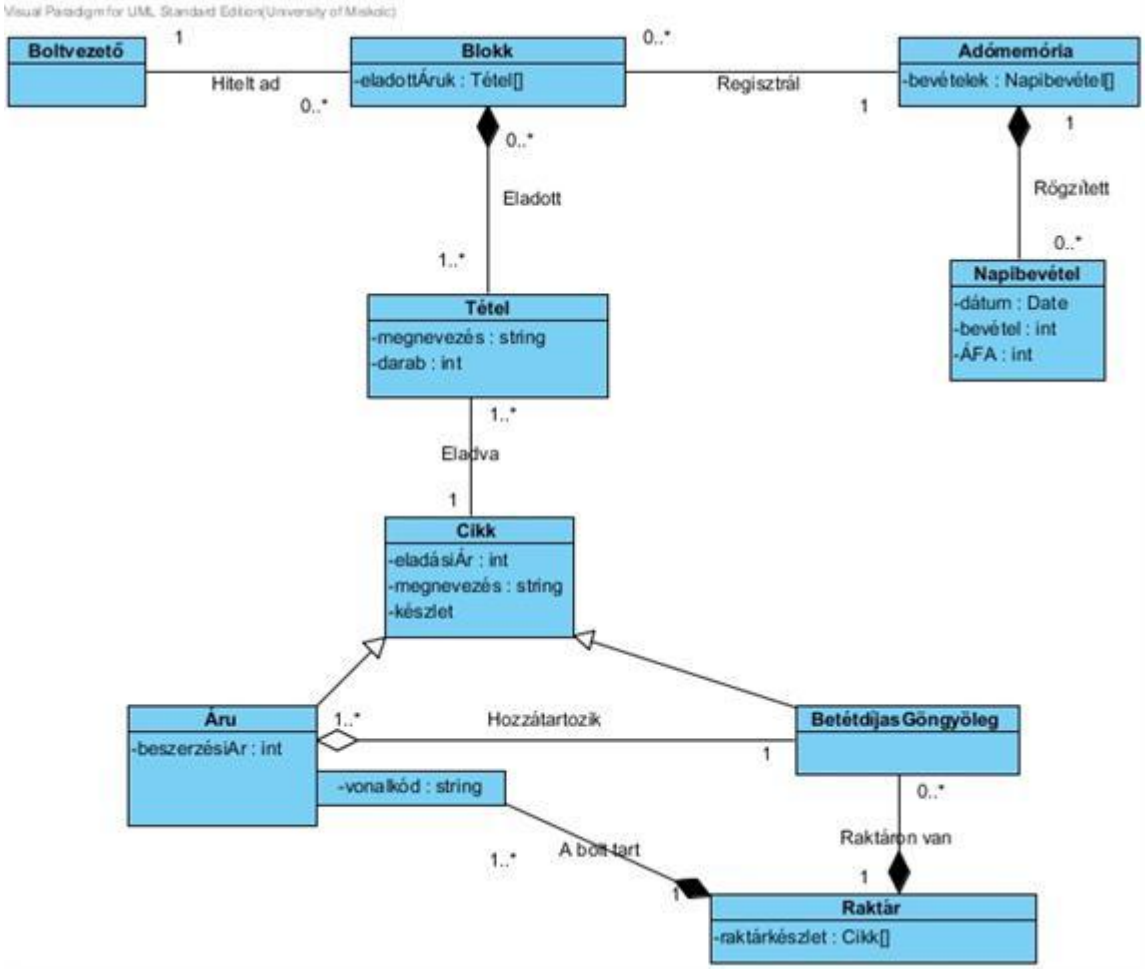
Az általánosítás kapcsolat megjelenhet a szövegben például azonos tárgyakhoz tartozó jelzős kapcsolatok formájában.

Az attribútumok vizsgálata során feltűnhet, ha több osztályban fordulnak elő azonos attribútumok. Ezeknek az adatoknak a kiemelése egy bázisosztályba, és a különbségek leszármazott osztályokba telepítése egyszerűsítheti a modellt.

A bázisosztály keresése során ügyelni kell arra, hogy az azonos elnevezés még nem feltétlenül jelent azonos adatokat. Az azonosság megállapításához az attribútum pontos jelentését kell elemezni.

A kisbolti rendszerben az Árucikk és a Göngyöleg osztályok esetén fedezhetünk fel azonos attribútumokat (megnevezés, ár, készlet). Ha ezeket egy közös „Cikk” osztályba delegáljuk, ezzel az „Árukészlet” osztályt is egyszerűsíthetjük. Ezekkel a változtatásokkal az osztálydiagram:

11.5. ábra - Osztálydiagram általánosítással



1.12. Elérési utak tesztelése

Az attribútumokkal kiegészített osztálydiagram ellenőrzéséhez az alábbi szempontokat célszerű figyelembe venni:

1. a működéshez szükséges asszociációk rendelkezésre állnak-e,

2. a működéshez szükséges attribútumok rendelkezésre állnak-e,

3. többszörös multiplicitás esetén a kiválasztáshoz van-e minősítő vagy név.

Az ellenőrzés eredményeként megállapíthatjuk, hogy a fenti a fenti diagramot az alábbiakkal kellene kiegészíteni:

1. A BetétdíjasGöngyöleg és a Raktár osztály közötti többes kapcsolathoz nincs minősítő, tehát be kell vezetni erre a célra egy kódot a gyöngyölegek azonosítására. Ugyanezt a minősítőt kellene használni a Cikk-el való kapcsolatában is.

2. A Cikk és a Tétel közötti kapcsolat többes végére bevezethetnénk egy „tételsorszám” minősítőt, de a jelenlegi ismereteink szerint ennek nincs funkciója, tehát elhagyhatjuk.

3. Az ábrán szerepel, hogy az adómemóriába gyűjteni kell az eladások ÁFÁ-ját. Nincs viszont olyan adat, amiből ezt kiszámolhatnánk. Ezért tisztázni kell, hogy a Cikk osztályban szereplő eladási ár a bruttó ár, és az Áru osztályt ki kell egészíteni az „ÁFA kategória” attribútummal.

A fenti változtatások diagramon történő átvezetését gyakorlásképpen az olvasóra bízzuk.

1.13. Modulok meghatározása

A követelmények elemzése során már elkészíthettük a használati eset leltárt, amely a rendszer funkcionális tagolását adja meg.

Az elemzési osztálydiagram elkészülte ismét lehetőséget ad arra, hogy alrendszerekre bontsuk a feladatot, megvizsgálva az architekturális részek kapcsolatait. Azokat az osztályokat sorolhatjuk egy modulba, amelyek között szoros kapcsolatok vannak. A modulok kialakítása során arra kell törekedni, hogy a minimalizáljuk a közöttük levő, a modul határokon átnyúló kapcsolatokat.

A modulok ábrázolására a használati eset leltárnál már említett csomag diagramot használhatjuk.

2. Az analízis modell dinamikus nézete

A dinamikus nézet megalkotása során megvizsgáljuk és dokumentáljuk a rendszer – és objektumai – időbeli viselkedését. A rendszer viselkedésének vizsgálatához újra elemezzük az elvárt funkciókat. Ennek segítségével felfedezhetjük az objektumok egymás közti együttműködési módjait, és a rendszer által végrehajtandó számítási lépéseket. A dinamikus nézet felépítése során felhasználjuk az osztálydiagram elemeit, amelyeket gyakran új elemekkel és a meglévő elemek további tulajdonságaival egészítünk ki.

A továbbiakban olyan lépéseket sorolunk fel, amelyeket a munka során célszerű követni. Ezeket részben már az OMT módszertan is tartalmazta.

2.1. Forgatókönyvek készítése

A használati eseteket végig véve el kell készíteni az adott funkciók forgatókönyveit. Egyszerű esetekben ezeket elég szövegesen rögzíteni, egyébként jól használhatjuk az UML a szekvencia vagy a kommunikációs diagramját.

Egy funkció menetének az elemzése során először keressük meg a tipikus végrehajtási módot vagy módokat. Következő lépésként a ritkábban előforduló (speciális) és a kivételes eseteket, és a hibát eredményező lehetséges lefutásokat kell dokumentálni.

A fejezet elején leírt kisboltos példa esetén a blokkolás forgatókönyvének tipikus lefutását szemlélteti a 10. fejezet 3. ábráján látható szekvencia diagram. A blokkolás egy speciális esete a használati eset modellben is szerepel: ha a vonalkód nem olvasható, az árut megnevezés szerint kell visszakeresni. További – és eddig nem felfedezett – speciális eset az, ha a blokkolás során a pénztáros hibát követ el (például rossz darabszámot üt be), és ezért a blokk valamelyik tételét törölni kell. Végül kivételes esetként kell kezelni azt, amikor az egész blokkot sztornózni kell: például a blokk lezárása után derül ki, hogy a vásárlónak nincs elég pénze, vagy utólag reklamál, mert a pénztáros tévedett.

Először mindig az aktoroktól induló eseményekkel kapcsolatos forgatókönyveket nézzük végig. Ezen események feldolgozása során fedezhetjük fel a rendszeren belüli, objektumok közötti üzenetváltásokat.

Az egyes üzenetek hatására a fogadó objektum által elvégzendő tevékenység aktivitás diagrammal írható le.

2.2. A felhasználói felület elsődleges terve

A felhasználói felület tervének első verziója ebben a fázisban már általában elkészíthető. A felhasználói felület vezérlő elemei felhasználói interakciókat jelképeznek, ezért segítenek a forgatókönyvek megtervezésében. A felületen megjelenítendő információk egy részét pedig a rendszernek kell előállítania, ami a szükséges belső működésre ad információt.

A felhasználói felület terve a leendő felhasználóval való kommunikációt is elősegíti, lehetővé téve a modell pontosítását és az esetleges félreértések viszonylag korai felfedezését.

2.3. Objektumok állapotainak vizsgálata

Egy adott objektum üzenetváltásai az objektum állapotának a megváltozását okozzák. A rendszer minden egyes (de legalább is a legfontosabb) objektumára megrajzolható ezért az állapotgép diagram. Az diagramban az állapot átmeneteket kiváltó eseményeknek valamelyik szekvencia diagramon szerepelniük kell üzenetként, ez lehetővé teszi a modell teljességének, ellentmondás mentességének az ellenőrzését is.

2.4. Input és output értékek vizsgálata

Az dinamikus nézet megalkotása során eddig elsősorban interakciókkal, eseményekkel, üzenetekkel, azok sorrendjével foglalkoztunk, és elhanyagoltuk az ezeket kísérő belső folyamatokat. A rendszert tehát elsősorban vezérlési szempontból vizsgáltuk.

Ebben a munkafázisban megvizsgáljuk, hogy milyen bemeneti adatokat kell fogadnia a rendszernek, és azokból milyen kimeneti értékeket kell előállítania. A bemeneti adatok a külső interakciók paramétereiként tekinthetők. A kimeneti értékek lehetnek:

1. a felhasználói felületen megjelenő információk,

2. perzisztens tárolásra kerülő adatok,

3. más, együttműködő rendszerek számára továbbított adatok.

Az be- és kimenő adatok vizsgálata akár új funkciók, sőt új osztályok megjelenését is eredményezheti, módosítva ezzel a használati eset modellt és/vagy a statikus nézetet.

Ha a kisbolti rendszer egyik alapvető kimenő adatát, a blokkot specifikáljuk, akkor felmerülhet, célszerű lenne a blokkon a pénztárost is feltüntetni. Ez az eddigi modellekben az alábbi változtatásokat jelenti:

1. A pénztáros adatait a rendszernek tárolnia kell, tehát nem csak aktorként, hanem a rendszer egy osztályaként is modellezni kell.

2. Egy új használati eset keletkezett azáltal, hogy a pénztárosnak az azonosítása érdekében be kell jelentkeznie a rendszerbe.

3. További új használati esetet jelent, hogy a pénztárosok adatait a rendszerben rögzíteni kell, és azokat karban kell tartani.

2.5. A számítási folyamatok specifikálása

Ha megvizsgáljuk a rendszer által fogadott és előállítandó adatokat, akkor a rendszert magát úgy is kezelhetjük, mint amely egy transzformációt valósít meg ezek között. Ennek a transzformációnak a módját kell tehát dokumentálni. Ez egyfajta funkcionális szemléletmód, amely nem arra koncentrál, hogy az egyes események milyen sorrendben következnek be, hanem hogy mivel mit kell csinálnia a rendszernek, a vezérlési folyamatoktól elvonatkoztatva, tehát nem azt vizsgálva, hogy mikor kell csinálnia.

A számítási folyamatok dokumentálására az UML aktivitás diagramja, vagy egyszerű esetben a szöveges leírás lehet alkalmas.

2.6. Az osztályok operációinak azonosítása

Az eddigi lépések végrehajtása során elemeztük a rendszer működése során keletkező eseményeket és a végrehajtandó számítási, adat transzformációs tevékenységeket. Az események fogadását, a végrehajtandó műveletek a rendszer objektumainak kell végrehajtania. Ehhez meg kell határoznunk az egyes osztályokhoz telepítendő operációkat. Ezek lehetnek:

1. Események a dinamikus modellből: egy esemény fogadása és feldolgozása fogadó objektum osztályának operációja lehet. Ezzel szemben a kivételként kezelt eseményt legtöbbször maga a generáló objektum dolgozza fel.

2. Az állapotgép diagram tevékenységei és akciói: az állapotdiagram egy objektumhoz tartozik, így ezen műveletek egyértelműen az objektum osztályához telepítendők.

3. Összetett, több objektum együttműködését igénylő belső folyamatok: ebben az esetben a művelet nem egy osztály operációja lesz, ezért bonyolultabb a modellbe történő beépítés.

4. szabványos műveletek: az objektumhoz természeténél fogva hozzátartozó műveletek. Ilyenek lehetnek például az attribútumok elérése szolgáló, vagy a kapcsolatok kezelését lehetővé tevő operációk.

12. fejezet - A tervezési modell

Az analízis fázis során a rendszer egy absztrakt modelljét alakítottuk ki, ennek során a megoldandó probléma alkotóelemeit, a közöttük levő kapcsolatokat és a rendszerben zajló folyamatokat térképeztük fel.

A feladat azonban egy működő modell elkészítése, amelyben az analízismodellben szereplő elemeket, a rendszer szereplőit és folyamatait szoftver- és hardver elemek realizálják. Ezt a feladatot általában három lépésben hajtjuk végre: először az analízis modellt tervezési modellre, azt implementációs modellre, végül pedig implementált, működő modellre képezzük le.

A tervezési és az implementációs modell még továbbra is absztrakt modell, de egyre több, a megvalósításhoz szükséges részletet tartalmaz.

A tervezési modell a már meglévő információkat: a követelmények leírását, azaz a rendszer funkcionális nézetét, az analízis modell statikus és dinamikus nézetét felhasználva, ezeket bővítve és pontosítva készül el.

Az analízis modell elkészítése során a megoldandó probléma fogalmaira és folyamataira koncentráltunk, a tervezés során ezeket kiegészítjük a működéshez szükséges részletekkel.

A tervezés során végrehajtandó tevékenységeket a „Szoftvertervezés folyamata” című fejezet foglalja össze. Ebben és a következő fejezetben az ott leírtakat egészítjük ki olyan ajánlásokkal, amelyek a tervezési folyamatok végrehajtását segítik.

A tervezési folyamat része az implementálási eszközök, technológiák (programozási nyelvek, adatbázis kezelők, működtető környezetek stb.) kiválasztása. A munka során számos döntést kell meghozni, amelyekhez az implementáláshoz felhasznált eszközök ismerete is szükséges.

1. Tervezési szintű osztálydiagram

Az analízis fázis során elkészítettük a rendszer elemeit reprezentáló osztálydiagramot, amely statikus és dinamikus aspektusokat is tartalmaz.

A tervezési fázisban az elemzési osztályok további elemekkel, a már meglevő elemek további tulajdonságokkal egészülhetnek ki, illetve megjelennek olyan új osztályok, amelyek nem a szakterületi modell részeit reprezentálják, hanem a megvalósításhoz szükségesek. A tervezési osztálydiagram a tervezési lépések végrehajtása során szinte folyamatosan változik.

2. Rendszer architektúra kialakítása

Már az eddigi fejlesztési fázisokban is foglalkoztunk architekturális kérdésekkel. A használati eset leltár a rendszer funkcionális részrendszereit ábrázolja, az analízis modell során pedig a strukturális elemek alrendszereket különítettük el.

A tervezés során a működési szintű alrendszerek kialakítása a feladatunk, amelyek implementációs egységek lesznek. Az alrendszerek megtervezéséhez célszerű az alábbi szempontok figyelembe vétele:

1. Egy alrendszerbe logikailag összetartozó, együttműködő elemek kerüljenek, amelyek egy funkcionális egységet képeznek.

2. Az egyes alrendszerek közötti kommunikációt igyekezzünk minimalizálni. Ezzel segítjük a programozási munka szétosztását, párhuzamosítását, és egyszerűsítjük az alrendszerek tesztelését és későbbi karbantartását.

3. Egy rendszer felépítése során felhasználhatjuk korábbi rendszerek moduljait, könyvtári elemeket, keretrendszerek szolgáltatásait, amelyek szintén alrendszereket alkothatnak.

Az alrendszerek ábrázolására felhasználható UML eszközök:

1. csomagdiagram

2. komponens diagram

3. összetett struktúra diagram

3. Alrendszerek telepítési terve

Ma már a legtöbb rendszer egymással együttműködő számítógépek és egyéb intelligens eszközök igénybevételével működik. Ezért szükséges annak megtervezése, hogy az egyes alrendszerek milyen hardware elemekre, illetve milyen működtető környezetbe telepítendők. Azt is meg kell határozni, hogy a hardver elemek összeköttetéseit felhasználva hogyan biztosítható az alrendszerek kommunikációja.

A telepítési tervek az UML telepítési diagramja segítségével dokumentálhatjuk.

4. Adattárolás eszközének kiválasztása

A számítógépes rendszerek az esetek többségében perzisztens adatokat is kezelnek. Ezeknek háttértáron kell elhelyezkedniük. A tárolás formája függ az adatok mennyiségétől, a szükséges elérési módoktól és az adatok közötti kapcsolatok bonyolultságától. A leggyakoribb adattárolási módok a file és az adatbázis.

4.1. Adattárolás adatbázisokban

Ha nagy tömegű, kapcsolatokkal rendelkező adatokat kell tárolnunk, amelyeket sokféleképpen kell elérnünk, felhasználnunk az adatokat adatbázisba szervezzük. Jelenleg a legátalánosabban használt adatbázisok a relációs adatbázisok.

A tervezés során össze kell gyűjteni a perzisztens tárolást kívánó adatok körét, és a közöttük lévő kapcsolatokat. Ezek alapján fel kell építeni a szemantikai adatmodellt, amit relációs modellé kell konvertálni. Az UML ezekhez a modellekhez nem biztosít eszközöket, de a szemantikai modellre az ER vagy EER diagram, a relációs modellre a relációs diagram a legtöbbször alkalmazott eszköz.

Az adatbázis tervezés a fejlesztési folyamaton belül is speciális szakértelmet igényel, ezért az informatikus képzésekben külön tárgyak foglalkoznak ezzel a területtel. A fejlesztő cégek is gyakran külön erre specializálódott szakértőikre bízzák ezt a munkát.

A tervezés során célszerű eldönteni, melyik adatbázis kezelőt akarjuk használni, vagy éppen azt a tényt, hogy a rendszernek bármely adatbázis kezelővel működnie kell tudni.

4.2. Adattárolás file-okban

Speciális formátumban tárolható információk (például kép, hang, videó) esetén, ha nem túl sok elemet kell tárolni, célszerű lehet ezeket speciális szerkezetű file-okban tárolni. Ha azonban ez nagyon sok file tárolását jelentené, vagy az információk között bonyolult módon kell tudni keresni, az adatbázisban való tárolás a jobb megoldás.

Egyszerű, nem túl nagy elemszámú, ritkán változó adat tárolására választhatjuk a szövegfile-ban való tárolási formát is. Ilyen jellegű adatok lehetnek például a rendszer működését szabályozó adatok („konfigurációs file-ok”- például mi a használt adatbázis neve, mi legyen az alkalmazás nyelve), többnyelvű alkalmazás esetén az egyes nyelvekhez tartozó szövegek stb. Az ilyen módon tárolt adatok változtatása általában a file újraírását jelenti. A file-ban tárolt adatok elérése korlátozott, ezért többnyire úgy használjuk, hogy a tartalmát beolvassuk és a memóriában tároljuk.

A szövegfile-ok egy speciális formája az XML file. Az XML nyelv szabályai szerint létrehozott file-ok strukturált formában tartalmazzák az adatokat, és az adatokhoz való hozzáférés majdnem olyan rugalmas lehet, mint az adatbázisok esetén. XML formátumú file-ok kezelésére előre megírt eszközök állnak a programozók rendelkezésére. Ma már egyre gyakoribb a használata például a konfigurációs adatok tárolására, és alkalmas lehet rendszerek közötti adatátvitel megvalósítására is. Nagyon nagy számú, vagy gyors elérést követelő adatok esetén az adatbázis használata a célszerű.

5. Konkurencia kezelése

Az analízis modell szekvencia diagramjainak és a kommunikációs diagramjainak az átvizsgálása, pontosítása is a tervezési lépésekhez tartozik.

Ezen diagramok vizsgálata során az objektumok az alábbi két csoportba sorolhatók:

1. aktív objektum: képes a saját állapotát megváltoztatni és üzeneteket küldeni anélkül, hogy üzenetet kapott volna,

2. passzív objektum: ha kap egy üzenetet (meghívódik egy metódusa) képes megváltoztatni az állapotát és üzenetet küldeni, de ezután újra egy következő üzenetre vár.

Programozási szempontból minden aktív objektum egy külön végrehajtási szálat igényel. A sok szálból álló program futása során erőforrás igényes, programozása bonyolultabb, és osztott erőforrások kezelését, kölcsönös kizárási mechanizmusok alkalmazását igényelheti. Az aktív objektumok számát tehát a lehető legkisebbre (ideális esetben egyre) kell csökkenteni.

Az aktivitás diagramok kapcsán már említettük a lényegi és nem lényegi konkurencia fogalmát. A nem lényegi konkurencia esetén az eseménynek feldolgozása szekvenciába rendezhető, és a részt vevő objektumok passzívvá válhatnak.

A lényegi konkurencia esetén is sokszor lehetőségünk van új (nem a feladat végrehajtásához szükséges, hanem „aktivizáló") üzenetetek beiktatásával addig aktív objektumokat passzívvá tenni.

6. Vezérlés elvének kiválasztása

A programok legegyszerűbb (és legkönnyebben implementálható) vezérlési elve az eljárás orientált vezérlés. Általában ilyen programokat írunk a programozás alapjainak tanulása során. Ilyenkor a program futása egy kitüntetett metódus meghívásával indul, és metódus hívások sorozataként tekinthető. A feltételes utasítások alternatív hívási láncok kialakítását teszik lehetővé.

Ez a vezérlési mód az interaktivitásnak csak elég korlátozott formáját teszi lehetővé: ha egy programnak adatbevitelre van szüksége, kéri azt, és addig várakozik, amíg megkapja a kért adatot, majd folytatja a munkát.

A mai programok ablakos felhasználói felületei számára ez a vezérlési mód nem elegendő, ezért kidolgozták az esemény vezérelt vezérlési módot. Ebben az esetben minden inputot egy központi elosztó objektum kap meg, amely azokat különböző típusú eseményként kezeli, és adatokat is kapcsol hozzá. (Például egy egérkattintás egy esemény, és adata, hogy melyik pozícióban történt.) Az esemény jellege és adatai alapján aztán az eseményt átadja egy megfelelő objektum esemény kezelő metódusának, amely lefut. (Például az egérkattintás a koordináták alapján egy OK gomb fölött történt – ekkor ennek a gombnak az eseménykezelőjére kerül a vezérlés).

A lényegi konkurenciát tartalmazó programok esetén minden egyes aktív objektum egy külön vezérlési szálba kerül. Ennek lehetséges megvalósításai:

1. Több szálú program. Az egyes szálak közötti kommunikáció osztottat elérhető memória területeken keresztül történhet.

1. Több taszkból álló alkalmazás. Az egyes aktív objektumok működtetését egy-egy processz végzi. A kommunikáció interprocessz kommunikációs mechanizmusokkal oldható meg.

2. Osztott alkalmazás. Az egyes aktív objektumok működtetését külön gépeken futó processzek végzik. A kommunikáció hálózati együttműködést igényel.

7. Rendszer határfeltételeinek meghatározása.

Egy alkalmazásnak az üzemszerű működésén kívül speciális határállapotai is lehetnek. A tervezés feladata ezeknek a határállapotoknak a kezelése.

7.1. A rendszer telepítése

Egy összetett alkalmazás telepítése az egyes moduljainak az installálásán túl számos egyéb tevékenységet is magában foglalhat. Ilyenek lehetnek például:

1. az adatbázis kezdeti állapotának elállítása, esetleg már meglevő adatbázisok konvertálásával

2. felhasználók felvétele, jogosultságok meghatározása

Osztott alkalmazás esetén ez a munkafolyamat sok gépet érinthet.

Ezeket a tevékenységeket egy alkalmazás életében csak egyszer kell elvégezni.

7.2. A rendszer üzemszerű indulása (inicializálás)

Meg kell határozni, hogy a rendszer indulásakor milyen tevékenységek elvégzése szükséges az üzemszerű működés előkészítéséhez. Tipikusan ilyenek lehetnek például az adatbázis kapcsolat, hálózati kapcsolat létrehozása, bejelentkezési ablak megjelenítése stb.

7.3. A rendszer üzemszerű leállása (terminálás)

Az üzemszerű leállásnál szükséges lehet tranziens adatok lementése, erőforrások felszabadítása stb.

7.4. Hibás befejeződés

Már a tervezés során fel kell készülni arra, hogy rendszernek a hibaállapotok kezelésére is fel kell készülnie. Ilyen lehet például egy adatbázis kapcsolat vagy hálózati kapcsolat megszűnése. Ezek a hibaállapotok a működés során váratlanul léphetnek fel, de a tervezés során előre számítani lehet rájuk. A felhasználónak a hibaállapotról korrekt tájékoztatást kell kapnia, és az alkalmazásnak gondoskodnia kell arról, hogy a lehetőségekhez képest a legkevesebb adatvesztéssel kezelje a problémát.

8. Felhasználói felület tervezése

A mai alkalmazások döntő többsége interaktív működésű, tehát a felhasználói felület alapvető részét képezi.

A felhasználói felületek tervezése határterület a grafikusok, web dizájnerek és a programozók szakmái között.

„A Szoftvertervezés folyamata” című fejezet részletesebben foglalkozik ezzel e kérdéssel.

9. Külső interface tervezése

Számos alkalmazásnak kell más rendszerekkel kommunikálnia. Ezek a kapcsolatok mindig egyedileg tervezendők, mert a külső rendszerek interfészétől függenek. Számos esetben komoly feladat ennek a megoldása, főleg régebbi, rosszul dokumentált rendszerek esetén.

Mivel a mai viszonyok között a különböző rendszerek közötti kommunikáció igénye egyre gyakoribb, számos szabványosított megoldás létezik ennek a problémának a megoldására (ez többnyire XML alapú adatcserét jelent). Ezért a modern fejlesztésű rendszerek esetén az összekapcsolás könnyebben megoldható.

13. fejezet - Az implementációs modell

Az implementációs modell az utolsó absztrakt modell a működő változat előtt. Minden információt tartalmaz, amik a megvalósításhoz kell. Úgy is fogalmazhatunk, hogy ha lenne olyan virtuális gép, amely képes lenne végrehajtani az implementációs modellben foglaltakat, az implementációs fázis elmaradhatna.

A tervezési és az implementációs modell a gyakorlatban gyakran összemosódik, elválasztásuk szinte csak elméleti jelentőségű. Egy egyszerűbb alkalmazás fejlesztése esetén nincs éles határvonal a kettő között. Egy összetettebb alkalmazás fejlesztése során pedig a munka ütemezésétől függően az egyes részrendszerek aktuális modellezési fázisa egészen eltérő lehet egy adott időpillanatban: bizonyos alrendszerek lehet, hogy már az implementáció vagy tesztelés fázisában járnak, míg mások még az analízis vagy tervezés szintjén.

Az implementációs modell előállítása és az implementációs fázis is összemosódhat. A modellező eszközök re-engineering funkcióját kihasználva például sok fejlesztés során alkalmazott „trükk” az, hogy implementáljuk az osztályok „vázát” a tervezési modell alapján, majd ebből dokumentációs célokra előállítjuk az implementációs osztálydiagramot.

Az implementációs modell elkészítése az alábbi munkafázisokat foglalhatja magában.

1. A megvalósítási osztálydiagram

A megvalósítási szintű (implementációs) osztálydiagramok a programnyelvre fordításhoz szükséges valamennyi információt tartalmazzák. Számos vizuális modellező eszköz képes ezekből a diagramokból a kiválasztott programozási nyelv szabályainak megfelelő kód generálására is. Ez a generált kód általában nem teljes (például tartalmazza a metódusok vázát, de nem tartalmazza a törzsét.)

A modellező eszközök képesek a fordított irányú leképezésre is: egy program kódjából elő tudják állítani az implementációs osztálydiagramot (re-engineering). Ez a szolgáltatás egy új alkalmazás fejlesztése során alkalmas lehet annak ellenőrzésére, hogy az implementáció során nem tértünk-e el az előzetes tervektől. Különösen hasznos lehet a kódból a statikus modell előállítása abban az esetben, ha egy már meglévő rendszer továbbfejlesztése a feladatunk, mivel a generált modell elősegíti a rendszer működésének a megértését.

2. Algoritmus tervezés

Az egyes folyamatok végrehajtását lehetővé tevő algoritmusok kiválasztása. Számos feladatra léteznek kidolgozott algoritmusok, így új algoritmus kidolgozására ritkán van szükség. Egy probléma általában ismert algoritmusok adaptációjával vagy kombinációival megoldható.

Az algoritmusok dokumentálására számos eszköz áll rendelkezésünkre. Használhatók matematikai jelölések, az UML aktivitás diagramja, egyszerű folyamatábra, pszeudó kód vagy akár tényleges programkód.

3. Asszociációk tervezése

Ennek a lépésnek a feladata az osztálydiagramon jelölt kapcsolatok implementációs formáinak megtervezése. A pontos implementációs sémák függhetnek az alkalmazott programozási nyelvtől.

1. Egyirányú asszociációk

1. Kétirányú asszociációk: Implementálásukhoz a call-back technika ajánlott. A megoldás az, hogy a tartalmazó (vagy hívó) objektum a metódus hívás során az üzenethez tartozó paraméterek mellett átadja paraméterként a saját címét vagy referenciáját a tartalmazottnak (vagy hívottnak), amely ez alapján „visszahívhat” egy megadott metódust.

4. Láthatóság biztosítása

A forgatókönyvek, szekvencia és kommunikációs diagrammok alapján látható, hogy mely objektumoknak kell tudni elérniük egymást. A láthatóság biztosítása szintén erősen programozási nyelv függő.

5. Nem objektum orientált környezethez való illesztés

Gyakran szükség lehet kész függvénykönyvtárak, vagy már létező alkalmazások integrációjára. Ezek nem feltétlenül objektum orientált szemléletűek, és esetleg az adott fejlesztés nyelvétől eltérő programozási nyelven készültek. Az ilyen integrációs problémák mindig egyedi kezelést igényelnek. A megoldás általában megfelelő „csomagoló" (wrapper) osztályok írása, amely többnyire az implementációs környezet alapos ismertét igényli.

Még gyakoribb, hogy az alkalmazás relációs adatbázist használ. A relációs szemlélet nem illeszkedik az objektum orientált szemlélethez, ezért a kettő között valamilyen hidat kell kialakítani. Ennek jelenleg két szokásos megoldása van.

5.1. Adatbázis elérési felület

A programozási nyelvek általában biztosítanak olyan API-t, amelynek segítségével egy adatbázis kapcsolat megnyitása s SQL parancsok kiadása és a lekérdezés eredményének a feldolgozása megvalósítható. (Ilyen például a Java JDBC API-ja.) Ennek a megoldásnak a hátrány az, hogy az egyes adatbázis kezelők felülete nem feltétlenül azonos. Ha ezt a megoldást választjuk, akkor a tervezés során ügyelni kell arra, hogy az adatbázis elérés egy jól definiált, alkalmazás specifikus interfészen keresztül legyen implementálva. Így az adatbázis kezelők eltérései csak néhány, jól behatárolható osztály implementációját érintik.

Az egyszerűbb alkalmazások esetén, és ha a követelmények között szerepel egy meghatározott adatbázis kezelő alkalmazásának az előírása, ez a módszer jól alkalmazható.

5.2. Objektum-relációs leképezés (ORM)

Ennek a megközelítésnek az alapgondolata az, hogy egy osztálynak feleltessünk meg egy adatbázis táblát, amelynek a soraiban (rekordjaiban) az osztály egyes objektumai tárolódnak. Az adatbázis tábla mezői az osztály adattagjainak feleltethetők meg. Ha egy API vagy szoftver réteg képes biztosítani az objektumok és az adatbázis táblák közötti automatikus leképezést, akkor nem kell külön adatmodellt készíteni, és az adatbázis műveletekkel foglalkozni, ami az alkalmazás fejlesztését meggyorsítja.

A gyakorlatban használható leképezésnek biztosítania kell az objektumok közötti kapcsolatok adatbázis táblákba történő leképezését is.

Ma már számos ilyen API (például a Java JPA) és keretrendszer (például a Java alapú Hibernate) létezik ennek a problémának a megoldására.

6. Ütemezés

Az aktív objektumok számától függően ütemezési problémát is meg kell oldani egy alkalmazás fejlesztése során.

A lehetséges esetek:

1. Nincs aktív objektum a rendszer modelljében. Ebben az esetben létre kell hozni egy "alkalmazás" objektumot, aminek a feladata az alkalmazás indítása.

2. Egy aktív objektum van. Ebben az esetben ennek az objektumnak a feladata az alkalmazás indítása.

3. Több aktív objektum van. Ebben az esetben az ütemezést meg kell oldani

Az ütemezés implementációjának módja függ

1. a használt fejlesztői környezet lehetőségeitől,

2. a futtatási környezet tulajdonságaitól.

7. Osztályok egyedi vizsgálata:

Minden osztályt meg kell vizsgálni abból a szempontból, hogy önmagában konzisztens-e a megvalósítása. Például megfelelő konstruktorokkal rendelkezik, megfelelőek-e a destruktor jellegű metódusok, megoldott-e az objektum másolása, az adattagokhoz szükséges kezelő függvények rendelkezésre állnak-e stb.

8. Implementációs adatstruktúrák:

Az alkalmazás folyamatait megvalósító algoritmusok működéséhez számos adatstruktúra megvalósítása lehet szükséges. Minden mai programozási nyelv rendelkezik beépített adatstruktúrákkal, és API szinten további lehetőségekkel. Ha ezek az algoritmusokhoz nem elegendőek, új elemként megjelenhetnek az implementációs modellben a szükséges speciális adatstruktúrákat megvalósító osztályok.

14. fejezet - Tervezési minták

1. Bevezetés

A program tervezési minták (design patterns) gyorsítják, segítik a fejlesztés menetét, használatuk javasolt. Bár a megismerésük, megtanulásuk kezdetben sok időt, energiát igényel, de ez a későbbiekben bőségesen megtérül. A követendő minták objektum orientált módszerre épülnek, és a gyakran előforduló problémákra, esetekre ad megoldásokat. Nem mesterségesen kreált új tudomány, csak a gyakorlati tapasztalatokon alapuló dokumentáció. Definíciója: „Egymással együttműködő objektumok és osztályok leírásai, amelyek testreszabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben.”. Egy kezdőkből álló csoportban erősen ajánlott az alkalmazásuk, mert így a programozók tudnak támaszkodni valamire, és egymást jobban tudják helyettesíteni. Használatuk előnyei:

- tapasztalatszerzési időt lerövidítik,
- lerövidítik a tervezési időt,
- megkönnyítik a kommunikációt, a tagok helyettesíthetőségét.

2. Létrehozási minták(Creational patterns)

A létrehozási minták osztályokat egyedeket példányosítanak. Két részre oszthatjuk ezeket: osztály vagy objektum létrehozó minták. Amíg az új osztályt létrehozó minták eszköze a származtatás, addig a példányt létrehozó mintáké a delegáció. Általában szeretnénk e létrehozás módját, a létrehozandó elem szerkezetét elrejteni, és a létrehozás módját függetleníteni az elemektől.

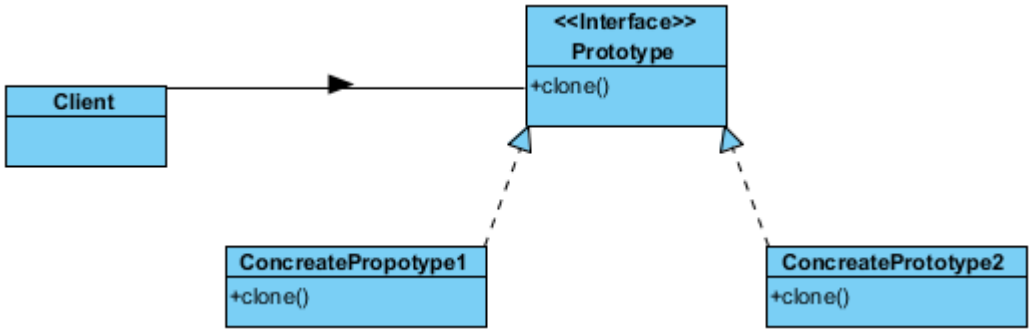
2.1. Prototípus (Prototype)

Cél
: Elemek létrehozása előzetes minta (prototípus) alapján

Motiváció
: Akkor használjuk, ha egy objektum létrehozásának módjától függetlennek kell lennie a rendszernek. Nem szeretnénk gyárakat építeni. Gyakran használjuk, ha a példányosítandó osztály típusa futásidőben derül ki.

Alkalmazhatóság
: Akkor használjuk, ha a rendszernek függetlennek kell lennie az új példány létrehozásától, szerkezetétől, megjelenítésétől.

Felépítés



Résztvevők

1. Client osztály, ami az új objektumot szeretné létrehozni, a prototípust kéri meg önmaga klónozására.
2. Protype, ami egy interfészt deklarál önmaga klónozásához.
3. ConcretePrototype osztály, ami implementálja klónozó függvényt (saját maga másolására)

Együttműködés
A kliens a prototípus interfészt használva megkéri a konkrét prototípust objektumot önmaga klónozására.

Következmények
Elrejti a termék osztályok részleteit, függetleníti azt a termék szerkezetétől. Nem kell ismernie a kérő (kliens) objektumnak a konkrét elemnek a nevét, amit létre akar hozni, hiszen az interfészt használja. Aki az interfészt implementálja, azt lehet létrehozni, és nem kell arról a klienst értesíteni.

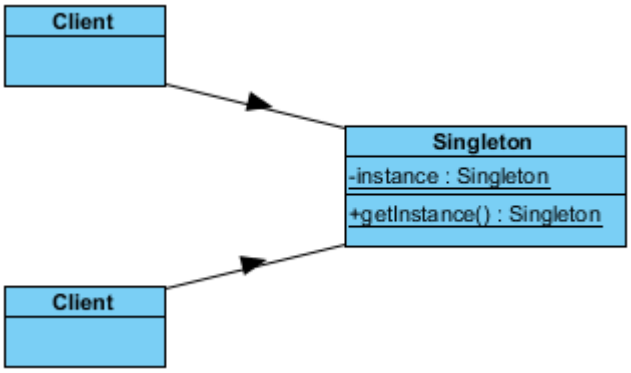
2.2. Egyke (Singleton)

Cél
Biztosítani azt, hogy egy osztályból csak egy példány létezzen a rendszerben, és az elérhető legyen globálisan több elem számára is.

Motiváció
Hogyan biztosíthatjuk, hogy egy osztálynak csak egy példánya létezzen, és ez példány könnyen elérhető legyen? A globális változó segítségével a globális használat könnyen biztosítható lenne, de nem akadályozza meg a több példány létrehozását. Helyesebb, ha magát az osztályt tesszük felelőssé azért, hogy privát elemként tartalmazza a példányt, és kérés esetén kiadja annak elérhetőségét csak olvasható módon.

Alkalmazhatóság
Csak olvasható módon biztosítani szeretnénk egy példányban futó objektum globális elérését. Tipikus példa az adatbázis kapcsolat (ConnectionPool), loggolást, és az üzenet sorok.

Felépítés



Résztvevők

1. Kliens osztály, ami kéri az egyetlen globális példányt.

2. Singleton definiál egy Instance (példány) műveletet, amivel a kliensek hozzáférhetnek az egyedüli példányhoz. Instance mindenképpen egy osztályművelet. Általában felelős a saját egyedüli példányának a létrehozásáért. Ha csak lehet tiltja a több példány létrehozását (privát konstruktor).

Együttműködés
A kliensek a Singleton példányát kizárólag a Singleton Instance műveletén keresztül érik el, amely ha nem létezett, akkor létrehozza, és visszatér a példánnyal. Általában a létrehozásnak nincs paramétere.

Következmények
Nem lehet létrehozni több példányt az objektumból. Globális változókat váltja ki, így szabályzottan (loggoltható, debuggolható) férhetünk a globális objektumhoz. Általában paraméter nélküli példányosítást tesz lehetővé, vagy a paraméterek nem a külvilágból származnak.

2.3. Építő (Builder)

Cél
Egy építési folyamattal több, különböző szerkezetű elemek létrehozása. A létrehozás folyamata független az ábrázolástól.

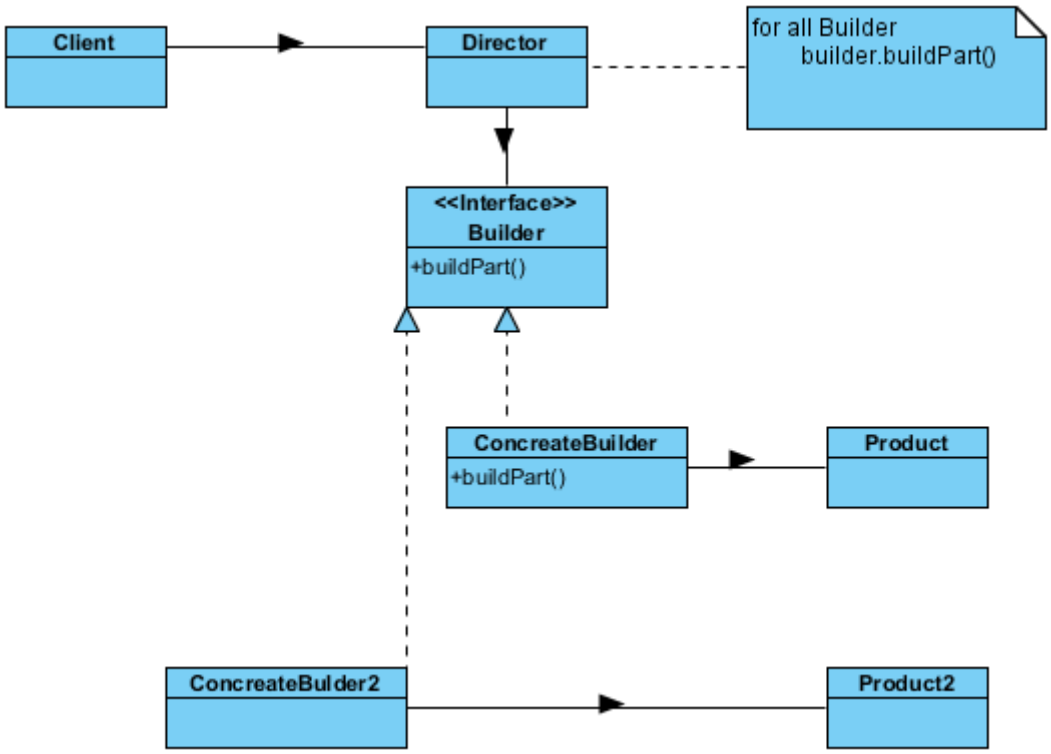
Motiváció

Akkor használjuk, ha lépésről lépésre kell létrehozni az elemet (több lépés), és a termék azonnal létrejön. Gyakran Factory mintával kezdődik a tervezés, fejlesztés, de ha túl sok lépésből áll, akkor áttérünk Builder-re.

Alkalmazhatóság

Akkor alkalmazzuk, ha a létrehozási folyamatnak függetlennek kell lennie, a létrehozás algoritmusa (inicializáló algoritmusok) független kell legyen a szerkezettől. Ha egy osztály sok rész-osztályt használ (komplex osztály), mindenképp használjuk ezt, mert a részek változásakor változtatni kell a létrehozó kódokat is. Új részek felvételét is kezeli, hoszen a használó (kliens) nem változik.

Felépítés



Résztvevők

1. Builder interfész, amely deklarálja a termékrész objektum létrehozására szolgáló absztrakt műveleteket.
2. ConcreteBuilder osztály, amely pontosan definiálja a Builder interfészben előírt létrehozó metódusokat, felépíti és összeállítja a termék részeit. Definiálja és nyomon követi az általa készített megjelenítési módokat. Metódus(oka)t biztosít a termék beolvasásához.
3. Director osztály, amely ismeri, és használja a Builder interfészt, kéri a termék létrehozását.
4. Product osztály, ami az elkészítendő osztály maga. A ConcreteBuilder felépíti a ennek a termék belső ábrázolását és definiálja az összeállítási folyamatokat.

Együttműködés

1. A kliens létrehozza a Director objektumot és beállítja a kívánt Builder objektummal.
2. A Director értesíti a builder-t, ha egy objektum részét fel kell építeni.
3. A Builder kezeli a dircetor-tól érkező kéréseket és a részeket ad a termékhez.
4. A kliens lekéri a terméket a builder-től.

Következmények

Csak a director felügyelete mellett lehet felépíteni a terméket, amely több lépésből áll. A director csak akkor veszi át a terméket, ha az már kész van.

2.4. Elvont gyár (Abstract Factory)

Cél

Létrehozási interfészt biztosítani kapcsolódó vagy függő objektumok családjának, a konkrét osztályok megadása nélkül.

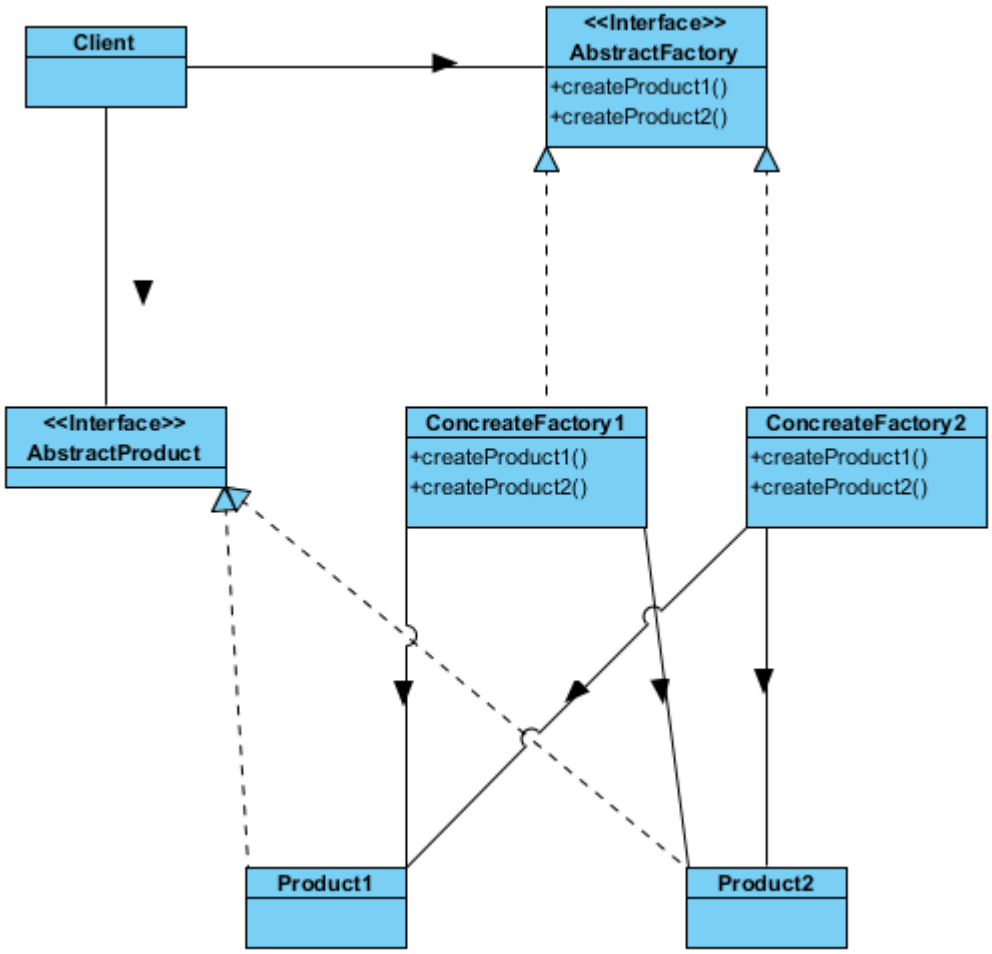
Motiváció

Akkor használjuk, ha a létrehozandó objektumokat családba rendezhetjük, azaz függőség van közöttük. Általában a létrehozás kevés lépésből áll.

Alkalmazhatóság

Akkor használjuk az Abstract Factory mintát, ha -a rendszernek függetlennek kell lenni attól, hogyan hozza létre, állítja össze és jeleníti meg a termékeit. A rendszernek be kell állítania egyet termékcsalád közül, a kapcsolódó termékobjektumok egy családját együttes használatra tervezték, és ezt a megszorítást ki kell kényszeríteni, gondoskodni kell a termékek osztálykönyvtáráról, és csak az interfészüket fedhetjük fel, de a megvalósításukat nem.

Felépítés



Résztvevők

1. AbstractFactory interfész, amely az absztrakt termékobjektumokat létrehozó interfészt deklarálja. Több is lehet belőle, mindegyiket ismerni kell a kliensnek.

2. ConcreteFactory osztály, amit a konkrét termékobjektumok létrehozására szolgáló műveleteket implementálja.

3. AbstractProduct interfész, amely a termékobjektumok egy típusának interfészt deklarálja. Több is lehet belőle, mindegyiket ismerni kell a kliensnek.

4. ConcreteProduct osztály, amely definiálja a létrehozandó termékobjektumot a megfelelő konkrét factory (gyár) mellett, implementálja az AbstractProduct interfészt.

Együttműködés
: A ConcreateFactory egy példánya jön létre futásidőben, amely ismeri a konkrét termék osztályt. Ha a kliens más osztályú objektumot akar létrehozni, akkor más ConcreateFactory kell használnia.

Következmények
: A konkrét osztályok elszigeteltek, a kliensnek nem kell ismernie azokat. Megkönnyíti a termékcsaládok cseréjét, de megnehezíti az újak bevezetését.

3. Szerkezeti minták (Structural Patterns)

Objektumok közötti kapcsolatok kialakítására szolgálnak. Az objektumminták azt mondják meg hogyan ragasszuk össze objektumainkat, hogy új szolgáltatásokat nyújtsunk. Oly módon, hogy az objektum(ok) változása ne okozzon a kapcsolatokban is változást. Osztályminták örökléssel felületeket vagy megvalósításokat építenek fel. Egy felületet egy másikhoz illeszt.

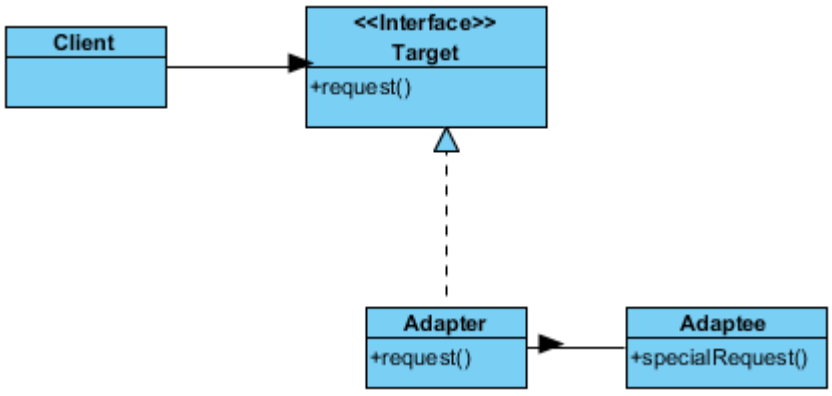
3.1. Illesztő (Adapter)

Cél
: Felület átalakítása egy másikra. Összeférhetetlen osztályok együttműködésének biztosítása. Ebből következően a két osztálynak nem szükséges egymásról tudni. A kapcsolatot az illesztő(adapter) valósítja meg.

Motiváció
: Néha előfordul olyan, hogy egy újrahasznosításra tervezett típust nem elég előrelátóan írtak meg és később mégsem használható fel az új kódba. Az is megtörténhet, hogy egy elemkészlet nem beilleszthető, sőt külső készítőtől származó könyvtárban található. Ilyenkor jön jól az illesztő, mely kompatibilis az új felülettel, így már beilleszthetővé válik az új kódba.

Alkalmazhatóság
: Egy meglévő osztály felhasználásánál, aminek felülete nem megfelelő. Újra felhasználható osztály létrehozásánál, ami képes előre nem ismert osztályokkal együtt működni.

Felépítés



Résztvevők

1. Cél felület (Target): Létrehoz egy tartomány specifikus felületet az Ügyfél osztály (Client) számára.

2. Ügyfél osztály (Client): A teljes folyamat haszonélvezője, felhasználja Cél felület (Target) objektumait az interfész segítségével.

3. Illesztendő felület (Adaptee): Meghatározza a létező interfészt, ami az illesztéshez kell.

4. Illesztő felület (Adapter): Összeilleszti a Cél felület (Target) interfészét az Illesztendő felület (Adaptee) – vel.

Együttműködés
: Az Ügyfél osztály (Client) - ok egy Illesztő felület (Adapter) típusú példánynak műveleteit hívják meg. Az Illesztő felület (Adapter) az Illesztendő felület (Adaptee) műveleteit hívja meg.

Következmények
: Alkalmazásának előnyeit és hátrányait megkülönböztetjük az osztály- és objektumillesztők szempontjából.

  Objektumillesztők szempontja alapján: Egyetlen Illesztő felület (Adapter) több Illesztendő felület (Adaptee) is működik, azaz egy ősosztállyal és összes alosztályával. Lehetőség van arra is, hogy az összes illesztett osztályt egyszerre bővítsük ki további funkciókkal. Nehezebb felülírni az Illesztendő felület (Adaptee) osztály folyamatait. Ezért származtatnunk kell azt, és az Illesztő felületnek (Adapter) egy leszármazott-osztályra kell majd hivatkoznia az eredeti helyett.

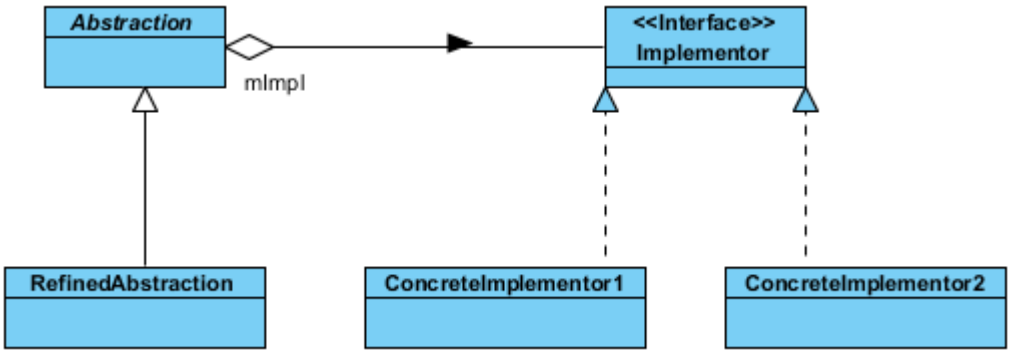
  Osztályillesztők szempontja alapján: Az illesztendő osztály a célfelülethez pontosan egy illesztő (adapter) felület támogatásával kapcsolódik. Ennek eredményeként nem alkalmazható amennyiben egy osztályt és összes leszármazottját akarjuk illeszteni. Mód van az Illesztendő felület (Adaptee) osztály némely funkcióinak sima felülírására, mivel az Illesztő felület (Adapter) az Illesztendő felület (Adaptee) osztály leszármazottja. Csak egy objektumot alkalmazunk, és a kevesebb memória-hivatkozás vezet el a Illesztendő felület (Adaptee) osztályig.

3.2. Híd (Bridge)

Cél
: Az elvont ábrázolást („felület”) és a megvalósítás elválasztása, ezáltal azok egymástól függetlenül is módosíthatóak. Míg az illesztő (Adapter) mintát akkor használjuk mikor különböző felületeket szükséges összekapcsolni, addig a híd (Bridge) olyan minta, amit előre gondolva, tervezett módon kell használni.

Motiváció
: Olyan esteknél mikor egy feladatot, alkalmazást más vagy több felületen, platformon szeretnénk megvalósítani, implementálni. Ez esetben az öröklléssel történő megvalósítás nem mindig célszerű, mert maradandó kötést hoz létre. Ezért ilyenkor célszerű a Híd (Bridge) használata. Ha például egy ablakkezelőt több felületen szeretnénk megvalósítani, akkor az ablaktípusokat nem szükséges megírni minden felületre. Elég a Híd (Bridge) minta használata, amivel megvalósítunk minden felületre egy alosztályt. Az ablaktípusok pedig a Híd (Bridge) felület-függvényeit használják.

Alkalmazhatóság
: El szeretnénk kerülni az elvont fogalom és a megvalósítás közti szoros kapcsolatot. Bővíthetőség biztosítása a különböző felület és megvalósítás között. A felület változása nem lehet hatással az ügyfélre.

Felépítés
: 

Résztvevők
: 1. ElvontÁbrázolás felület (Abstraction) Felületet definiál az elvont fogalomhoz. Egy Megvalósító objektumra vonatkozik. Magasabb műveleteket biztosít.

  2. FinomítottElvontÁbrázolás felület (RefinedAbstraction) Az abstraction kibővítésére szolgál.

3. Megvalósító felület (Implementator) Felületet definiál a megvalósító osztályok számára. Általában csak alapműveleteket biztosít.

4. KonkrétMegvalósító osztály (ConcreteImplementor) Implemetálja a Megvalósítófelületet.

Együttműködés
Az ElvontÁbrázolás felület (Abstraction) továbbítja a kéréseket a hozzá tartozó Megvalósító felület (Implementator) -nek.

Következmények
Alkalmazásának előnyei: Különválaszthatjuk használatával az absztrakciót és az implementációt. Ezáltal az implementáció futásidőben, dinamikusan állítható be. Különválasztásból adódik, hogy az implementáció részleteit az ügyfelek elől elrejthetjük. Az implementációs hierarchiának köszönhetően könnyebben bővíthetjük őket és gyorsítható a fordítás is. A megvalósító objektum máshol is alkalmazhatjuk.

3.3. Összetétel (Composite)

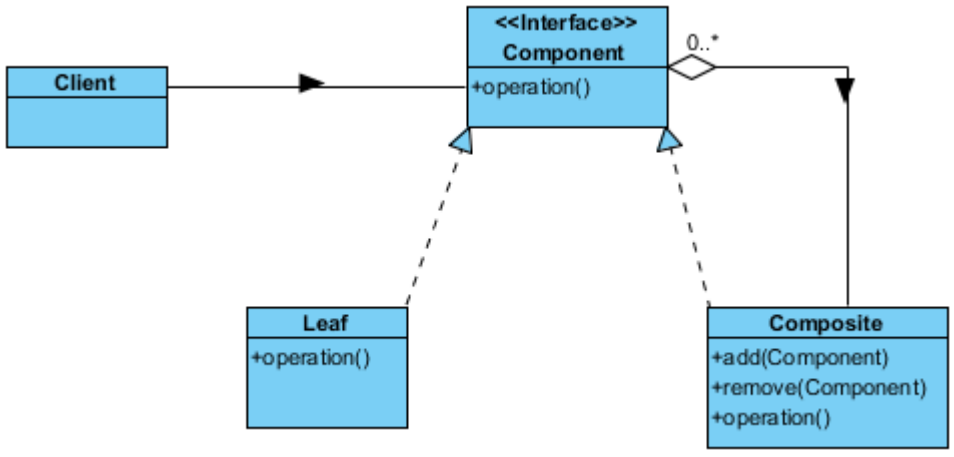
Cél
Az objektumok faszerkezetbe való ábrázolásának megvalósítása. Egységesíteni a kezelését az objektumnak és az összetételnek. A faszerkezet részei:

- levél: egyszerű objektum
- kompozíció: levelek együttese

Motiváció
Olyan grafikus alkalmazás, amely lehetővé teszi egyszerű alkotóelemekből bonyolult grafikus objektumok létrehozását. Például szimpla szöveg – és vonalelemeket kategorizálhatunk, és azokat egy egységként kezelhetjük. Ilyen a flash szerkesztő azon része is ahol különböző objektumokat vonalakat alakzatokat rakhatunk a rajzfelületre. De a programnak meg kell különböztetni az egyszerű alkotóelemeket és az összetett elemeket, mert csak az utóbbihoz lehet elemeket adni illetve elvenni. Ezeken túl vannak olyan operációk, melyek alkotó –és összetett elemekre is alkalmazhatóak.

Alkalmazhatóság
Rész egész viszony alkalmazásánál. Felhasználó ne érezzen különbséget komplex és egyszerű objektumok között.

Felépítés



Résztvevők

1. Elem felület (Component) Összetétel objektumok felületének kialakítása. Megfelelő alapértelmezett viselkedést biztosít.

2. Levél osztály (Leaf) A fa szerkezet leveleit írja le (levél = nincs gyereke). Az összetételt és az összetétel objektumainak viselkedését írja le.

3. Összetétel osztály (Composite) A gyermekkel rendelkező elemek viselkedését írja le. Gyermekeket tárol. Végrehajtja az Elem felület (Component) műveleteit.

4. Ügyfél osztály (Client) Különböző műveletet hajt végre az Összetétel osztály (Composite) objektumaival az Elem interfészen keresztül.

Együttműködés
Az Ügyfél osztály (Client) az Elem (Component) osztály felületén keresztül kapcsolódik az objektumhoz. Ha a címzett egy Levél osztály (Leaf), akkor végrehajtódik. Amennyiben a címzett Összetétel osztály (Composite), akkor továbbítódik az összes Levél osztály (Leaf) -hoz.

Következmények
Alkalmazásának előnyei és hátrányai: Olyan osztályhierarchiát valósit meg, ami az alap objektumokat rekurzívan építi fel egyre bonyolultabb objektumokká. Egyszerűbbé válik az kliensprogram, mert nem szükséges megkülönböztetni az alap és az összetett objektumokat, azokat egységesen kezelik szokásos esetekben. Könnyebb az új komponenseket hozzáadása és nem kell a klienst megváltoztatni. Az új komponensek könnyebb hozzáadásával viszont túl általánossá válhat az alkalmazás. objektumhoz. Ha a címzett egy Levél osztály (Leaf), akkor végrehajtódik. Amennyiben a címzett Összetétel osztály (Composite), akkor továbbítódik az összes Levél osztály (Leaf) -hoz.

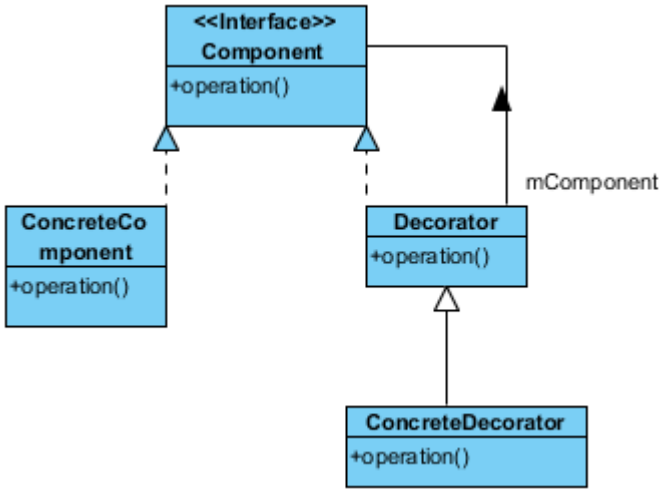
3.4. Díszítő(Decorator)

Cél
Az objektumokat lehet kibővíteni további funkciókkal dinamikus módon. Rugalmas alternatívát nyújt az alosztályok létrehozásához.

Motiváció
Főleg grafikus felhasználói felületeknél (GUI) használjuk. Tegyük fel, hogy egy GUI rendszer adott, például egy szövegszerkesztő és ezt szeretnénk felruházni díszítő tulajdonságokkal vezérlőelemeken keresztül. Ilyen tulajdonság a keret (border), görgetősáv (scrollbar) vagy akár az árnyék is. Ezért ezeket egy díszítő objektumba célszerű ágyazni, mert így nem statikusan dől el az alkalmazásuk, hanem a dinamikusan. Ezáltal az ügyfelek döntenek arról, hogy mikor alkalmazzák őket. A példa egy megoldása lehet az, hogy létrehozunk egy általános keret (border) és görgetősáv (scrollbar) osztályt és ebbe ágyazzuk bele az alap szövegnézet megfelelő elemét képviselő objektumot.

Alkalmazhatóság
Dinamikusan szeretnénk kibővíteni az egyes objektumok funkcionalitását. Átlátszóan szeretnénk kibővíteni az objektum felelőségi köreit, funkcionalitását. Amikor a származtatott osztályok használata nem előnyös.

Felépítés



Résztvevők

1. Elem (Component) Az objektumok számára meghatározza a felületet, ami dinamikusan bővíthető eltérő felelősegekkel.

2. KonkrétElem (ConcreteComponent) Objektumot definiál, mely az előzőhöz köthető kiegészítő felelőségekkel.

3. Díszítő (Decorator) Hivatkozást tart fent az Elem (Component) - re, és meghatároz egy interfészt ami az Elem (Component) interfészéhez illeszkedik.

4. KonkrétDíszítő (ConcreteDecorator) Az Elem (Component) - hez hozzáadja a felelőségeket.

Együttműködés
: A Díszítő (Decorator) a fenntartott hivatkozáson keresztül kéréseket küld az Elem (Component) objektumának. A kérelem küldése előtt vagy után további operációkat hajt végre.

Következmények
: Alkalmazásának előnyei és hátrányai: Díszítők (Decorator) láncolásával több tulajdonságot tudunk adni, egyszerűbb a többszöri felvételük. A kód rugalmasan újra felhasználható és objektumok felelősség köre is rugalmasan bővíthető, vegyíthető. Futásidőben határozható meg a Díszítő (Decorator), ellentétben az öröklődéssel. El lehet kerülni a hatalmas öröklődés fákat, de gondot okozhat a számtalan kisméretű objektum.

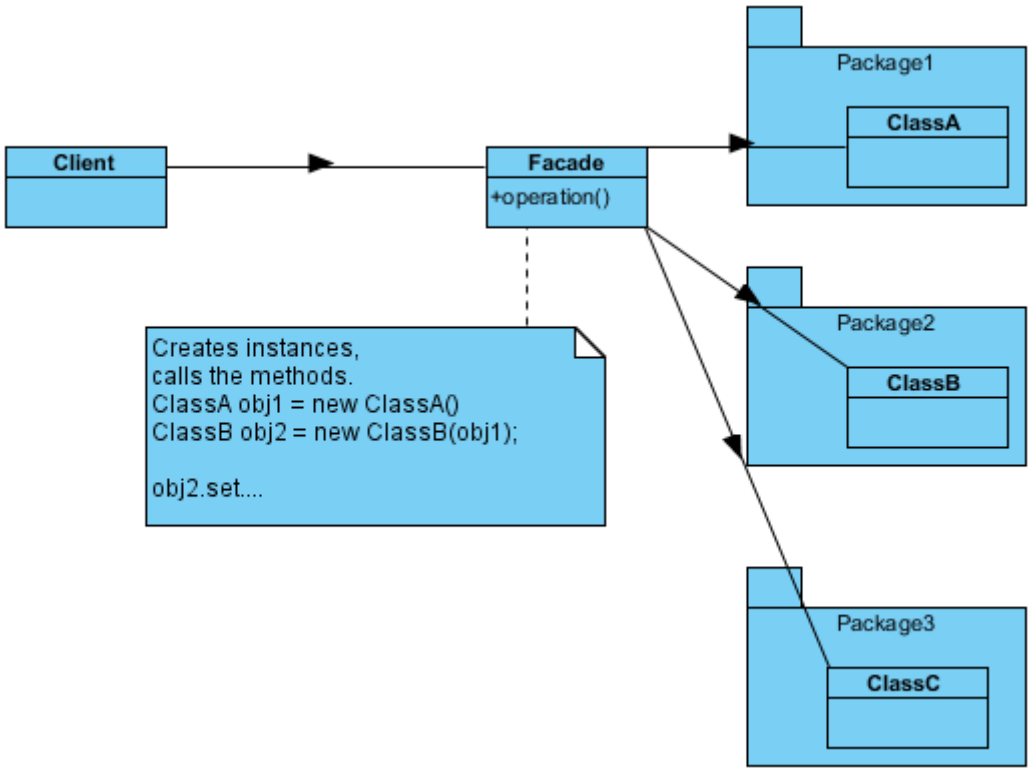
3.5. Homlokzat(Facade)

Cél
: Egységes (magasabb szintű) interfész megvalósítása egy alrendszer interfészeinek könnyebb használatára.

Motiváció
: Adott egy rendszer, sok interfésszel és sok osztállyal. Hogyan csökkenthetjük annak bonyolultságát, hogyan engedjük a felhasználót hozzáférni a rendszerhez, anélkül, hogy ismerné az összes osztály felépítését, metódusait, attribútumait? Erre jelent megoldást a rendszer alrendszerekre való bontása, ennek egyik megvalósítását biztosítja a Homlokzat(Facade) programtervezési minta. A Homlokzat (Facade) objektum egy szimpla felületet ad az adott alrendszer széleskörű szolgáltatásainak számára.

Alkalmazhatóság
: Egyszerű interfészt szeretnénk szolgáltatni egy összetett alrendszerhez. Az Alrendszer (Subsystem classes) és Ügyfél (Client) osztályai közti függőség csökkentése oly módón, hogy kizárólag a Homlokzaton(Facade) keresztül engedjük meg a kommunikációt. Így hordozhatóbbá is válik az alrendszer. A rendszerek közti kevesebb kapcsolat jobb újrafelhasználhatóságot idéz elő, aminek biztosítása napjainkban elengedhetetlen feladat a szoftverfejlesztésben. Továbbá alkalmazható még az Alrendszeri osztályok (Subsystem classes) rétegezése (Layers) esetén.

Felépítés



Résztvevők

1. Ügyfél (Client) Meghívja Homlokzat(Facade) eljárásait rajta keresztül kapcsolódik az Alrendszeri osztályokhoz (Subsystem classes).

2. Homlokzat (Facade) Kezeli az Ügyfél (Client) kéréseit. Tudja, melyik kérését, melyik Alrendszeri osztályhoz (Subsystem classes) kell továbbítania.

3. Alrendszeri osztályok (Subsystem classes) Nem tudnak a Homlokzat (Facade) létezéséről, vagyis részükről átlátszó. Ők valósítják meg a Homlokzat (Facade) objektumtól kapott megbízásokat, feladatokat.

Együttműködés

Az Ügyfelek (Client) a Homlokzaton (Facade) keresztül kérelmek formájában lépnek kapcsolatba az Alrendszeri osztályokkal (Subsystem classes). A Homlokzat (Facade) a megfelelő kérelmet a megfelelő Alrendszeri osztály (Subsystem classes) objektumhoz továbbítja. A Homlokzat (Facade) felületét az Alrendszeri osztályok (Subsystem classes) felületéhez illeszti. A valódi feladatot az Alrendszeri osztályok (Subsystem classes) objektumai végzik. Ha szükséges az egyes Ügyfelek (Client) közvetlenül is elérhetik az Alrendszeri osztályt (Subsystem classes).

Következmények

Alkalmazásának előnyei és hátrányai: Fő előnye hogy kombinálni tudjuk a nagyon összetett metódushívásokat és kód blokkokat egy egyszerű metódusba, ami elvégzi azokat. A gyártási kódnál könnyebb a használata és megértése, csomagok és könyvtárak között csökkentik a kódfüggőségeket. Gyenge kapcsolatokat alakít ki a Ügyfél (Client) kódok és más megengedett csomagok es könyvtárak között, ami nekünk lehetőséget add arra, hogy úgy változtassuk meg a csomagokat vagy a könyvtárak belső komponenseit, hogy az Ügyfél (Client) nem hivatkozik rájuk közvetlenül. Választást biztosít a könnyebb használat és a nagyobb általánosság között, mivel megengedi, hogy az alkalmazások használják az Alrendszeri osztályokat (Subsystem classes). Gondot okozhat a számtalan kisméretű objektum.

4. Viselkedési minták (Behavioral Patterns)

A viselkedési minták az osztályok és az objektumok közötti kommunikációt írják le. A középpontban az algoritmusok megvalósítása, és a felelősségi körök elosztása (kommunikáció) áll. Segítenek abban, hogy a

kapcsolatokra helyezzük a hangsúlyt, ahelyett hogy a vezérlésre kellene figyelnünk. Vannak osztály minták és objektum minták. Az osztályminták öröklődéssel rendelik az osztályokhoz a szükséges viselkedést. Az objektum minták meghatározzák a viselkedés és objektum kompozíciót, azaz hogyan működjenek együtt társobjektumok egy csoportja a több objektumot igénylő műveleteknél.

4.1. Parancs (Command)

Cél

Kérelmek objektumba ágyazása. Ezáltal a klienseknek különböző parancsokat adhatunk át, amit naplózhatunk, sorba rendezhetünk és a visszavonást (undo) kezelhetjük le. Azaz az egyes kérelmek teljesen kivizsgálhatóak és hatálytalaníthatók lesznek. Mivel objektumba vannak ágyazva a kérelmek így lehetőség van a kérések ideiglenes tárolására.

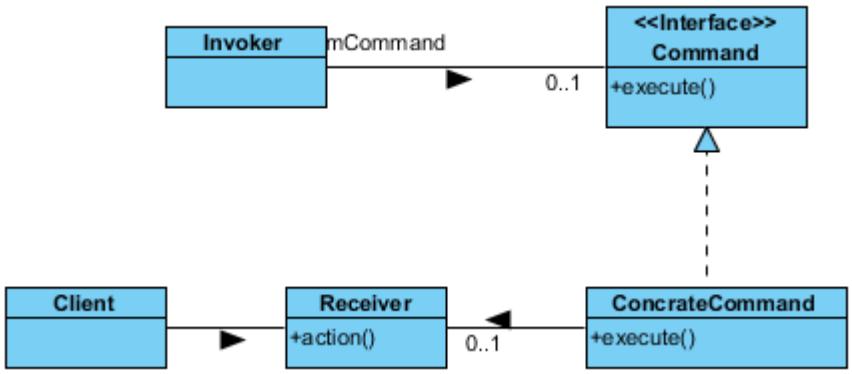
Motiváció

Néha muszáj, hogy kéréseket küldjünk egyes objektumokhoz úgy, hogy bármi ismeretünk lenne a kért folyamatról vagy a kérést fogadóról. A felhasználói felületek programozására gyakran használt könyvtárak, eszközkészletek többek között olyan objektumokat menüket és gombokat tartalmazhatnak, amelyek egy felhasználói esemény hatására indítnak el valamilyen műveleteket. Konkrét implementációt a menü, vagy a gomb objektumok nem képesek megvalósítani, mert csakis az eszközkészleteket használó alkalmazás tudja, mi mit csinál ilyenkor. Az eszközkészlet tervezői nem tudják előre, hogy a konkrét tevékenységet végül milyen objektum és hogyan hajtja végre. A Parancs (Command) mintával ezen kérések az elemkészlet objektumok számára megvalósíthatóvá válik oly módon, hogy a kéréseket is objektumként kezelik.

Alkalmazhatóság

Commit támogatás: a műveletek ismét végrehajtásának támogatása Undo támogatás: a műveletek visszavonásának támogatása, Unexecute művelet szükséges hozzá Wizard-ok Swing Progress bar Naplózás: változások naplózása rendszerösszeomlás, helyreállítás esetén.

Felépítés



Résztvevők

1. Parancs felület (Command) Létrehoz egy interfészt egy művelethez.

2. KonkrétParancs osztály (ConcreteCommand) Egymáshoz rendeli Fogadó osztály (Receiver) - t és egy végrehajtandó műveletet. Fogadó osztály (Receiver) megfelelő műveletének hívásával implementálja a Végrehajt (Execute) műveletet. Undo megvalósítása esetén KonkrétParancs osztály (ConcreteCommand) állapotát is tárolni kell.

3. Ügyfél osztály (Client) Létrehoz egy KonkrétParancs osztály (ConcreteCommand) - t és beállítja annak Fogadó osztály (Receiver) – át.

4. Hívó osztály (Invoker) Felkéri a Parancs felület (Command) - et a kérelem teljesítésére úgy, hogy meghívja annak Execute() metódusát.

5. Fogadó osztály (Receiver) A kérést fogadja, birtokában van az adott kérelemhez kapcsolódó műveletek végrehajtásához szükséges tudásnak. Bármelyik osztály lehet fogadó.

Együttműködés

Az Ügyfél osztály (Client) létrehoz egy KonkrétParancs osztály (ConcreteCommand) - t és meghatározza annak fogadóját. Valamelyik Hívó osztály (Invoker) elraktározza a KonkrétParancs osztály (ConcreteCommand) - t. A Hívó osztály (Invoker) kérelmet bocsát ki. A KonkrétParancs osztály (ConcreteCommand) műveleteket hív meg a fogadóján.

Következmények

Alkalmazásának előnyei és hátrányai: A Parancs (Command) minta alkalmazása feloldja a kapcsolatot a műveletet kezdeményező és az azt végrehajtó objektumok között. A parancs objektumok is ugyanúgy használhatók és kibővíthetők, tipikus objektumok. A parancsok összetett parancsokká rendezhetők, mint például makrót képezve. Parancsot osztálymódosítás nélkül vehetünk fel.

4.2. Megfigyelő (Observer)

Cél

Objektumok közötti egy-több függőségi kapcsolat létrehozása. Egy kiválasztott objektum módosulásáról értesítő információt küldeni a tőle függő objektumoknak, amik ezek alapján frissülnek.

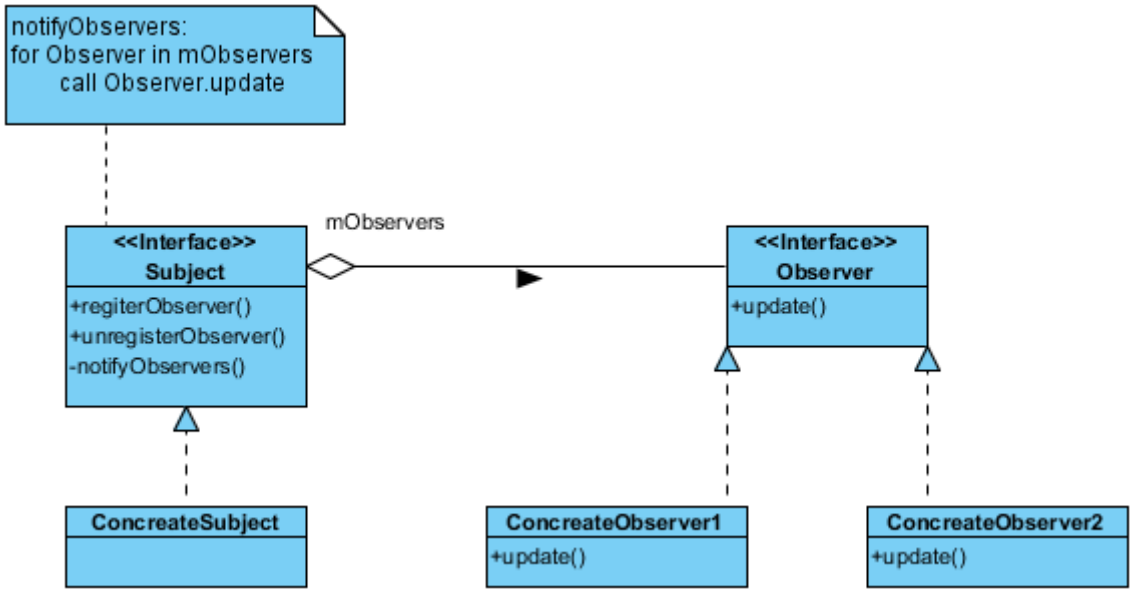
Motiváció

Modell-Nézet-Vezérlő (MVC) vagy Document-View architektúránál gyakran előfordul, hogy a felhasználó megváltoztatja az egyik nézeten az adatokat, ezeket a többi nézeten is frissíteni kell, következetességet fent kell tartani. Egy táblázatkezelő különböző adatmegjelenítéseket használ, megjelenítheti táblázatként és diagramként is az információt. A két megjelenítési objektum nem tud egymásról elég nekik a szükséges objektumok újrahasznosítását elvégezni. Ha felhasználó módosít egy adatot a táblázatban, akkor ezzel egy időben a diagram is módosul és fordítva is. Ezt nem célszerű szoros kötéssel megoldani, mert akkor csökken az osztályok újra felhasználásának lehetőségeinek száma. Erre jó megoldást nyújt a Megfigyelő (Observer) minta. Megadja a szükséges kapcsolatok megvalósításának módját. Főobjektumai az Alany (Subject) és a Megfigyelő felület (Observer), közöttük 1-több kapcsolat áll fent.

Alkalmazhatóság

Olyan esetekben, amikor egy fogalomhoz két olyan interfésszel rendelkező objektum tartozik, hogy az egyik függ a másiktól. Amennyiben külön objektumba zárjuk ezeket a jelentéseket, úgy egymástól függetlenül módosíthatjuk, vagy hasznosíthatjuk újra őket. Amikor egy objektum megváltoztatása további objektumokon való operációt vonja maga után, viszont ezen további objektumok száma ismeretlen számunkra. Amikor változásokról kell értesíteni más objektumokat, azonban nincs ismeretünk arról, melyek ezek az objektumok.

Felépítés



Résztvevők

1. Alany (Subject) Tárolja a Megfigyelőket. Interfészt biztosít a Megfigyelő objektumok csatolására és leválasztására.

2. Megfigyelő felület (Observer) Frissítő interfészt határoz meg az értesítendő objektumok számára. Frissítő (update) művelet.

3. KonkrétAlany osztály (ConcreteSubject) Érdekes állapotokat tárol a Megfigyelő felület számára és saját állapotváltozásairól értesíti őket

Együttműködés
: Az Alany (Subject) értesíti a Megfigyelő felület (Observer) - t, az olyan változásokról, ami különböző állapotot eredményez, mint a sajátja. Miután értesül az Alany (Subject) - ban bekövetkező módosulásokról + információkat kérhet, és összehangolhatja működését.

Következmények
: Alkalmazásának előnyei és hátrányai: Laza kapcsolat van az Alany (Subject) és Megfigyelő (Observer) között, ami azt jelenti, hogy az Alanynak (Subject) tudomása van a Megfigyelőkről (Observer), de pontos információja nincs róla. Üzenetszórás támogatás, értesítésnek nem kötelező definiálnia a fogadóját. A modell újrafelhasználható! Új osztályokkal egyszerűen bővíthető a struktúra a modell és nézet osztály módosítása nélkül. Az Alany (Subject) objektum változáskor informál több más objektumot úgy, hogy lenne róluk bár mi információja is. Előfordulhatnak szükségtelen, váratlan frissítések a nem precízen definiált függőségi feltételek következtében.

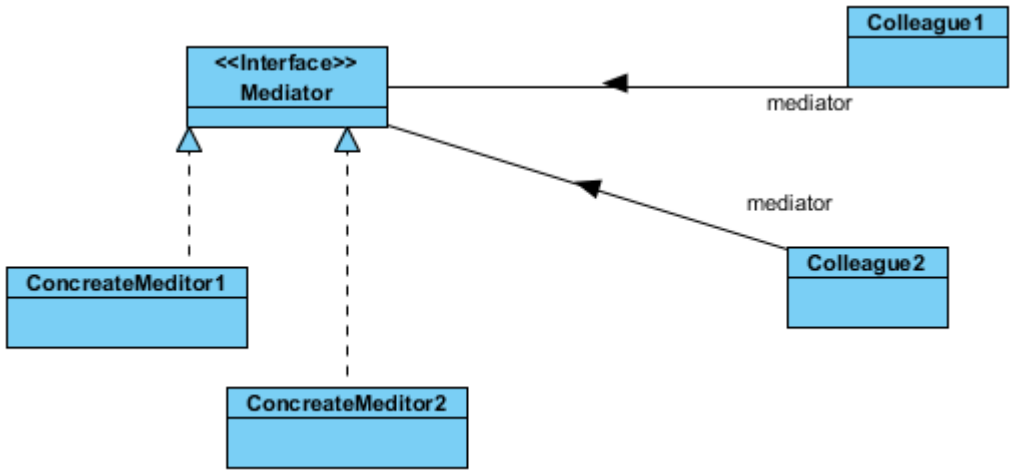
4.3. Közvetítő (Mediator)

Cél
: A minta célja definiálni egy olyan közvetítő (mediator) objektumot, ami egymásra ható objektumhalmazok közötti interakciók irányítását végzi. Segítségével laza kapcsolat építhető fel azáltal, hogy az objektumok között nem történik egymásra való direkt hivatkozás. Ez lehetőséget ad az objektumok közötti kapcsolatok független alakítására.

Motiváció
: Az objektum orientált tervezés a viselkedések objektumok közötti szétosztására biztat, melynek eredményeképpen bonyolult, rengeteg kapcsolatot tartalmazó objektum szerkezet alakulhat ki. Legrosszabb esetben a rendszer összes objektumának tudomása van egymásról. Ha egy rendszert objektumokra bontunk fel az általában fokozza az újra felhasználhatóságot, viszont ha túl sok az objektumok közötti kapcsolat az már csökkenti azt. A rengeteg kapcsolat valószínűtlenné teszi annak a lehetőségét, hogy egy objektum önállóan működhessen, más objektumok támogatása nélkül, emiatt a rendszer viselkedése olyan, mintha tömbszerű volna. Sőt mi több a rendszer viselkedésének módosítása is bonyolult lesz bármilyen úton, mivel a viselkedés számos objektum között van elosztva. Végül pedig a tervező kénytelen lesz számos alosztályt létrehozni a rendszer viselkedésének testre szabásához.

Alkalmazhatóság
: Akkor érdemes használni, ha olyan objektum halmazunk van, mely egyértelműen meghatározott, viszont egymással összetett módon kommunikálnak. A kialakuló kapcsolat rendszer szervezetlen és nehezen érthető. Ha egy objektum újrafelhasználhatósága nehézzé válik a sok más objektumra való hivatkozás és azokkal való kommunikáció miatt. Amennyiben egy viselkedést, mely számos osztály között osztottunk szét szeretnénk testre szabni anélkül, hogy sok alosztályt hoznánk létre.

Felépítés



Résztvevők

1. Közvetítő (Mediator) A Kolléga (Colleague) elemhez definiál egy kommunikációs interfészt.

2. KonkrétKözvetítő (Concrete Mediator) Az együtt működő objektumokat irányítja. Társairól tud és ezekkel a kollégákkal kapcsolatot tart fent.

3. Kolléga (Colleague) osztályok Minden Kolléga (Colleague) elemnek csak a Közvetítőjéről (Mediator) van tudomása. A Közvetítő (Mediator) elemnek központi elosztó szerepe van, ezen keresztül fut át minden interakció, kérés a Kollégák (Colleague) között.

Együttműködés
A Kolléga (Colleague) elemek a Közvetítőnek (Mediator) küldik a kéréseiket és az arra kapott válaszokat is tőle fogadják.

Következmények

1. Korlátozza az alosztályok számát: A közvetítő lokalizálja a viselkedési módokat, melyeket máskülönben számos osztály között kellene szétosztani. Amennyiben szeretnénk ezt a viselkedést módosítani úgy csak a Közvetítő elemből kell képeznünk alosztályokat, a Kolléga elemek felhasználhatósága így megmarad.

2. Megszünteti a Kollégák közötti szoros kapcsolatot: A közvetítő minta segítségével laza kapcsolat alakítható ki az objektumok között, ami lehetőséget ad arra, hogy a Kolléga (Colleague) és a Közvetítő (Mediator) elemeket egymástól függetlenül is felhasználhassuk vagy alakítsuk.

3. Leegyszerűsíti az objektum protokollok: A Közvetítő a több-több kapcsolatok helyett a közvetítő és kollégák közötti egy-több kapcsolatokat teremt, melyek könnyebben érthetőek, kezelhetőek és kiterjeszthetőek.

4. Az objektumok együttműködésének elvonatkoztatása: Közvetítő minta alkalmazása lehetőséget ad arra, hogy az objektumok saját viselkedése helyett az elemek közötti interakciókra helyezzük a hangsúlyt. Ezzel tisztább képet kaphatunk a rendszer objektumai közötti kölcsönhatásokról.

5. A vezérlés központosítása: A Közvetítő minta a kölcsönhatások bonyolultságát bár leegyszerűsíti, viszont a Közvetítő (Mediator) elem felépítése bonyolulttá válik. Mivel egy közvetítő protokollokat ágyaz be, így az sokkal összetettebbé válhat, mint bármelyik egyedi kolléga. Emiatt a közvetítő elem olyan nagyméretű egységgé válhat, aminek a kezelése igen nehéz.

4.4. Bejáró (Iterator)

Cél
Aggregált (összetett) objektumok elemeinek szekvenciális elérésének biztosítása, anélkül hogy az objektum belső tárolási formáját felfednénk.

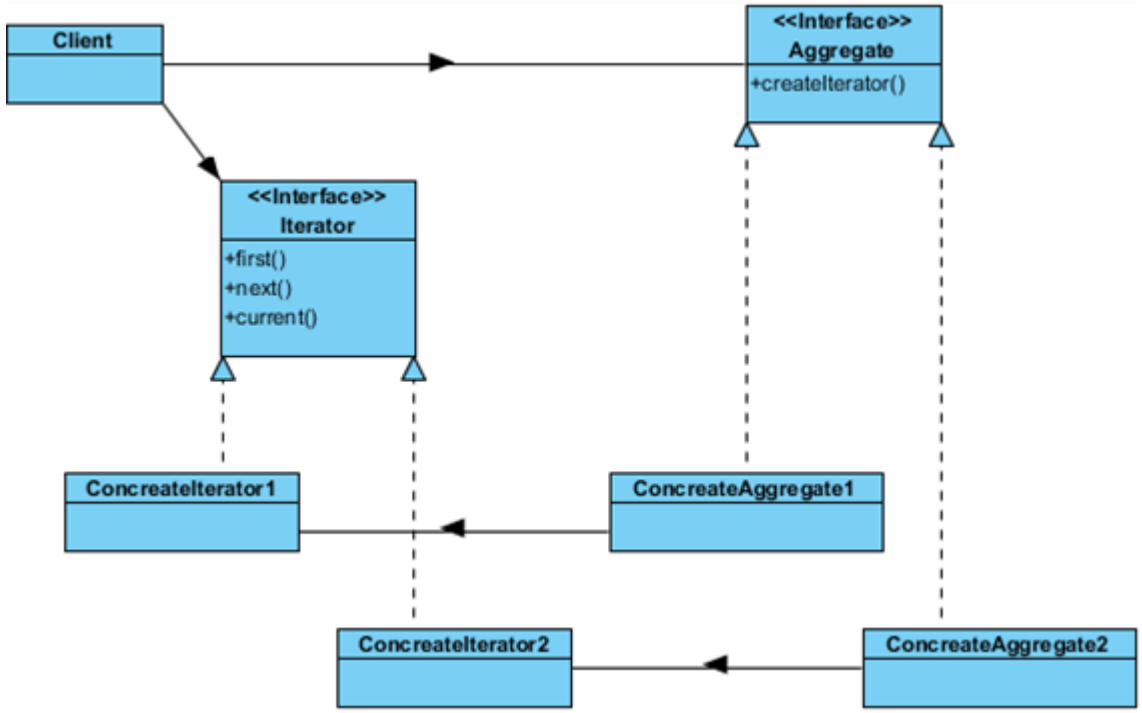
Motiváció

Egy összetett objektumnak, mint például egy lista, módot kell adnia arra, hogy az elemeit anélkül érhessük el, hogy annak belső szerkezetét ismernénk.

Alkalmazhatóság

Összetett objektum tartalmához való hozzáférés a belső szerkezet feltárása nélkül. Többféle bejárás biztosítása összetett objektumok elemeihez. Egy egységes interfészt akarunk nyújtani egymástól eltérő aggregált szerkezetek bejárásához. (azaz a bejárás többalakúságát támogatjuk) .

Felépítés



Résztvevők

1. Bejáró (Iterator) az elemek hozzáféréséhez és bejárásához definiál egy interfészt.

2. KonkrétBejáró (ConcreteIterator) a Bejáró (Iterator) interfészt valósítja meg. Nyomon követi az aktuális elem helyét az Összesítő (Aggregate) bejárása közben.

3. Összesítő (Aggregate) a Bejáró (Iterator) objektum kreálásához definiál interfészt.

4. KonkrétÖsszesítő (ConcreteAggregate) A Bejáró (Iterator) elem létrehozását biztosító interfészt implementálja a megfelelő KonkrétBejáró (ConcreteIterator)példány visszatéréséhez.

Együttműködés

A KonkrétBejáró (ConcreteIterator) nyomon követi, hogy a bejárandó objektum mely eleménél járunk és ki tudja számítani, hogy melyik az objektum soron következő eleme.

Következmények

1. Változatokat támogat egy összetett objektum bejárásához Komplex aggregált objektumokat számos módon lehet bejárni. A bejárók lehetőséget adnak arra, hogy egyszerűen változtathassuk a bejárást megvalósító algoritmust: mivelhogy csak az bejáró példányát kell kicserélni egy másikra. Ezen felül definiálhatunk Bejáró alosztályokat is az új bejárási algoritmusok támogatásához.

2. Az iterátorok egyszerűsítik az Összesítő interfészét Az iterátor bejárási interfésze megelőzi egy hasonló Összesítő interfész alkalmazását, ezáltal leegyszerűsíti azt

3. Egyidőben több bejárási folyamat is működhet egy aggregátumon Egyszerre több bejárást is folytathatunk, mivel egy iterátor nyomon követi a saját bejárásának állapotát.

4.5. Felelősséglánc (Chain of Responsibility)

Cél

Az üzenet vagy kérés küldőjének függetlenítése a fogadótól. Megvalósítása felelősségláncok kialakításával történik. A Felelősséglánc (Chain of Responsibility) nem más, mint láncolt lista, amin a kérelem végig halad mindaddig, amíg egy objektum le nem tudja kezelni. A kérelem a láncon meg is szakítható.

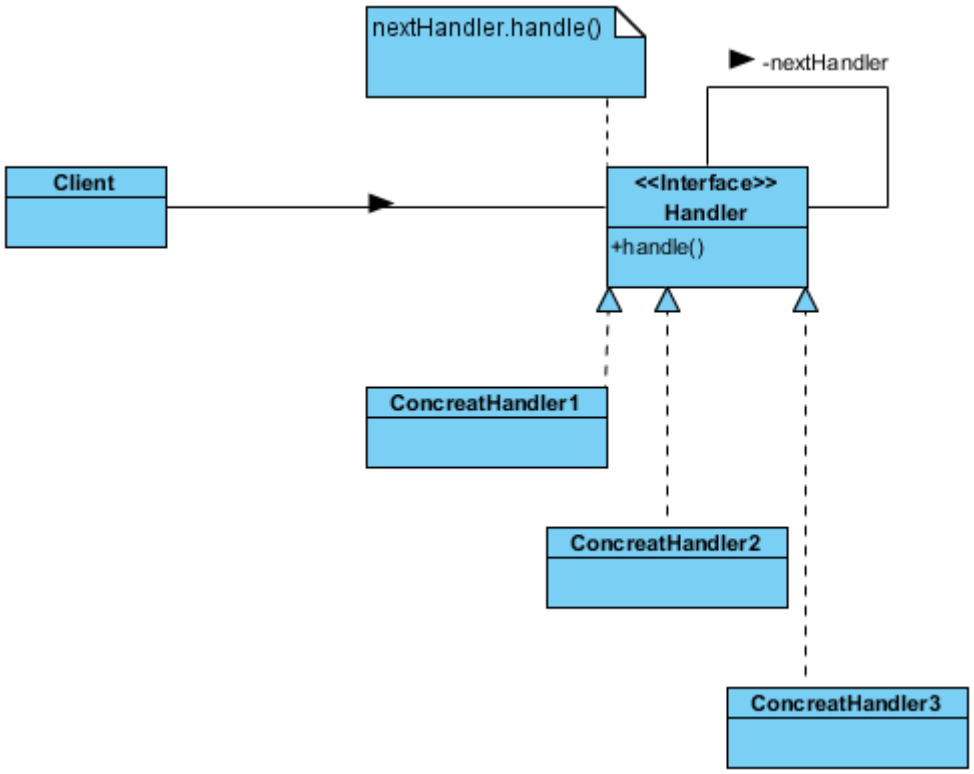
Motiváció

Vegyünk egy grafikus felhasználói felület, amihez környezettől függő help rendszert akarunk csinálni. A felhasználó a felület bármely részére kattint, kapjon help információt. A help nem csak attól függ, hogy a felhasználó a felület mely részére kattint, hanem a kiválasztott felületelem környezetétől is. Például egy dialógusablak gombjához más help információt tartalmazhat, mint a főablak hasonló gombjához. Ha nincs a kijelölt felületelemnek saját help értesítése, akkor a help rendszer egy általánosabb help értesítést szemléltet a felületelem közvetlen környezetéről. (Például a dialógusablakról.) Egyértelmű, hogy az értesítések rendezettséget mutatnak a legspeciálisabbtól a legáltalánosabbig. Azonfelül az is magától értetődő, hogy a help értesítést a felhasználói interfész egyik objektuma kezeli le. Környezettől és a rendelkezésre álló help jellegzetességétől függ, hogy melyik objektum. Ami a gondot okozza, hogy a kiszolgáló objektum, ami kiszolgálja a help kérést nem mindenképpen ismert a kérést elindító objektum számára (nyomó gomb). Vagyis külön kell választani a gombot, ami a kérést elindítja azoktól az objektumoktól, amik a help értesítést biztosítják. A Felelősség lánc (Chain of Responsibility) tervezési minta ezt valósítja meg.

Alkalmazhatóság

Mikor egynél több objektum kezelhet le egy igényt és az igényt lekezelő eleve nem ismert, automatikusan kell kiválasztani, hogy melyik objektum az. Igényünket objektumok együttesének egy objektumnak akarjuk címezni, a fogadó objektum pontos definiálása nélkül. Igényt megvalósító objektumok csoportja dinamikusan választható ki.

Felépítés



Résztvevők

1. Kezelő (Handler) a kérések kezeléséhez definiál egy interfészt. Implementálhatja (Opcionálisan) a következő egyedhez a kapcsolatot is.

2. KonkrétKezelő(ConcreteHandler) Lekezeli azokat a kéréseket, melyekért ő a felelős. Hozzáférhet a láncban az utána következő elemhez. A hozzáérkező kérést vagy lekezeli, ha le tudja, egyébként pedig a láncban továbbítja.

3. Ügyfél (Client) Egy KonkrétKezelő (ConcreteHandler) objektumhoz intéz kérést, ezzel elindítva a kérelmét a láncban.

Együttműködés

Először felépítik a kezelőkből a felelősség láncot. Minden elem (láncszem tudja a feladatát, hogy miért felelős, mit tud kezelni. Ha a láncban elindul egy kérés az addig halad a láncon belül, míg egy olyan KonkrétKezelő (ConcreteHandler) objektumhoz nem ér, aminek hatáskörébe tartozik az adott kérés lekezelése.

Következmények

A minta alkalmazása szükségtelenné teszi, hogy a kérést intéző objektumoknak ismerete legyen a kérésüket lekezelő objektumról. Csupán azt tudják, hogy a kérésüket ki fogják szolgálni. A kérést feldolgozó elemeknek sincs ismerete a küldő kilétéről, illetve a lánc szerkezetéről, amelybe tartoznak. Azáltal, hogy a lánc objektumai csupán csak a következő elemre referálnak, így egyszerűsödik az objektumok közötti kapcsolati háló. A Felelősséglánc viselkedési minta alkalmazása magasabb rugalmassági szintet nyújt az objektumok közötti felelősségek kiosztásában. A lánc futási időben is módosítható vagy kibővíthető az egyes kérések lekezeléséhez. Ezt a módszert kombinálhatjuk azzal, hogy statikusan egyes kérésekre specializált alosztályokat is létrehozunk. Azonban nincs garantálva, hogy a kérések minden esetben teljesítve lesznek. Mivel az egyes kérések nem egy konkrét fogadó objektumhoz futnak be, így nincs rá garancia, hogy a láncon végigfutva az le lesz kezelve. Ezt az is okozhatja, ha a láncot nem helyesen konfigurálják.

15. fejezet - További fejlesztési tevékenységek áttekintése

Valamely szoftver kifejlesztése a korábbi fejezetekben áttekintett nélkülözhetetlen folyamattevékenységek mellett számos további kisebb fejlesztési tevékenységen mehet át. Természetesen e fázisok bizonyos esetekben nélkülözhetők, melyet mindig a fejlesztendő szoftver komplexitása és típusa, valamint a környezet határoz meg.

Az alábbi fejezetben olyan további tevékenységeket tekintünk át, melyek igaz hozzátartoznak a szoftverfejlesztés irodalmához, bár nem szükségszerűek, de a fejlesztés menete megkívánhatja. A nélkülözhetőség kritériuma alól egyetlen egy tevékenység lesz a kivétel, maga a szoftvertesztelés. Ezt a tevékenységet minden szakirodalom szorosan a többi fontos tevékenység mellett szerepelteti, mint ahogy látni fogjuk teljesen jogosan.

Egy szoftver készítése során több különböző fázison megy keresztül. Az egyes fázisok után, de kitüntetetten bizonyos fázisokban elengedhetetlen a tesztelés folyamata. Ennek oka nagyon egyszerű. A fejlesztés folyamata megköveteli az emberi intelligenciát, és mivel az ember könnyen hibázik, szükségszerű, hogy az egyes fázisok elkészülte után valamilyen ellenőrzést hajtsunk végre a terven, kódrészleten, stb. Ennek hiányában a készülendő szoftver valószínűleg tele lesz hibákkal, melyet a megrendelő pedig nem fog elfogadni.

A szakirodalomban számos technika alakult ki a tesztelés folyamatának segítésére. A következőkben általánosan áttekintjük ezt a folyamatot.

1. 15.1 Verifikáció és validáció

A verifikáció és a validációt (V & V) általánosan szoftvervalidációnak nevezik. Legfőbb célja, hogy megmutassa, a rendszer konform a saját specifikációjával, és hogy a rendszer megfelel a rendszert megvásárló ügyfél elvárásainak. Ez olyan ellenőrzési folyamatokat foglal magában, mint például szemléket, felülvizsgálatokat a szoftverfolyamat minden egyes szakaszában a felhasználói követelmények meghatározásától kezdve egészen a program kifejlesztéséig. Röviden tehát:

1. Verifikáció: a terméket jól készítetjük el?

2. Validáció: a megfelelő terméket készítetjük el?

A verifikáció magába foglalja annak ellenőrzését, hogy a szoftver megfelel-e a specifikációnak, azaz eleget tesz-e a funkcionális és nem funkcionális követelményeknek. Általánosabban: a szoftver megfelel-e a vásárló elvárásainak.

A validáció ennél kicsit általánosabb fogalom. Végcélja az, hogy megbizonyosodjunk arról, hogy a szoftverrendszer „megfelel-e a célnak”. Azaz teljesül-e a vásárló elvárása, amibe beleértendő olyan nem funkcionális tulajdonságok is, mint a hatékonyság, hibatűrés, erőforrásigény.

A V & V két, egymást kiegészítő különböző perspektíva segítségével végzi az ellenőrzési folyamatot:

1. Statikus: szoftverátvizsgálások. Olyan technikák, melyek kimondottan csak a rendszer reprezentációját elemzik. Ilyen a követelmény dokumentum, a tervek, és forráskódok.

2. Dinamikus: a klasszikus értelemben vett szoftvertesztelés. Csakis az implementáció fázisában végezhető el. Valamely tesztadatok segítségével ellenőrzi, hogy a rendszer adott inputra megfelelő outputot nyújt-e.

Az átvizsgálási technikák közé tartoznak a programátvizsgálások, az automatizált forráskód elemzés és formális verifikáció. A rendszert csak akkor tesztelhetjük, ha elkészült egy végrehajtható változatának prototípusa. Az inkrementális fejlesztés előnye, hogy a különböző inkremensek külön tesztelhetők, és az egyes funkciók már hamarabb tesztelhetők.

Ne felejtsük el, hogy a tesztelési folyamat önmagában véve nagyon általános. A klasszikus értelemben vett „futtatom a programot és megvizsgálom jó lett-e az eredmény” csak egy részét képezi a teljes folyamatnak. A

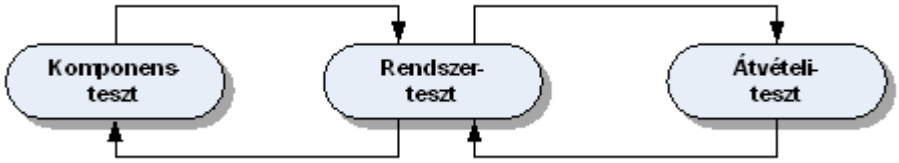
szakirodalmi tapasztalatok szerint az átvizsgálási folyamatot a szoftverfolyamat minden olyan lépésében használható és alkalmazni is kell, ahol már rendelkezésre áll valamilyen kézzel fogható, olvasható reprezentáció a rendszerről. Ez általában a követelmények feltárása fázisban a követelmények validálásával kezdődik egészen a végleges rendszer leszállításáig. Nyugodtan kijelenthető, hogy gyakorlatilag minden részfolyamat validációja szükséges.

15.1 .1 A tesztelés folyamata általánosan

A kis programok kivételével a rendszerek nem tesztelhetők magukban, mint monolitikus egységek. Egy elkészült rendszer már túl komplex ehhez, ezért fontos, hogy a tesztelési folyamatot a fejlesztési modelltől függően lehetőleg minél kisebb komponensek vagy egységek szintjére szorítsuk le. Ez azt jelenti, hogy minden elkészült komponensnek, inkrementális fejlesztés esetén pedig minden inkremensnek rendelkeznie kell a működésüket igazoló tesztekkel. A fejlesztés során az elkészült részegységek így önmagukban tesztelhetők lesznek, melyek pedig a hibák jobb felderíthetőségéhez vezetnek. Az elkészült komponensek integrálása során új funkciókkal bővül a rendszer. Minden integrációs lépés azonban további teszteket von maga után, nem szabad megelégedni a komponensek külön-külön működő tesztjeivel. Az integráció nem várt hatásokat eredményezhet, melyeket csak szisztematikus teszteléssel deríthetünk fel.

A következő ábra egy háromlépéses tesztelési folyamatot mutat be, ahol teszteljük a rendszer komponenseit, majd az integrált rendszert, és végezetül a teljes rendszert a megrendelő adataival.

15.1. ábra - A tesztelési folyamat



A programban felderített hibákat természetesen ki kell javítani. Ez olyan következménnyel járhat, hogy a tesztelési folyamat egyéb szakaszait is meg kell ismételni. Ha a programkomponensekben található hibák az integrációs tesztelés alatt látnak napvilágot, akkor a folyamat iteratív, a későbbi szakaszokban nyert információk visszacsatolandók a folyamat korábbi szakaszaiba.

A tesztelési folyamat szakaszai:

1. Komponens (vagy egység) tesztelése: az egyedi komponenseket tesztelni kell, és biztosítani kell tökéletes működésüket. Minden egyes komponenst az egyéb rendszerkomponensektől függetlenül kell tesztelni.

2. Rendszer tesztelése: a komponensek integrált egysége alkotja a teljes rendszert. Ez a folyamat az alrendszerek és interfészeik közötti előre nem várt kölcsönhatásokból adódó hibák megtalálásával foglalkozik. Ezen túl érinti a validációt is, vagyis hogy a rendszer eleget tesz-e a rendszer funkcionális és nemfunkcionális követelményeinek és az eredendő rendszertulajdonságoknak.

3. Átvételi tesztelés: ez a tesztelési folyamat legutolsó fázisa a rendszer használata előtt. A rendszert ilyenkor a megrendelő adataival kell tesztelni, amely olyan hiányosságokat vethet fel ami, más esetben nem derül ki.

Az átvételi tesztelést alfa-tesztelésnek is szokták nevezni. A megrendelésre készített rendszerek egy egyedülálló kliensnek készülnek. Az alfa-tesztelési folyamatot addig kell folytatni, amíg a rendszerfejlesztő és a kliens egyet nem ért abban, hogy a leszállított rendszer a rendszerkövetelményeknek megfelelő.

Amikor egy rendszer, mint szoftvertermék piacra kerül, gyakran egy másik tesztelés is végbemegy, amelyet béta-tesztelésnek nevezünk. A béta-tesztelés magában foglalja a rendszer számos potenciális felhasználójához történő leszállítását, akikkel megegyezés történt a rendszer használatára, és ők jelentik a rendszerrel kapcsolatos problémáikat a rendszerfejlesztőknek. Ezáltal a rendszer valódi használatba kerül, és számos olyan hiba válik felfedezhetővé, amelyeket a rendszer építői esetleg nem láthattak előre. Ezután a visszacsatolás után a rendszer módosítható és kiadható további béta-tesztelőknek, vagy akár általános használatra is.

15.1 .2 A dinamikusan tesztelés

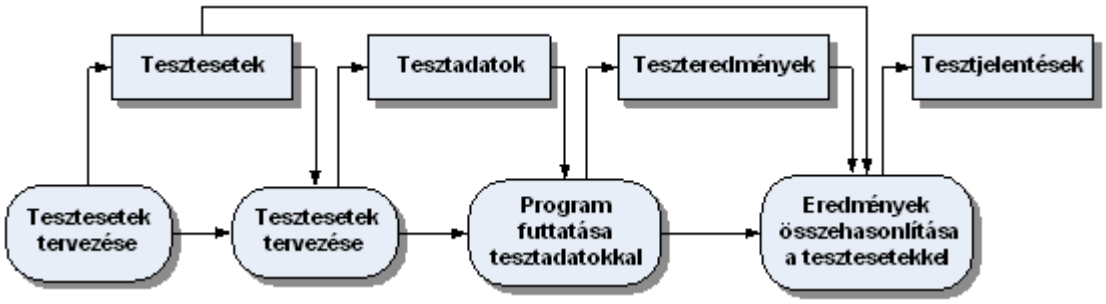
A szoftvertesztelés, mint dinamikus V & V a gyakorlatban inkább alkalmazott technika. Ekkor a programot a valós adatokhoz hasonló adatokkal teszik próbára. A kimenetek eredményei lehetőséget adnak anomáliák, problémák feltárására. Ezen tesztelés két fajtája ismert:

1. Hiányosságtesztelés: célja a program és a specifikációja között meglévő ellentmondások felderítése. Az ilyen teszteket a rendszer hiányosságainak feltárására tervezik, nem a valós működés szimulálására.

2. Statisztikai (vagy validációs) tesztelés: a program teljesítményének és megbízhatóságának tesztelése valós körülményeket szimulálva. Annak megmutatása, hogy a szoftver megfelel-e a vásárlói igényeknek.

A dinamikus tesztelés folyamatának általános modellje látható következő ábrán.

15.2. ábra - A szoftvertesztelési folyamat modellje



A teszteset nem más, mint a teszthez szükséges inputok és a rendszertől várt outputok specifikációja. A tesztadatok kifejezetten a rendszer tesztelésére létrehozott inputok. A tesztadatok néha automatikusan generálhatók, az automatikus teszteset-generálás viszont általában nem lehetséges. A tesztek outputjait csak azok tudják előre megjósolni, akik értik, hogy a rendszernek mit kellene csinálnia.

Beszélhetünk kimerítő tesztelésről is, ahol az összes lehetséges program-végrehajtási szekvenciát teszteljük. A gyakorlatban nem praktikus, ezért a tesztelésnek a lehetséges tesztesetek egy részhalmazán kell alapulnia. Erre irányelveket kell kidolgozni a szervezetnek, nem pedig a fejlesztőcsoportra hagyni.

2. 15.2 Projekt menedzsment áttekintése

A szoftverprojekt-menedzsment fontos része a szoftvertervezésnek. Szakirodalma az elmúlt évtizedekben rengeteget fejlődött, nagyon szerteágazó. Jelen dokumentumban sajnos nincs lehetőségünk ennek teljes áttekintésére így egy általános bemutatásra, néhány fontosabb menedzsmenttevékenységre kell szorítkoznunk.

A szoftvermenedzserek felelőssége megtervezni és ütemezni a fejlesztési projektet. Ők felügyelik a munkát, hogy biztosítsák, az a szükséges szabványok szerint van végrehajtva. Megfigyelik és folyamatosan ellenőrzik, hogy a fejlesztés megfelelő időterv és költségvetés szerint halad. A jó menedzsment persze nem garantálja még a projekt sikerét. Mindazonáltal a rossz menedzsment általában a projekt sikertelenségét okozza. A szoftvert későn szállítják le, a költségek meghaladják az eredetileg tervezett költségeket, és az is előfordulhat, hogy nem fognak megfelelni a követelményeknek.

A szoftverek projektmenedzsmentjét számos tényező tovább bonyolítja. Mégpedig az, hogy egy szoftver tervezése számos tekintetben különbözik minden más egyéb területen végzett tervezéstől. Néhány ilyen különbség:

1. A szoftver nem kézzelfogható: a szoftver megfoghatatlan. Nem látható vagy tapintható. A szoftverprojekt menedzserei nem látják annak haladását. Csak arra támaszkodhatnak, hogy a projekt átvizsgálásához szükséges dokumentációt mások előállítják.

2. Nincsenek szabványos szoftverfolyamatok: a mérnöki tudományágakkal ellentétben a szoftverfolyamat az elmúlt pár évben jelentősen átalakult, fejlődött. Ennek eredménye, hogy szabványos folyamatokat még nem sikerült kialakítani.

3. A nagy szoftverprojektek gyakran több „különálló” projekt: a nagy szoftverprojektek gyakran különböznek minden más korábbi projekttől, ezért több bizonytalanságot rejtenek magunkban, melyekre nehéz előre

felkészülni. Ráadásul a gyors technológiai váltások a számítástechnika és kommunikáció területén a korábbi tapasztalatokat elavulttá teszik.

Nem meglepetés tehát, hogy számos szoftverprojekt késik, túllépi a költségvetést és a határidőket. A szoftverrendszerek gyakran újak és technikailag innovatívak. Az innovatív tervezői projektek szintén gyakran küszködnek határidőkből fakadó problémákkal. Látva az érintett nehézségeket, talán figyelemre méltó, hogy léteznek projektek, melyek a megadott időn és költségvetésen belül elkészülnek.

2.1. 15.2.1 A projektek tervezése

Egy projekt sikeres előrehaladása nagymértékben függ a folyamatok előre való megtervezésétől. Nagyon fontos, hogy a vezetők előre átgondolják az egyes fejlesztési fázisokat, a fázisok buktatóit, a felmerülő problémákat. Fel kell készülni és hozzávetőlegesen megoldásokat kell előkészíteni az adott problémákra. Mindezen feladatok a projektvezető személyére hárulnak, aki ezekre reagálva gondos tervezés útján elkészíti a projekt tervét. Ezt a tervet úgy érdemes használni, mint a projekt egyik irányvonalát. Ennek a kezdeti tervnek a lehető legjobb tervnek kell lennie, amely a rendelkezésre álló információkból adható. Természetesen ez fokozatosan továbbfejlődik, ahogy a projekt előrehalad és több, jobb információ válik elérhetővé.

A menedzsernek különböző terveket kell felvázolnia. Ezek rövid összefoglalása a következő:

1. Minőségi terv: meghatározza azokat a minőségi eljárásokat és szabványokat, amelyeket a projektben használni kell.

2. Validációs terv: meghatározza a megközelítési módot, az erőforrásokat és az ütemterveket, melyeket a rendszer validációjához használni kell.

3. Konfigurációkezelési terv: meghatározza a konfigurációkezelési eljárásokat és szerkezeteket, amelyeket alkalmazni kell.

4. Karbantartási terv: előre megadja a rendszer karbantartási követelményeit, a karbantartási költségeket és a szükséges erőfeszítéseket.

5. Munkaerő-fejlesztési terv: meghatározza, hogyan kell a projektcsapat tagjainak szaktudását és tapasztalatait fejleszteni.

A tervezés iteratív folyamat. A projekt előrehaladása során újabb és újabb információk látnak napvilágot a fejlesztéssel kapcsolatban. Amin ezen információi elérhetővé válnak, a tervet rendszeresen át kell vizsgálni és szükség esetén módosítani azt.

A tervezési folyamat a projektet meghatározó megszorítások összegzésével indul (megkívánt szállítási határidő, rendelkezésre álló munkaerő, teljes költségvetés stb). Ezekkel a projekt olyan paramétereinek becslésével együtt kell foglalkozni, mint például annak szerkezete, mérete, funkcióinak eloszlása. Az előrehaladás mérföldköveit és a részeredményeket csak ezután határozzuk meg. Ezek után a folyamat belép egy ciklusba. A projekt ütemtervét fel kell vázolni és az ütemterv által meghatározott tevékenységeket el kell indítani, vagy engedélyezni kell azok folytatását. Bizonyos idő után (általában 2-3 hét) a folyamat átvizsgálandó és az eltérések feljegyzendők. Mivel a projekt paramétereinek becslése csak hozzávetőleges, a tervnek mindig szüksége van a módosításokra.

A projektmenedzsereknek felül kell vizsgálniuk a projekttel kapcsolatos feltételezéseiket, ahogy egyre több információ áll rendelkezésükre. Újra kell tervezniük a projekt ütemtervét. Ha a projekt késik, újra kell tárgyalniuk a megrendelővel a projekt megszorításait és a leszállítandó részeket. Ha ez sikertelen és az ütemterv nem tartható, technikai felülvizsgálatra van szükség. Ennek a felülvizsgálatnak az a célja, hogy valamilyen alternatív megközelítési módot találjon a fejlesztéshez, amely a projekt megszorításaiba még belefér és megfelel az ütemtervnek.

2.1.1. 15.2.2 A projektterv

A projektterv lényegében egy induló dokumentum, megpróbálja összefogni a projekthez szükséges minden összetevőt. Ilyen például a rendelkezésre álló erőforrások, a munka felosztása és a munka végrehajtásának ütemterve. A részletezettségi szintje mindig az adott fejlesztési projekthez igazodik és a szervezetek típusától

függően változik. Ennek ellenére mindig érdemes egy olyan ajánlást tenni, amely egy javasolt mintaként szolgál a dokumentum készítése során.

1. Bevezetés. Legfőbb célja a projekt céljainak tömör leírása, és azon megszorítások felsorolása (például költségvetés, idő stb.), amelyek befolyásolják a projekt menedzselését.

2. Projektszervezet. A projektben résztvevő személyeket és szerepüket tisztázza.

3. Kockázatelemzés. A projekttel szemben felmerülő lehetséges kockázatai tényezőket tárgyalja. A bekövetkezésük valószínűségét és a kockázat csökkentésének ajánlott stratégiáját.

4. Hardver- és szoftvererőforrás-követelmények. A fejlesztéshez szükséges hardver és szoftver komponensek összegyűjtése. Vásárlás esetén meg kell becsülni az árakat és a beszerzés határidejét.

5. Projekt költségterv. A szoftver megvalósításának követelményei és a rendelkezésre álló erőforrások alapján a projekt megvalósításának pénzügyi realizációját fekteti le.

6. Munka felosztása. Megadja a projekt tevékenységekre történő felosztását, és azonosítja a tevékenységekhez kapcsolódó mérföldköveket és részeredményeket.

7. Projekt ütemterve. Meghatározza a tevékenységek közötti függőségeket, megbecsüli a mérföldkövek eléréséhez szükséges időket, és javaslatot tesz a tevékenységekhez rendelendő emberekre.

8. Figyelési és jelentéskészítési mechanizmusok. Meghatározza a menedzser által előállítandó jelentéseket, azok előállításának idejét és a használandó projektfigyelési, monitorozási technikát.

A projekttervet rendszeresen át kell vizsgálni a projekt ideje alatt. Az olyan részek, mint például a projekt ütemezése, rendszeresen meg fognak változni, míg egyéb részek pedig stabilan megmaradnak. Olyan dokumentumszervezést érdemes használni, mely lehetővé teszi az egyszerű cseréket, bővítéseket a dokumentumban.

2.1.2. 15.2.3 Mérföldkövek

A menedzsereknek folyamatosan információra van szükségük a projekt előrehaladásával kapcsolatban. Mivel a szoftver kézzel nem fogható dolog, és az információk csak dokumentum formájában biztosíthatók, szükség van olyan ellenőrző pontokra, melyek újragondolási, értékelési lehetőséget biztosítanak a fejlesztés során. Ezeket a pontokat mérföldköveknek nevezik.

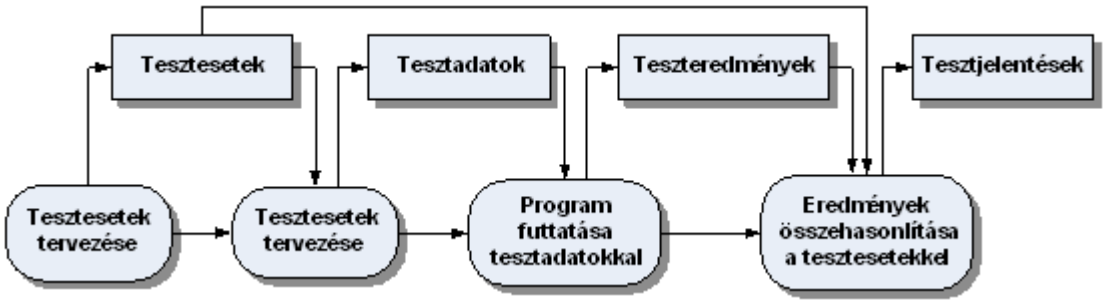
Amikor egy projektet tervezünk, számos mérföldkövet kell előre meghatároznunk, ahol minden egyes mérföldkő egy szoftverfolyamat-tevékenység végpontja. Célszerű a mérföldkövek elérésekor jelentést készíteni, amely bemutatható a vezetésnek. Ezen jelentések nem szükségszerűen nagy dokumentumok. Akár a projekt tevékenységei teljesítésének egyszerű rövid leírása is lehet. A mérföldköveket úgy érdemes reprezentálni, mint a projekt logikai szakaszainak végeit.

2.2. 15.2.4 A projekt ütemezése

A projekt ütemezése a projektmenedzserek kitűntetett feladata. A menedzserek megbecsülik a tevékenységek elvégzéséhez szükséges időt és erőforrásokat, és egy összefüggő szekvenciába rendezik őket. Hacsak a projekt ütemezése nem egyezik meg teljes mértékben az előző projekttel, annak becslései bizonytalan alapjai az új projekt ütemezésének.

Ha a projekt technikailag fejlettebb, a kezdeti becslések majdnem mindig optimistábbak lesznek, még akkor is, ha a menedzserek megpróbálnak minden eshetőséget figyelembe venni. Az ütemterveket folyamatosan frissíteni kell, ahogy egyre több információ áll rendelkezésünkre.

15.3. ábra - A projekt ütemezési folyamata



A projekt ütemezése magában foglalja a projekt teljes munkájának különálló tevékenységekre bontását és a tevékenységek elvégzéséhez szükséges idő megítélését is. Általában ezek közül a tevékenységek közül néhány párhuzamosan is elvégezhető. A projekt ütemezőinek kell koordinálniuk ezeket a párhuzamos tevékenységeket, és úgy megszervezniük a munkát, hogy a munkaerő kihasználtsága optimális legyen. El kell kerülniük az olyan szituációkat, ahol a teljes projekt azért csúszik, mert egy kritikus feladat még nincs befejezve.

A projekttevékenységek normális esetben legalább egy hétig tartanak. A finomabb felosztás azt jelenti, hogy aránytalanul sok időt kell szánni a becslésre és a felülvizsgálatra. Szintén hasznos, ha felső időkorlátot szabunk azokra a tevékenységekre, amelyek 8-10 hétig tartanak. Ha ezt túllépik, akkor ajánlatos a tervet és az ütemezést részekre bontani.

Az ütemtervek megállapításakor a menedzsereknek nem ajánlatos azt feltételezniük, hogy a projekt minden egyes szakasza problémamentes lesz. A projekten dolgozó egyének megbetegedhetnek vagy eltávozhatnak, a hardverek meghibásodhatnak, vagy az elengedhetetlen szoftvereket és hardvereket késve szállítják. Ha a projekt még új és technikailag is újszerű, bizonyos részeiről kiderülhetnek, hogy nehezebbek és hosszabb ideig tartanak, mint azt gondoltuk.

A projekt ütemtervét legtöbbször diagramok és a tevékenységhálók, grafikus jelölések, halmazaként reprezentálják, amelyek a munka felosztását, a tevékenységek függőségeit, a munkaerők elhelyezkedését stb. szemléltetik. A diagramok előállításának automatizálására gyakran használnak szoftvermenedzselési eszközöket, mint például a Microsoft Projectet.

3. 15.3 Konfigurációkezelés

Egy szoftver fejlesztése, használata során a környezet, és így a követelmények egy része folyamatosan változhat. A változás új követelményeket, új konfigurációkat idézhet elő, melyeket be kell építeni a rendszer új verzióiba. A verziók tartalmazzák a változtatások terveit, a hibák javítását és a különböző hardverekhez és operációs rendszerekhez való illesztéseket. Sőt, egyszerre több verzió fejlesztése is történhet párhuzamosan.

A rendszer fejlődését valamilyen módon kezelni kell, mert könnyű eltéveszteni, hogy melyik változás a rendszer melyik verziójába került be. A konfigurációkezelés fogalom tehát eljárások és szabványok fejlesztését és alkalmazását takarja a változó rendszertermékek kezelésénél.

A konfigurációkezelést általában a rendszerminőség kezelése részének tekintik, melyek szabványt is biztosítanak a konfigurációkezelésre. Ilyen szabványra példa az IEEE 828-1998, amely a konfigurációkezelés tervezésének szabványát definiálja.

A konfigurációkezelés tervezése

A többi szoftverfejlesztési tevékenységhez hasonlóan a konfigurációkezelést is tervezni kell. A terv leírja azokat a szabványokat és folyamatokat, amelyeket a konfigurációkezelés során alkalmazunk. A terv fejlesztésének kiindulópontja néhány általános, vállalati szintű konfigurációkezelési szabvány, és szükség szerint ezeket kell alkalmazni minden egyes projektre. A terv fontos része a felelősségek definiálása is. Vázolni kell, hogy ki a felelős minden egyes dokumentum és szoftverkomponens továbbításáért a minőségbiztosítás és a konfigurációkezelés felé. Le kell írni a konfigurációkezelési politikákat, amelyek a változtatásvezérlés és a konfigurációkezelés során követendők, a konfigurációkezelés során használt eszközöket, és végül a konfigurációs adatbázis szerkezetét.

3.1. 15.3.1 Verziókezelés

A verziókezelés során a rendszer különböző verzióit kell nyomon követnünk valamilyen előre meghatározott módon. Fontos, hogy a rendszer új verzióját a konfigurációkezelő személyzet állítsa elő, nem pedig a rendszerfejlesztők. Ez könnyebbé teszi a konfigurációs adatbázis konzisztenciájának fenntartását, mivel csak a konfigurációkezelő személyzet változtathatja meg a verzióinformációkat.

Mit is nevezünk verziónak? A rendszer verziója kifejezés gyakorlatilag a rendszer egy példányát jelenti, amely valamilyen módon különbözik a rendszer többi példányától. A különbség jelentkezhet teljesítményben, funkciókban, hibákban, stb. Amennyiben csak kis különbségek vannak a verziók között, akkor az egyiket néha a másik variánsának is nevezik.

A verziók menedzselését manapság már CASE-eszközök támogatásával végzik, melyek különböző hatékonyságnövelő szolgáltatásokat nyújtanak. Tárolják az összes rendszerverziót, és vezérlik a rendszer komponenseinek elérését. Az eszközök alapvetően különböznek a biztosított eszközökben és a felhasználói felületben, de az itt áttekintett általános verziókezelési alapelvek alkotják az alapját minden támogatóeszköznek.

3.1.1. 15.3.2.1 Verzióazonosítás

Egy rendszer egy önálló verziójának létrehozásához meg kell adni a tartalmazott rendszerkomponens-verziókat. Egy nagy szoftverrendszer több száz szoftverkomponensből állhat, amelyek mindegyikének több verziója is létezhet. A verziókezelés eljárásainak olyan módszert kell használnunk, amellyel egyértelműen azonosítható minden komponens minden verziója. A komponensek speciális verzióit visszaállíthatjuk, ha további változtatásokat akarunk rajtuk végrehajtani.

A gyakorlatban a komponensek verzióinak azonosítására a következő három eljárás alakult ki:

Verziószámozás: a komponenshez egy explicit, egyedi számot rendelünk. Ez a leggyakrabban használt azonosítási technika. Az egyszerű verziószámozási sémában a komponensek és a rendszer nevét egy verziószámmal egészítjük ki. Az első verziót nevezhetjük 1.0-nak, az alverziók az 1.1,1.2 és így tovább. Például: OpenGL 1.5 elnevezés az OpenGL grafikus könyvtár 1.5-ös verziójára utal. Ez a séma egyszerű, de a kapcsolódó információk megfelelő kezelését igényli, hogy nyomon tudjuk követni a verziók közti különbségeket. A név nem mond semmit a verzióról vagy arról, hogy miért jött létre. Emiatt a konfigurációs adatbázisban tárolni kell olyan rekordot, amely leír minden verziót és azt is, miért jött az létre.

Attribútum alapú azonosítás: a megoldás azon az elgondoláson alapul, hogy minden komponensnek van egy neve, amely nem egyedi a különböző verziókat tekintve. A komponensekhez egy attribútumhalmaz is társul, amely a komponens minden verziójában különböző. A komponenseket így a nevük és az attribútumhalmaz együttese azonosítja. Ilyen azonosító attribútumok lehet a vevő, a fejlesztőnyelv, a fejlesztési állapot, a hardverplatform, a készítés dátuma.

Lehetséges, hogy az attribútum alapú azonosítás közvetlenül a verziókezelő rendszerbe van implementálva, és a komponens-attribútumokat a rendszer adatbázisa kezeli.

Változtatásorientált azonosítás: a módszer alapja az attribútum alapú azonosítás. Azonban egy verzió visszakereséséhez ekkor ismerni kell a szükséges attribútumok értékeit. A változtatásorientált azonosítást inkább használjuk a rendszerek verzióira, mintsem a komponensekére. E szerint nevezzük el a komponenseket, valamint ehhez még társítunk egy vagy több változtatási kérelmet. Ez feltételezi, hogy a komponens minden verziója egy vagy több változtatáskérés alapján jött létre. A rendszerverziót így a név és a komponensben implementált változtatás összekapcsolásával azonosítjuk.

A változások kezelését napjainkban számos CASE eszköz segíti. Egy komplex rendszer esetén nem is lehetne valamilyen segítő szoftver eszköz segítsége nélkül hatékonyan menedzselni a projekt változásai és előrehaladását. Ezek az eszközök részben automatizáltan képesek elvégezni és megoldani számos kritikus problémát. Használhatóságuk ma már olyan szintre fejlődött, hogy betanulásukhoz szinte minimális időráfordítás szükséges. Néhány példa verziókövetést támogató CASE eszközökre:

1. Microsoft Project: Projektmenedzselő rendszer

2. SVN, Mercurial, GIT,CVS: open source verziókövető rendszerek

3. MS SourceSafe: Microsoft saját verziókövető rendszere

4. StarTeam: Borland verziókövető menedzsere

5. ClearCase: napjaink legnépszerűbb fizetős verziókövető rendszere

4. 15.4 Ellenőrző kérdések

1. Röviden vázolja mit értünk a verifikáció és a validáció fogalmakon.

2. Miért fontos az átvételi tesztelés a tesztelési folyamatban?

3. Mit értünk béta-tesztelés alatt?

4. Ismertesse mit értünk a teszteset fogalma alatt? Lehetséges-e automatikus generálása?

5. Miben különbözik egy szoftverprojekt tervezése más egyéb területek projektjeitől?

6. Melyek azok a fontosabb tervek, melyeket a tervezés során célszerű elkészíteni, mérlegelni?

7. Mit értünk a mérföldkő fogalma alatt?

8. Milyen eszközökkel reprezentálják egy-egy projekt ütemtervét?

9. Mit értünk verziószámozás alatt?

10. Soroljon fel néhány ma ismert és használt verziókövető rendszert.

16. fejezet - Esettanulmány

1. Bevezetés

Az alkalmazás fejlesztésére nincs tökéletes megoldás, csak javaslatok és ajánlások vannak. A fejlesztés valójában a programozói, rendszerfejlesztési tapasztalatunk használata a szükséges ismeretek birtokában. Ebben a fejezetben egy valós fejlesztés mentét mutatjuk be. Az MTA SZTAKI Kognitív informatika laborja tűzte ki magának a feladatot, hogy egy virtuális együttműködési rendszert hozzon létre, melynek neve VIRCA (VIRtual Collaboration Arena). Az alkalmazás egy 3 dimenziós térben valós és virtuális objektumokat jelenít meg, és kezel. A rendszerben tetszőleges számú és típusú komponenseket lehet elhelyezni (virtuális vagy valóságos), amelyek mindegyike egy önálló alkalmazás. Az első verzió után újabb igények merültek fel. Az elkészült rendszer egy osztott alkalmazás lett, ami CORBA és ICE technológiákat használ. A rendszer egy közös adatstruktúrát használ az elérhető komponensek nyilvántartására, a CORBA naming service-t (név szolgáltatást), ami fa struktúrában tárolja a komponenseket. A 3 dimenziós megjelenítő ablak OGRE (nyílt forráskódú grafikai motor) rendszer segítségével implementálták c++-ban. A mi rendszerünk a hálózati kommunikáció lebonyolítására egy Openrtm-aist robot middleware-t használ. Írtunk ehhez pár kiterjesztést (ICE port, 1 fogyasztó több szolgáltató lehetőség), és a projekt kezdetekor a gyári rendszer szerkesztőt használtuk. A gyári szerkesztő egy Eclipse plugin, amely használatához egy több 10 megabájtos alkalmazás szükséges (Eclipse), és mi rendszerünk használatakor kényelmetlen egy külső szerkesztőt használni (váltogatva az aktív alkalmazást).

A fejlesztés a klasszikus vízesés modell lépéseiből áll, a módszerek is azt javasolják, csak a legtöbb több iterációt ír elő, és így a rendszer finomodik. Érdemes már az elején definiálni egy szótárt, amit a felmerülő újabb és újabb fogalmakkal bővítünk. Általános elv minden fázisnál, hogy először egy durva megközelítés után egyre részletesebb leírást, modellt készítünk.

2. Követelmények felderítése

Minden projekt egy ötlettel / fő igénnyel kezdődik. Ebben az esetben ez a következő volt: Kellene egy olyan rendszer szerkesztő a VIRCA rendszerünkhöz,

- amely a rendszerünk OGRE 3D virtuális terében is használható,
- támogatja az Openrtm-aist rendszert (hiszen a mi rendszerünk azon alapszik),
- támogatja továbbá az általunk korábban elkészített kiterjesztést.

Nevezzük ezeket fő követelményeknek. Ezt már most érezni lehet, hogy ezek vizsgálata újabb követelményeket fognak felvetni. Ezeket sorra megvizsgáltuk, majd pontosítottuk, részleteztük. Nézzük ezeket sorban!

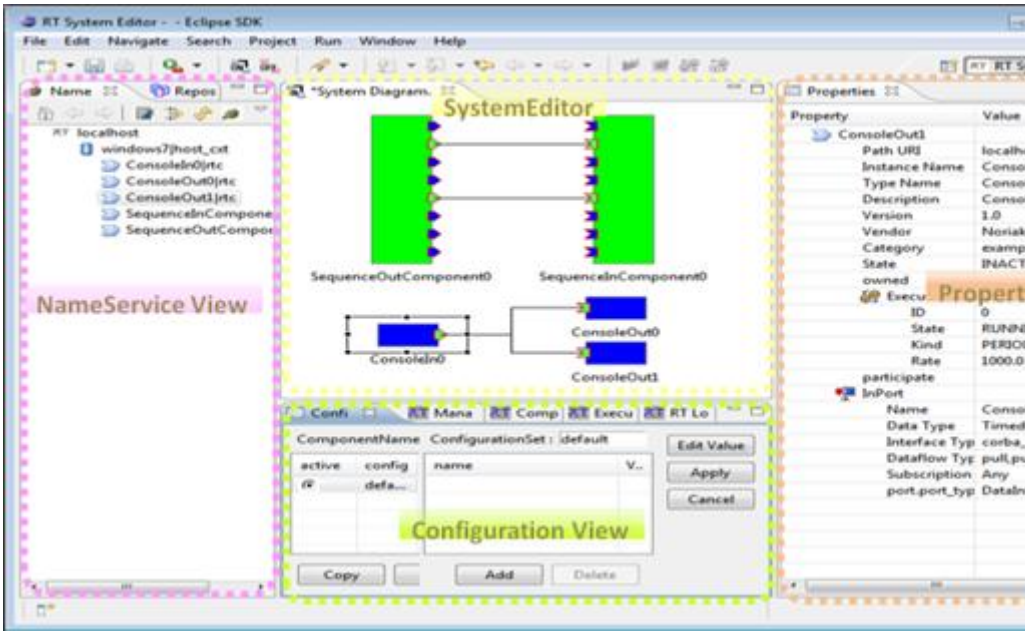
OGRE 3D virtuális terében használható. Az OGRE C++-ban implementált SDK, amit használhatunk 3 dimenziós modellek létrehozására mozgatására. Egyik lehetőség az volna, hogy mi felépítünk egy 3 dimenziós menüt, és szerkesztő felületet. Az Openrtm-t Japánban fejlesztették ki és ott sok helyen használják. Ha OGRE alatt írjuk meg, akkor a menürendszer változása miatt folyamatosan újabb és újabb verziót kell fordítanunk és letölteni a felhasználóknak. Ez nagy függőséget jelentett volna. Fejlesztés során törekedni kell az általános megoldásokra, a könnyen frissen tartható rendszer komponensekre. Szerencsére akkoriban adták ki a Chrome böngésző memóriába dolgozó verzióját, amit rögtön implementáltak OGRE alá Berkelium néven. A böngészőben működő szerkesztő lényegesen leegyszerűsíti a fejlesztés menetét és az előbb említett problémákra is megoldást kínál. (A felhasználó minden használatkor a legfrissebb verziót használja.). Első fontos döntésünk tehát, hogy egy webalkalmazást (vagy legalábbis böngészőben működő) készítünk, így a felhasználók köre bővülhet, hiszen olyanok is használhatják, akik az alap Openrtm rendszert használják. Mivel ez egy grafikus szerkesztő, ezért valamilyen kliens oldali technológiát kellett használnunk. Erre három lehetőség kínálkozik:

- Java FX
- Flash/Flex
- Silverlight

A Microsoft által fejlesztett Silverligth technológiát azonnal elvetettük, mert nem létezik Linux és Mac OS rendszerek alá lejátszó plugin a böngészők alá. A Java FX és Flash támogatottság elég széleskörű, ezért azok teljesítményét, erőforrás igényét, programozhatóságát, használhatóságát vizsgáltuk meg. Sajnos a Berkelium alatt nem működött a Java FX lejátszó, így az akkori körülmények miatt maradt a flash. (Az összehasonlítás egyik fontos szempontja a minimális munka befektetése. Ezen okból a Java FX jobb lett volna, hiszen ha a szerver oldal jsp-ben íródik, akkor számos osztály közös lehetett volna.) Mivel a Flash rendszerben nem létezik CORBA és ICE könyvtár, így csak vékony kliensről beszélhetünk, azaz a kliens oldalon csak a komponensek és azok kapcsolata jelenik meg, a tényleges aktiválás összekötés a szerver oldali logika feladata.

Támogatja az Openrtm-aist rendszert. Nyilván ha használni, csatlakozni, vezérelni szeretnénk egy rendszert, akkor ismernünk kell annak minden részletét. A rendszer felépítését és működését a Robot Technology Component (RTC) specifikáció rögzíti, amely letölthető az OMG weboldaláról. Tulajdonképpen minden olyan funkciót támogatnia kell, mint amit az alap rendszer szerkesztője támogat. Ezt viszonylag könnyű feltérképezni, hiszen a gyári szerkesztő letölthető és kipróbálható. Jellegében is hasonlítania kell az eredetihez, melyet a következő ábra mutat:

16.1. ábra - Az openrtm-aist rtcse rendszer szerkesztőjének képernyője



Az ábra felső részén látható a fő ablak, amiben balra az elérhető komponensek, középen a szerkesztő terület, balra pedig a tulajdonság ablak látható (tartalma a kiválasztott elem függvénye). Törekednünk kell arra, hogy a fontos funkciókat implementáljuk először (sőt ezen belül a meghatározókat és amelyik kevés ráfordítást igényel). A szerkesztő funkciói a következők:

- komponensek listázása (kapcsolódás a CORBA név szolgáltatóhoz),
- komponens adatainak a megjelenítése (portjainak elhelyezése a komponens körül),
- komponensek állapotának jelzése,
- komponensek vezérlése (aktivizálás, resetelés),
- portok összekötése (paraméterek megadása),

Támogatja az általunk korábban elkészített kiterjesztést. Kifejlesztett kiterjesztésünknek egyik nagy momentuma az ICE technológia támogatása lett. Az alap rendszerben használható volt egy

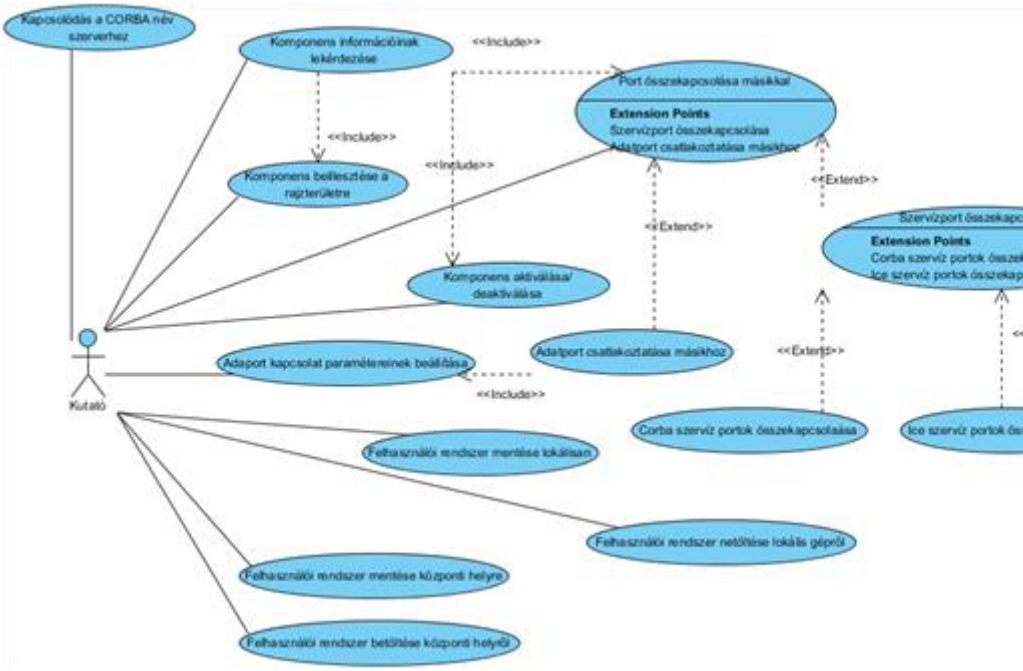
- adatport (beépített típusok továbbítására és fogadására)
- szervizport CORBA alapú(funkcionalitás, metódus távoli hívására)

A létező CORBA szerviz port használata (CORBA interfészek, osztályok készítése) nehézkes és nagy gyakorlatot igényel. Ezért fejlesztettük ki az ICE szerviz portot, amelyet sokkal egyszerűbb használni és megérteni. Másik nagy újdonság, hogy egy fogyasztó típusú szerviz porthoz (ICE) több szolgáltató port is kapcsolódhat, ami a VIRCA rendszerben szükséges is. Ezeket a kiterjesztéseket azonban a hivatalos szerkesztő nem érti, ezért kell beépíteni az új szerkesztőbe.

2.1. Funkcionális követelmények felderítése

Az fejlesztés következő lépése a funkcionális követelmények feltárása, erre remek módszer az UML használati eset diagramjának elkészítése. Az ábráról gyorsan és egyértelműen leolvashatóak a kapcsolatok, azonban szöveges leírás szükséges a modell elemek leírására. Egyes UML szerkesztők a diagramok és a modell alapján képesek dokumentációt generálni, ami igen hasznos, sok munkát takarít meg. A következő ábrán láthatjuk a rendszer használati eseteit:

16.2. ábra - A szerkesztő használati esetei



Ennek a létrehozása több lépésben szokott elkészülni, az nem baj, ha elsőre nem jut eszünkbe minden funkció (a módszerek is csak ajánlások tesznek mikor legyen kész a teljes funkciólista bizonyos része). A diagram pont arra jó, hogy a megrendelő, vagy kollégánk a diagram alapján megérti elképzelésünket és ki tudja egészíteni a listánkat. A diagram egyszerűsége miatt a megrendelő is tudja használni azt. A dokumentációnak egységesnek kell lennie minden azonos típusú elemre. A használati esetekről tudnunk kell, hogy mennyire fontos, milyen előfeltételei, utóhatásai vannak. Érdemes a használati eseteket összefoglalni, azaz definiálni a teljes listát, amit a következő ábra foglal össze:

16.1. táblázat - Használati eset diagram elemeinek összefoglaló táblázata

Rövid név	Megnevezés	Dokumentáció
A1	Kutató	Személyek egy csoportja, akik grafikus felületen használják a rendszert. Nincs programozói ismeretük, azaz nem biztos, hogy ők fejlesztették ki a komponenst.
HE1	Elérhető komponensek lekérdezése a CORBA nameserver-től	A szerkesztőnek fa struktúrában meg kell jelenítenie az elérhető komponenseket. Minden komponens regisztrálja magát egy corba name szerveren. A frissítés automatikus is és a felhasználó kérésére explicit is végre kell hajtani.
HE2	Komponens információk	A felhasználó lekérdezheti a komponensek különböző paramétereit:

Rövid név	Megnevezés	Dokumentáció
	lekérdezése	• tulajdonságait (készítő, típus, implementációs nyelv) • futási környezet paramétereit • státuszát
HE3	Komponens beillesztése a robot rendszerbe	A felhasználó grafikus felületen keresztül beilleszt egy komponenst a szerkesztő területre. A beillesztett komponensek főbb paramétereinek már lent kell lennie, pl.: Név, készítő, portok nevei típusai, ...
HE4	Port csatlakoztatása egy másik porthoz	Komponensek közötti kapcsolatot a rendszerben azok portjaival lehet létrehozni. Az alap rendszer nem minden esetben teszi lehetővé a több-több kapcsolatot, sőt sok esetben a gyári szerkesztő grafikája is szétesik, de ez a szerkesztő támogatja azt.
HE5	Komponens vezérlése	Komponensek vezérlése a kliensből elengedhetetlen. Aktivizálás / deaktivizálás, vagy reszetelés.
HE6	Adatport kapcsolat paramétereinek beállítása	Adatportok esetén a kapcsolat profil megadása döntően befolyásolja az adat mozgását, ezért elengedhetetlen ezen funkció implementálása.
HE7	Felhasználói rendszer mentése lokálisan	A felhasználó munkáját lementheti lokálisan. Ilyenkor a homokzsákban futó flash alkalmazás kell, hogy hozzáférjen a mentett rendszerhez. Legyen lehetőség egy létező mentett rendszer felülírására.
HE8	Felhasználói rendszer betöltése lokális gépről	A felhasználó munkáját betöltheti lokálisan. Ilyenkor a homokzsákban futó flash alkalmazás kell, hogy hozzáférjen a mentett rendszerhez. Legyen lehetőség a betöltés után ellenőrizni a komponensek elérhetőségét.
HE9	Felhasználói rendszer mentése központi helyre	A felhasználó munkáját lementheti központi helyre. Ilyenkor a homokzsákban futó flash alkalmazás kell, hogy hozzáférjen a mentett rendszerhez. Legyen lehetőség egy létező mentett rendszer felülírására.
HE10	Felhasználói rendszer betöltése központi helyről	A felhasználó munkáját betöltheti központi helyről. Ilyenkor a homokzsákban futó flash alkalmazás kell, hogy hozzáférjen a mentett rendszerhez. Legyen lehetőség a betöltés után ellenőrizni a komponensek elérhetőségét.
HE11	Adatport csatlakoztatása másikhoz	Komponensek alapvető kommunikációjának eszköze az adatport. Itt előre definiált típusú információk mozognak. A kommunikáció egyirányú.
HE12	Szervizport összekapcsolása	Az openrtm-aist rendszerben a komponens egy fekete doboz. Ha valamely másik komponens szeretné használni szolgáltatásait, akkor azt csak előre definiált interfészeken keresztül teheti. Ennek egyetlen módja a szerviz portok definiálása. Ha egy komponens fogyasztani akarja másik komponens által nyújtott szolgáltatást, akkor össze kell kötni azok szerviz portjait.
HE13	Corba szerviz portok összekapcsolása	Az openrtm-aist rendszer tartalmaz CORBA szerviz portokat. Ennek támogatása elengedhetetlen.
HE14	Ice szerviz portok összekapcsolása	A SZTAKI által kifejlesztett új szerviz típus az ICE szerviz port. Ennek támogatása fontos feladat.

Nézzünk meg példaként egy két használati eset részletes leírását:

16.2. táblázat - "Elérhető komponensek lekérdezése a CORBA nameservertől" használati eset részletei

Megnevezés	Érték

Megnevezés	Érték
Szint	Felhasználó
Bonyolultság	Alacsony
Prioritás	Magas
Előfeltételek	Futó CORBA name server, futó alkalmazás szerver.
Utóhatások	Az elérhető komponensek eltárolódnak a modellben, a nézetben a felhasználó láthatja a komponenseket.

16.3. táblázat - "Port csatlakoztatása egy másik porthoz" használati eset részletei

Megnevezés	Érték
Szint	Felhasználó
Bonyolultság	Közepes
Prioritás	Magas
Előfeltételek	1. A komponens portjának már ismertnek kell lennie: típus, név (azaz már előzőleg le kell kérdezni a komponenstől) . 2. Azonos típusúak kell legyenek.
Utóhatások	A komponens adatokat fogadhat/küldhet vagy szolgáltatást használhat/nyújthat.

Ha több aktor van, akkor érdemes szerepeltetni azt is egy plusz oszlopban. Szerepelhet még az alternatív működés is.

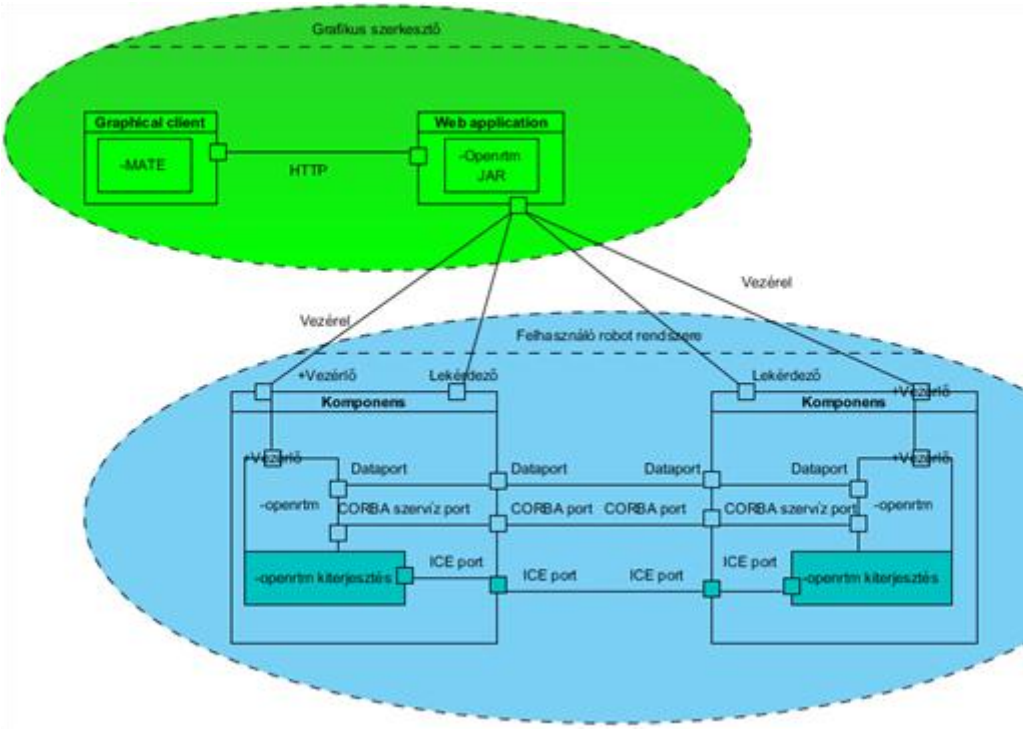
2.2. Nem funkcionális követelmények felderítése

Ehhez a lépéshez az UML nem ad segítséget, a legtöbb módszer szöveges dokumentum létrehozását javasolja. A SysML azonban lehetőség ad követelmény diagram formájában. Ez a diagram tartalmazhatja a használati eseteket is, azonban vigyázzunk arra, hogy ha változtatjuk, akkor mind a két helyen változtassuk. Nagy előnye a diagramnak, hogy a követelmények közötti tartalmazás, pontosítás kapcsolatokat is ábrázolja, sőt a követelményeket ellenőrző teszt eseteket! Itt is a fontos követelményekkel , követelmény kategóriákkal kezdünk, majd törekedünk a teljességre.

3. A rendszer elemeinek, struktúrájának azonosítása

Itt is a rendszer fő részeit határozzuk meg először, majd tovább bontjuk azokat. A létező rendszer kell először modellezni. Ezt a telepítés, üzemeltetés során figyelhetjük meg. Az együttműködő rendszerünket a következő ábra mutatja:

16.3. ábra - A szerkesztő használati esetei



Először a világoskék részeket határozzuk meg, ezek az alap rendszer részei, aztán jön a kiterjesztés (türkiz kék), végül a zöld rész jön, ami mostani projektünk részleteit mutatja. A részek dokumentációja a következő:

16.4. táblázat - A rendszer struktúrájának részletei

Megnevezés	Dokumentáció
Grafikus szerkesztő	Az új szerkesztő szoros együttműködésen alapszik. Az alapvető műveleteket a webalkalmazás végzi. A grafikus szerkesztő csak kéri azokat a webalkalmazástól. Egy webalkalmazáshoz általában több böngészőben futó flash alkalmazás fut.
Grafikus kliens	FLEX-ben írt grafikus kliens. Biztosítja egy felhasználó barát használatát a rendszernek. Böngészőben fut (biztosítva a platform függetlenséget), csak egy flash lejátszó szükséges a használatához.
Web alkalmazás	JSP-ben implementált szerver logika. A flash klienstől (de akár más klienstől is, amely tud hhtp protokoll felett kéréseket küldeni). Több szervleten keresztül érhetjük el a rendszer szolgáltatásait: • menedzselhetjük a felhasználót, rendszert, • komponensek információit lekérdezhetjük, vagy vezérelhetjük azokat, • portokat köthetünk össze, vagy kacsolhatunk szét
MATE	A Mate egy tag-alapú, eseményvezérelt Flex keretrendszer (framework). Arra szolgál, hogy megkönnyítse a Flex alkalmazások eseménykezelését. Biztosítja, hogy a esemény térképeket definiáljunk, és az alrendszerek teljesen függetlenek legyenek egymástól.
Openrtm	Gyári openrtm rész, alapszolgáltatásokat implementálja.
http port	Az alapértelmezett 80 tcp port, ami a kommunikációhoz kell.
Felhasználó Robot Rendszere	A kiterjesztett (iceport) felhasználói rendszer, amely a szerkesztő beavatkozása segítségével jön létre, indul el, de önállóan működik. Több komponens (fekete doboz) lehet részese, amelyek közti kapcsolatot a portok szolgáltatják.
Komponens	A felhasználó futtatja valamelyik támogatott operációs rendszeren. A rendszer egy komponense. Különböző nyelven implementált (c++, java).
Vezérlő	A komponensek vezérlését teszi lehetővé CORBA távoli objektum.

Megnevezés	Dokumentáció
Openrtm JAR	Gyári openrtm rész, alapszolgáltatásokat implementálja. Openrtm-aist keretrendszer java implementációja. Két jar file. Ez biztosítja az alap rendszer használatát.
Dataport	Típusos adatport, több alaptípusa van. a kapcsolat profil döntően meghatározza működését.
CORBA szerviz port	CORBA szerviz port, amely lehet • szolgáltató vagy • fogyasztó Támogatja az • egy szolgáltató egy fogyasztó, • egy szolgáltató több fogyasztó kapcsolatokat.
openrtm kiterjesztés	SZTAKI által korábban kifejlesztett iceport kiterjesztés. Segítségével bármely komponensnek lehet ice szerviz portja.
ICE szerviz port	CORBA szerviz port, amely lehet • szolgáltató vagy • fogyasztó Támogatja az • egy szolgáltató egy fogyasztó, • egy szolgáltató több fogyasztó, • több szolgáltató egy fogyasztó kapcsolatokat.

4. A rendszer statikus modelljének kidolgozása

A rendszert először egészében próbáljuk meg ábrázolni. Gyakori a réteges felépítés, mert a külvilág és más rendszerek számára logikát egységesen kell szolgáltatni. A statikus modell felépítésének általános menete a következő:

1. **Alapvető osztályok azonosítása**: kezdjük a használati eset diagramban szereplő aktorokkal, azok minden esetben osztályokká alakíthatóak. Ezután a megjelenítéshez, kommunikációhoz szükséges osztályok következnek.

2. **Alapvető kapcsolatok azonosítása** : A rendszer osztályai között levö kapcsolatok felderítése a következő lépés. Először a tartalmazás, aztán az ismeretség következik (ha tudjuk az irányt, akkor jelöljük azt is).

3. **Interfészek azonosítása**: Programozzunk interfészt! Egy jó technika, ha az osztályok nem osztályként használják a másik példányt, hanem csak interfészként. A megfelelő osztályok implementálják azokat. Így közben ki tudjuk majd cserélni az osztályokat is úgy, hogy a használó osztály nem veszi észre.

4. **Kapcsolatok pontosítása**: Valamennyi eddig felfedezett kapcsolatot megvizsgálunk, nevesítjük, megállapítjuk a szerepköröket és számosságokat. Ne feledjük, hogy a kapcsolatok pontosítása új osztályok definiálásához is vezethet. Vázoljuk azokat a meggondolásokat, amelyek az osztálydiagram pontosításához vezetnek, és leírjuk, hogy mit változtat az eddigi információkon.

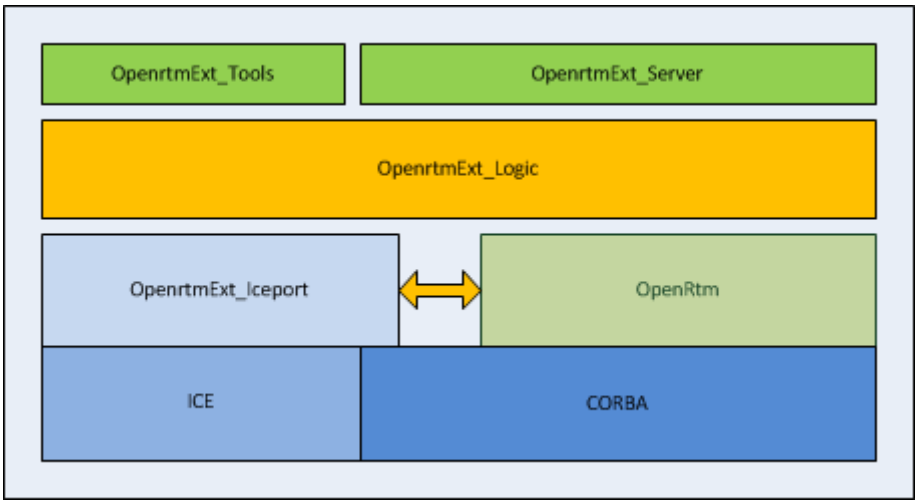
5. **Attribútumok azonosítása**: Minden osztályhoz (vagy kapcsolathoz, ha ahhoz tartozik) hozzárendeljük a legfontosabb attribútumait. Ez az elemzés is vezethet új osztályok felfedezéséhez. Vázoljuk azokat a

meggondolásokat, amelyek az osztálydiagram pontosításához vezetnek, és leírjuk, hogy mit változtat az eddigi információkon.

6. **Bázisosztályok keresése**: Megvizsgálandó, hogy vannak-e olyan esetek, ahol az egyes osztályokban ismétlődő argumentumok egy közös bázisosztályba emelhetők ki. Ha találunk ilyeneket, az általánosítás kapcsolatokkal kiegészítjük az osztálydiagramot. Vázoljuk azokat a meggondolásokat, amelyek az osztálydiagram pontosításához vezetnek, és leírjuk, hogy mit változtat az eddigi információkon. Az így pontosított osztálydiagram egy átmeneti munkaanyag, amely a dokumentum mellékletébe kerülhet (hivatkozással!), de nem kötelező része a dokumentumnak.

Itt is réteges, mert a konzol kötözgető is ugyanazt a logikát kell használja, mint a webes. A rendszer topológiája a következő:

16.4. ábra - A SZTAKI Openrtm kiterjesztésének felépítése



A rendszer 4 rétegből áll, melyek a következők:

- megjelenítési, és vezérlési,
- logika réteg,
- alacsonyszintű réteg.
- programozói middleware réteg

A **megjelenítés és a vezérlés réteg** (zöld színű). Feladatuk a rendszer interfészének biztosítása:

1. paraméterek fogadása, ellenőrzése,
2. esetleges eredmények megjelenítése,
3. Szolgáltatásokat kér a vezérlő rétegtől

A rendszer interfésze webalkalmazás esetében szervletek és jsp oldalak, konzol alkalmazás esetén pedig a futtatható konzol alkalmazások.

Az OpenrtmExt_Logic, **logikai réteg** (sárga színű) a logika réteg, amelynek feladata a megjelenítési réteg által kért funkciók kiszolgálása az OpenrtmExt_Iceport és az OpenRtm alrendszerek segítségével. Ez a réteg egy csomag (jar fájl), amely a web-, vagy konzol alkalmazás mellet kell legyen.

Alacsonyszintű réteg tartalmazza az Openrtm-aist implementációját, ez külső rész, és az OpenrtmExt_Iceport részt. Ezek végzik az alacsony szintű műveleteket. Ezek minden esetben a futó komponensben vannak beágyazva.

Fontos, hogy a verziószámok minden szinten vannak, hiszen bármelyik változása döntően befolyásolja a rendszer működését. A rétegeknek megfelelően 4 alprojektből áll rendszer a rendszer, melyek a következők:

- OpenrtmExt_Control -> jar file
- OpenrtmExt_Iceport -> jar file (ezt már kifejlesztettük)
- OpenrtmExt_Server -> (servletek) war file
- OpenrtmExt_Tools -> jar file (futtatható alkalmazások)

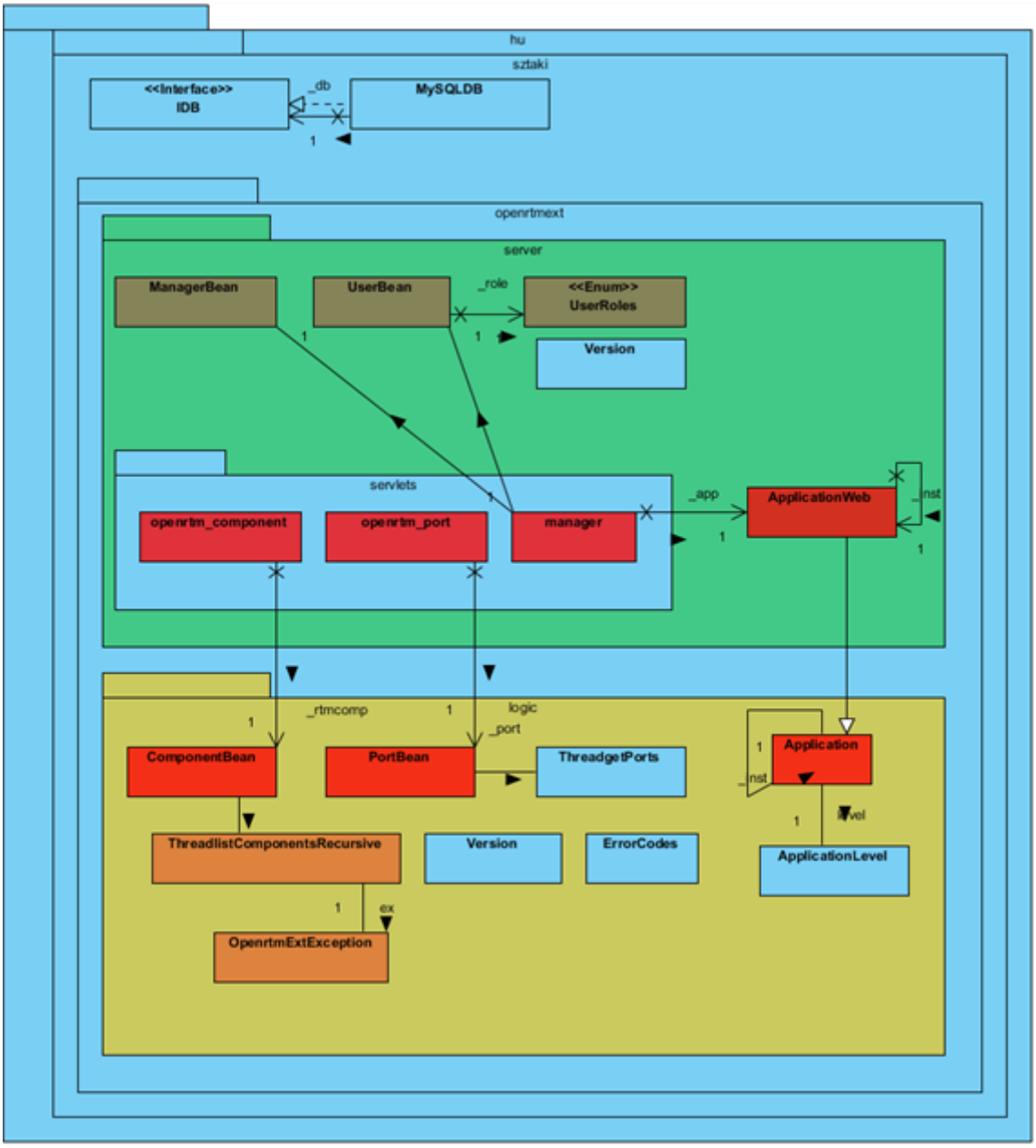
Webalkalmazás esetén a fontos paraméterek beállíthatóak a szervlet jsp motor alatt. Ennek az az előnye, hogy nem kell minden frissítés, újratelepítés után/előtt átírni a paramétereket. Használható a régi metodika is , de nem javasolt. A JNDI lehetőségét kihasználva globális erőforrásokat definiálunk, amelyeket a webalkalmazás lokális névterébe "linkelünk". A rétegek minden esetben egy alrendszert jelentenek, aminek egy a programozói nyelvben a csomag vagy névtér felel meg.

Ezek után azonosítsuk az alrendszerek meghatározó osztályait! Azokkal kell kezdenünk, amelyek az interfészét adják az alrendszernek, és azokat amelyek fogyasztják a másik alrendszert. Első körben csak a fontosakat dolgozzuk ki, majd az azokat közvetlen kiszolgálókat, végül mindet. Érdemes valamilyen felhasználói interfészen keresztül megközelíteni a rendszer feltérképezését. Ebben az esetben a weboldalaknál, szervleteknél kezdjük.

Tipp

Érdemes először a rendszer teljes topológiáját kidolgozni az osztály részleteitől eltekinteni (az osztály maradjon tehát egyszerű téglalap), mert ha minden osztály minden tagját feltüntetjük, akkor nem látjuk át és nem lehet kinyomtatni. Későbbiek során is rejtsük el a részleteket a megjelenítéskor, nyomtatáskor. Ne feledjük a modell és a nézet más. A modellben benne kell, hogy legyen minden részlet (hiszen ebből generáljuk a forráskódot), de a megjelenítést, megértést ne zavarjuk a felesleges részletekkel.

16.5. ábra - A SZTAKI Openrtm kiterjesztésének szerver oldali struktúrája



Az osztályok létrehozásának menete ebben az esetben a következő:

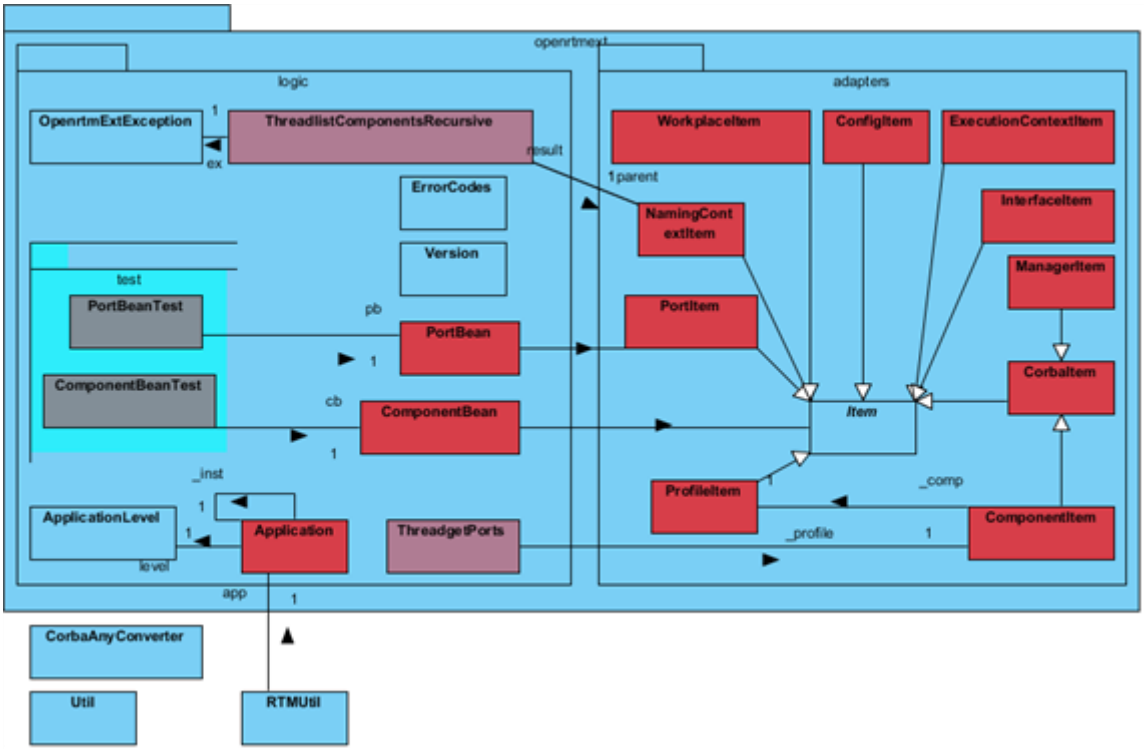
1. Application - ApplicationWeb; openrtm_port - PortBean; openrtm_component - ComponentBean; manager

2. MangerBean, UserBean,UserRoles; ThreadlistComponentsRecursive, OpenrtmExtException

3. végül mindegyiket.

Ezután a rendszer alap funkcióihoz kapcsolódó osztályokat kell részletesen kidolgozni. Ebben az esetben ez a logika alrendszer. Ebben az esetben egy flash alkalmazás kommunikál a szerverrel. Ilyenkor jó megoldás az xml alapú kommunikáció használata. Használhatunk web szervizeket, vagy sorosítást, de a leggyorsabb ha mi írjuk meg szerver oldalon az xml előállítást és a szerver oldalon az xml alapján létrehozzuk a példányokat. Ebben az esetben a kliens és a szerver oldal nyelve más, de a szerepek azonosak. Nehezebb webalkalmazást tervezni, fejleszteni, mint desktop alkalmazást, hiszen minden logika kettéválik és a http protokoll állapotmentessége miatt az objektumainkat valahol tárolni kell, át kell vinni az állapotot a különböző kérések között. Itt még nehezebb a helyzet hiszen a kliens nem a hagyományos vékony kliens (csak böngésző) hiszen a kliens oldalon rajzolni, ellenőrizni, dönteni kell. Újabb osztályokat vezethet be a felhasználó felület tervezése, és az alacsony szintű réteg változása vagy annak illesztése. A logikai rész részletesebb struktúráját a következő ábra szemlélteti:

16.6. ábra - A SZTAKI Openrtm kiterjesztés logikájának struktúrája



Az osztályok létrehozásának sorrendje:

1. Application, ComponentBean, PortBean;Item, PortItem, ComponentItem

2. FonálOsztályok

3. összes többi

Az Item osztály arra jó, hogy rendszer alacsony szintű szereplői képviselje. Lekérdezés esetén a megfelelő Item (megfelelő paraméterrel) létrejön, majd az appendToXmlWriter vagy a toString fv.ét hívva kiírja az állapotát (vagy a kérés válaszába vagy a kimenetre). Itt használhatunk interfészt vagy származtatást is. A lényeg, hogy a metodika ugyanaz. Nem baj, ha nincs meg minden osztály, amikor szükség lesz rá az új osztályra akkor a megfelelő fv. felüldefiniálásával beilleszthetjük. Érdemes már itt a tesztelést beépíteni. A főbb (felette lévő rétegnek szolgáltató) osztályokra unit (JUnit) teszteket írunk, így könnyen tudjuk ellenőrizni a rendszer rétegeit külön-külön, és nem kell a többi réteg a teszteléshez.

Amikor a rendszerük felépítése megvan, kezdhetünk neki a az egyes osztályok részleteinek a kidolgozásának. A tervezés fázis végén az összes (iterációs módszer eseten az adott iterációban kifejlesztendő osztályok) osztály teljes leírását meg kell határoznunk. Itt a cél a teljesség. Még a teszt osztályokat is meg kell terveznünk. A tervezési fázis végén az osztály diagramból generálunk osztályokat, amelyek metódusainak a törzse üres. Az ideális az volna, ha az implementáció során nem kellene újabb adattagot és metódust definiálni, de nagyon nehéz ilyen részletességű tervet készíteni és nagy tapasztalat kell hozzá. Ami azonban nagyon fontos:

- Az osztályok interfésze, publikus metódusainak a készlete nem változhat. Nem csökkenhet, hiszen ha valaki számít arra, és nincs meg, akkor fordítási hibát okoz. Nem bővülhet, mert akkor nem tervezett, ellenőrzött módon (kikerülő megoldásokkal) tudják használni az erőforrásokat. Ha terv során valamely interfész megszületik, az tudatos, alapos munka eredménye. Lehet, hogy jót akarunk, de nem ismerve a teljes rendszer felépítését logikáját, olyan kerülő utakat nyitunk, amely veszélyezteti a biztonságos stabil működést.

- Az interfészt biztosító metódusok aláírása nem változhat, hiszen egy paraméter tudatosan érték vagy referencia szerint adódik át. Információt kap, vagy ad a hívás helyére.

5. A rendszer dinamikus modelljének kidolgozása

A statikus modellel párhuzamosan, vagy egy általános dinamikus modell akár azt megelőzően is készülhet a rendszerről. Ezt mindenképpen a rendszer szolgáltatásainak a leírásával érdemes kezdeni. Vesszük a használati eseteket, és azokat leírjuk aktivitás vagy együttműködési diagrammal (a szekvencia diagram csak a későbbiekenben a részletes leírásra javasolnám, mert ott már kellenek az objektumok is). Egy-egy alpontban az SRS-ből átvett használati esetek szokásos működési eseteire elkészítjük a szükséges diagrammokat. A használati esetre a nevével hivatkozunk. Ebben az alpontokban dokumentáljuk a szükséges állapotdiagramokat is. Az állapotdiagramok ábra aláírásában szerepeljen a vonatkozó osztály neve. Én az aktivitás diagrammot szeretem használni, mert az tulajdonképpen egy kiterjesztett folyamatábra, így mindenki számára könnyen érthető, és megjelenik a párhuzamosság és a döntési pont is.

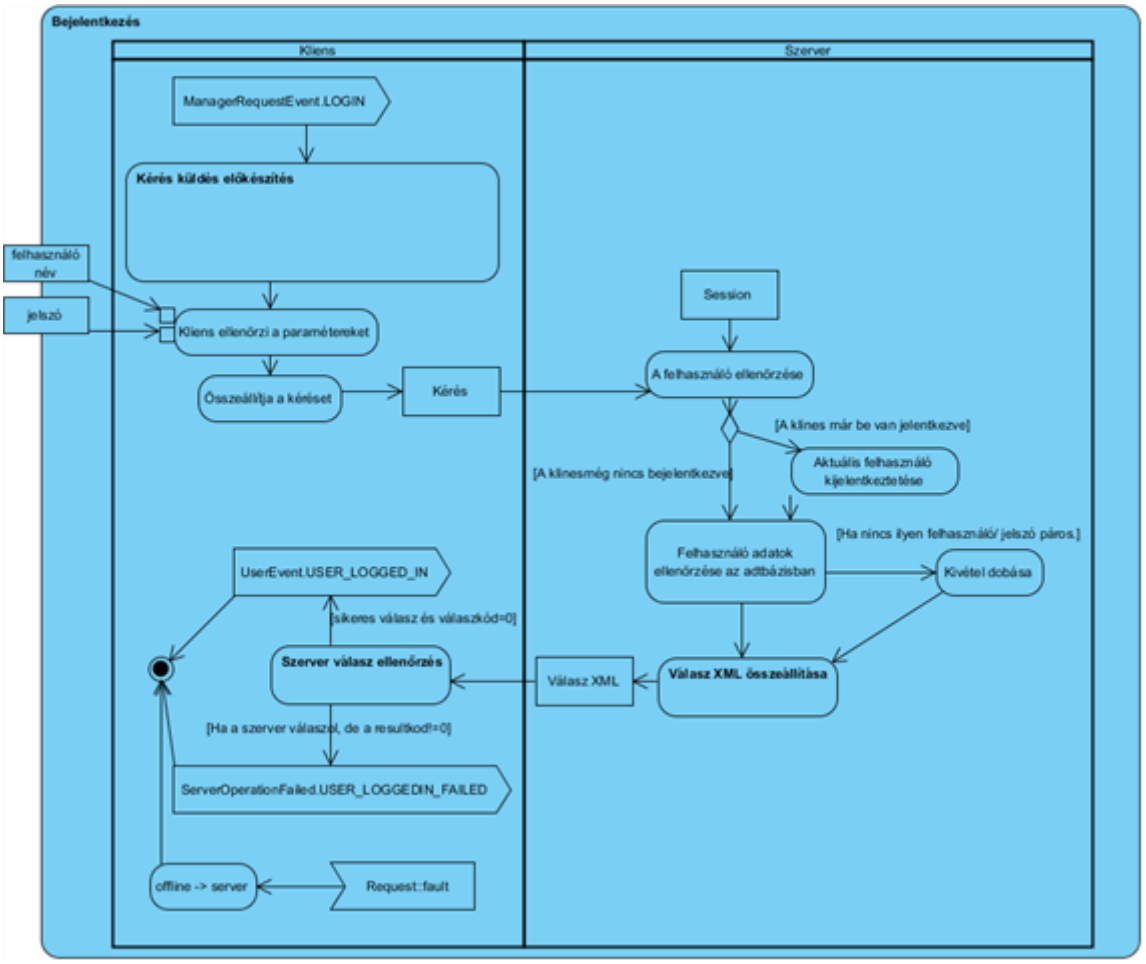
Formája lehet:

- Általános leírás
- UML diagram. Aktivitás diagram esetén a készítés menete:
  1. kezdünk egy aktivitással, amin belül lesznek a résztevékenységek (action)
  2. adjuk meg az aktivitás paramétereit (bemeneti és kimeneti).
  3. résztevékenységek megadása (action). Akciók sorrendjét határozzuk meg, de ha fontos akkor emeljük ki az objektumokat!
- Diagram elemeinek leírása (folyamat részletes leírás). Én ki szoktam venni a közös elemeket. Sok esetben mind a kliens mind a szerver oldalon (webalkalmazás esetében) ugyanazokat lépéseket hajtjuk végre csak esetleg más programozói nyelven. Vannak olyan elemek, amelyek összekötik a két oldalt, ezeket is érdemes itt szerepeltetni. Ha sávos, akkor minden sáv elemeit külön táblázatban érdemes szerepeltetni.

5.1. Bejelentkezés

A felhasználó a kliens grafikus felületén megadja a paramétereit, amit a hálózaton keresztül elküld a szervernek. A szerver ellenőrzi a felhasználó paramétereit, és ha megfelelő, akkor nyit session, amiben eltárolja a felhasználó objektumot. Válaszban elküldi a művelet sikerességét.

16.7. ábra - Felhasználó bejelentkezésének folyamata



Aktivitás paraméterei:

Megnevezés	Felhasználó név
Típus	String
Dokumentáció	Felhasználó azonosítója. Előzőleg regisztrálnia kell, minimum 6 karakter a mérete.
Irány	bemenet
Kötelező	Igen

Megnevezés	Jelszó
Típus	String
Dokumentáció	Felhasználó jelszava. Előzőleg meg kell adni, minimum 6 karakter a mérete.
Irány	bemenet
Kötelező	Igen

Közös elemek:

Megnevezés	Dokumentáció
Kliens	A rendszer flash plugin alatt futó logika része (grafikus).
Szerver	A rendszer szerveren futó logika része.
Kérés	Http kérés post vagy get (tartlama függ a funkkciótól)
Válasz XML	http válasz, melynek tartalma XML

Kliens elemei:

Megnevezés	Dokumentáció
ManagerRequestEvent.LOGIN	Esemény, amely indítja az aktivitást.
Kérés küldés előkészítés	Minden kérés azonos séma alapján épül fel. Egy indikátort frissít a felhasználói felületen. Ha a kérés blokkoló, akkor homokóra kurzor megjelenítése, felületek tiltása.
Kliens ellenőrzi a paramétereket	A bemenő paramétereket ellenőrzi (hossz és tartalom).
Összeállítja a kérést	Több szervlet-et használunk, de csak 1 session-t. Emiatt csak egy kapcsolat tartó objektum lehet, aminek a paramétereit minden kérés előtt változtatjuk.
Szerver válasz ellenőrzés	Kapott XML válasz tartalmának ellenőrzése.
ServerOperationFailed.USER_LOGGEDIN_FAILED	Válasz XML alapján a megfelelő üzenet generálása.
UserEvent.USER_LOGGED_IN	Válasz XML alapján a megfelelő üzenet generálása.

Szerver elemei:

Megnevezés	Dokumentáció
Session	Szerver oldalon tárolt objektum.
A felhasználó ellenőrzése	A felhasználói példány létezésének ellenőrzése a session-ban.
Aktuális felhasználó kijelentkeztetése	Ha a kliens már be van jelentkezve, akkor töröljük a régi objektumát. Valószínűleg egy másik böngészőből jelentkezett be.
Felhasználó adatok ellenőrzése az adatbázisban	Megfelelő táblában olyan rekord keresése, amelyben a mezők megfelelőek.
Kivétel dobása	Hibák esetén kivételek dobása (helyztelen jelszó,

	adatbázis nem érhető el, ...).
Válasz XML összeállítása	A válaszba generáljuk a kérés kiszolgálásnak sikerességét, és ha lekérdező volt, akkor az eredményt is.

5.2. Rendszer osztályainak részletei

Itt minden részletet meg kell határoznunk. A modellben eltárolt információk alapján generál(tat)juk az osztályokat. Vesszük sorra a statikus modell osztályait, és feltöltjük adattagokkal, tagvfüggvényekkel. Sok programozási nyelv támogatja a tulajdonságokat is, amit érdemes használni, mert az osztály használó elemek olvashatóbbak (generált kódjuk általában függvény). Itt is érdemes táblázatos formát használni. Minden osztály egy alpont a fejezeten belül. Minden osztályról a következőket kell megadnunk:

- felelőssége, feladata
- együttműködők
- attribútumok
- operációk

5.2.1. PortBean

Feladata, felelősség: Egy adapter osztály az Openrtm Port osztályához. Feladata a komponens portjaink a vezérlése, adatainak visszaadása.

Eggyütműködők: PortBeanItem

Attributumok:

16.5. táblázat - PortBean adattagok

log : Logger	
Típus	Logger osztály
Hozzáférés	privát
Getter	nincs
Setter	nincs
Statikus	igen
Számosság	1

17. fejezet - Animációk

Evolucios modell animációja

HVR modell animációja

Inkrementális modell animációja

Követelmény dokumentálás animációja

Mérföldkövek animációja

Projekt tervezés animációja

Réteges modell animációja

Teszteklési folyamat animációja

Irodalomjegyzék

Szoftverrendszerek fejlesztése, második kiadás. Ian Sommerville. Panem Kft.. 2007.

Rational Unified Process – áttekintés. Vég Csaba. http://www.logos2000.hu/docs/RUP.pdf. 2001.

Szoftvertechnológia, Egyetemi jegyzet,. Szabolcsi Judit. . 2008.

Az Extreme Programming programozás-technikai elvei.. Juhász Sándor Ferenc. . .

A SpiralModel of Software Development and Enhancement.. Boehm B. ACM SIGSOFT Software Engineering Notes, ACM, 11(4):14-24. 1986.

Szoftverfejlesztési folyamatok: A RUP és a V-modell. Kovács Györgyi, Lakatos Emőke, Nagy Sándor Hunor, Pasinszky Krisztián, Péntek Gábor, Sóvágó Péter. Internetes jegyzet. 2005.

The V-Model. Christian Bucanac. Quality Management course reportat University of Karlskrona. 1999.

Szoftverfejlesztés. Dr. Mileff Péter. Miskolci Egyetemi jegyzet MSc-s hallgatók számára. 2008.

Szoftvertechnológia és UML. Sike Sándor, Varga László. ELTE Eötvös Kiadó. 2008.

Programtervezési minták. Erich Gamma, Richard Helm. Kiskapu Kiadó. 2004.

Szoftverfejlesztés C ++ nyelven. Benedek Zoltán, Levendovszky Tihamér. Szak Kiadó. 2007.

Tiszta kód - Az agilis szoftverfejlesztés kézikönyve. Martin, Robert C. Kiskapu Kft. 2010.

Projektmenedzsment a szoftverfejlesztésben. Langer Tamás. Panem Könyvkiadó. 2007.

Implementációs minták. Kent Beck. Panem Könyvkiadó. 2008.

Agile Software Development, Principles, Patterns, and Practices. Robert C. Martin. . 2002.

Információrendszer-fejlesztés. Raffai Mária. Novadat Bt kiadó. 1999.

Egységesített megoldások a fejlesztésben,. Raffai Mária. Novadat Bt kiadó. 2001.

UML Applied – Second Edition; Object Oriented Analysis and Design. . Ariadne training. 2005.

Introduction to Software Testing. Paul Ammann, Jeff Offutt. Cambridge University Press. 2008.

VIRCA hivatalos oldala. . http://www.virca.hu. 2011.

Az RTM specifikáció hivatalos oldala (Openrtm-aist működését leíró dokumentáció). . http://www.omg.org/spec/RTC. 2011.

OpenRTM-aist hivatalos oldala. . http://www.openrtm.org. 2011.

Corba specifikáció hivatalos oldala. . http://www.omg.org/spec/CORBA. 2011.

ICE hivatalos oldala. . http://zeroc.com. 2011.

SysML hivatols oldala. . http://www.sysml.org. 2011.