Debreceni Egyetem
Informatikai Kar
Informatikai Rendszerek és Hálózatok tanszék

Webes alkalmazásfejlesztés

Témavezető:
Dr. Kuki Attila
egyetemi adjunktus

Készítette:
Molnár Zoltán
programtervező matematikus
hallgató

Debrecen, 2008

1 Bevezetés

Napjainkat rohamos fejlődés jellemzi a technika számos területén, nincs ezzel másképpen a számítástechnika sem. Habár az első elektronikus számítógépet (az ENIAC-ot) már 1946-ban elkészítették, és az Intel 4004-es processzorok 1971-es piacra kerülésével már negyedik generációs számítógépekről beszélünk, az első személyes használatra készített IBM PC-t csak 1981-ben mutatták be. Ekkor kezdődött a személyi számítógépek immár 17 éves története, és ezen idő alatt hatalmas változáson mentek keresztül. Ma már az élet szinte minden területén megjelent valamilyen formában a számítástechnika, számítógépek működtetnek rengeteg automatizált rendszert, parányi áramkörök vezérelnek számos hétköznapi eszközt.

Az általános célra kifejlesztett számítógépek természetesen magukkal hozták a programozhatóságot, így a hardver irányítását valamilyen szoftver végzi mind a mai napig. Ahogy egyre bonyolultabb és nagyobb teljesítményű hardverelemeket fejlesztettek ki, úgy tették lehetővé egyre nagyobb és összetettebb szoftverrendszerek létrehozását. A szoftverek bonyolultságának növekedése szükségszerűen megkívánta a különböző alkalmazásfejlesztési módszertanok kidolgozását, melyek segítségével továbbra is fenntartható maradhat a fejlődés, a szoftverek ellenőrzött minősége, és csökkenthető a egy-egy fejlesztési projekt kudarcának valószínűsége.

Ezzel gyakorlatilag el is érkeztünk a dolgozat témájához, az alkalmazásfejlesztéshez, melynek keretein belül szeretném kifejteni egy konkrét alkalmazás teljes fejlesztési életciklusát, a kezdeti elképzelés vázolásától kezdve, a követelmények feltárásán és a tervek elkészítésén túl egészen az implementációig. Szeretném ismertetni az egyetemi évek alatt elsajátított elméleti hátteret és ezek visszatükröződését a példaként létrehozott alkalmazás fejlesztése során. Célom továbbá, hogy a dolgozat végén összegezzem mit sikerült megvalósítani, és milyen módon lehetne bővíteni az alkalmazást.

1.1 Célkitűzés

A dolgozat keretein belül szeretném az alkalmazásfejlesztés elméleti hátterét ismertetni egy általam megvalósított alkalmazáson keresztül. Ez az alkalmazás egy motorcsónak kölcsönzéssel foglakozó képzeletbeli cég működését megkönnyítő webes alkalmazás. Szeretnék végigmenni a szoftver teljes életciklusán, elindulva a kezdeti elképzelések lefektetésével, a követelmények rögzítésén, és a tervek elkészítésén túl egészen egy kis mintaalkalmazás imp-

lementálásáig, és az elkészült alkalmazás dokumentálásáig. A hangsúlyt természetesen az elemzésre és a tervezésre szeretném helyezni, az elkészítésükkor szerzett tapasztalataimat szeretném megosztani, illetve a megszerzett tudásomat reprezentálni, mindezt példákkal illusztrálva. Az implementáció elkészítésével pedig betekintést szeretnék nyerni a mai webes technológiákba, azok lehetőségeibe és működésükbe.

1.2 Alkalmazásfejlesztés

Az alkalmazásfejlesztés olyan tevékenységek és kapcsolódó eredmények halmaza mely egy szoftverterméket állít elő. Egy olyan kis rendszer elkészítése esetében, amelyet egyetlen ember is képes véghezvinni, a fenti folyamat rövid, átlátható és egyszerű. Azonban egy összetettebb szoftver fejlesztése már több embert kívánhat, akikkel valamilyen módon meg kell osztani a feladattal kapcsolatos elképzeléseket és követelményeket. Ez már megköveteli a szervezést és az alapos tervezést is. Ezzel gyakorlatilag megjelenik a szoftvertervezés. Ez a fogalmat 1968-ban használták először, az első szoftverkrízis idején. Ugyanis a harmadik generációs számítógépek megjelenése lehetővé tette olyan problémák alkalmazásokkal való megoldását is, amelyekre korábban nem volt lehetőség. Azonban ez jóval nagyobb és összetettebb szoftvereket jelentett, melyek fejlesztése közben hamar megmutatkoztak az addigi fejlesztési módszerek gyengéi. Tudni kell ugyanis, hogy mivel a szoftver nem anyagi természetű, így nem érvényesek rá az ipari gyártási folyamatokból fakadó korlátok. Más szavakkal nincs korlátja annak, hogy egy szoftvernek mit kell tudnia, és ez a „hiányosság” azt eredményezi, hogy az adott szoftver könnyen szélsőségesen bonyolulttá és nehezen érthetővé válhat.

A korábbiakban említett szoftverkrízis idején ezzel az égető problémával szembesültek a programozók, melyre feltétlenül megoldást kellett találniuk, hiszen a nagyobb projektet sokszor éveket csúsztak, és akár 2-szer, 3-szor több pénzt emésztettek föl, mint az előzetes becslések során azt megállapították. Ezeknek megfelelően a fejlesztési projektek jelentős része kudarcra volt ítélve, sokuk befejezésük előtt kifutott az rendelkezésre álló időkeretből és pénzügyi forrásokból, így sikertelen projektek száma rohamosan nőtt. Ezért új technikákra és módszerekre volt szükség, hogy ellenőrizni lehessen a nagy és összetett rendszerek fejlesztését. Ezek az új technikák a szoftvertervezés részét képezik, melyek segítségével biztosítható a fejlesztés költség- és időkorlátainak betartása, felügyelhető a készülő szoftver minősége, és ezáltal növelhető a projekt sikerességének esélye.

Az alkalmazásfejlesztés alapvetően négy szakaszra bontható:

1. Szoftverspecifikáció: mely során rögzítjük az elkészítendő szoftver működését és az ehhez kapcsolódó követelményeket
2. Szoftverfejlesztés: a fenti specifikációnak megfelelően elkészítjük a szoftvert
3. Szoftvervalidáció: ezen lépés során biztosítjuk azt, hogy a megrendelő igényeinek megfelelően készült el a szoftver
4. Szoftverevolúció: a szoftvert az idő múlásával tovább kell fejleszteni a megrendelő későbbi felmerülő igényei szerint.

A fent felsorolt tevékenységeket kell megszerveznünk és különböző szinteken és mélységben részletezni a szoftver méretének és összetettségének megfelelően. Ezeknek a tevékenységeknek a végrehajtásában van segítségünkre a szoftvertervezés kapcsán említett módszerek, melyek körültekintő és megfelelő megválasztása esetén nő a fejlesztés hatékonysága és a készülő szoftver minősége. Ezekről a technikákról a későbbiekben lesz szó részletesebben.

A dolgozat témája szerint az alkalmazásfejlesztés egy szűkebb területéről szól, a webes alkalmazásfejlesztésről. Ez a terület természetesen más technológiai megoldásokat kíván, mint amiket az asztali alkalmazások fejlesztése során felhasználhatunk, és bizonyos szempontból több kockázatot is rejt magában, hiszen ha egy nagyobb webportálról van szó, akkor a szokásos dolgokon túl kiemelten kell foglakoznunk a biztonsággal, a rendszer skálázhatóságával és terhelhetőségével, melyek tovább növelik a rendszer bonyolultságát.

1.3 Az internet és a web

A számítógépek megjelenése után csak idő kérdése volt azok valamilyen módon való összekötésének igénye. 1969-ben ARPANET néven hozták létre az USA-ban az első számítógépes hálózatot, és ez tekinthető a mai Internet ősének. Ez a hálózat még egy ún. NCP csomagkapcsolt szabvány szerint működött, de a későbbiekben a Vinton Cerf és Robert Kahn által kifejlesztett TCP/IP protokollt alkalmazták mely szintén csomagkapcsolt elven működik, és ma is alapját képezi az Internetnek, a hálózatok hálózatnak. Az ARPANET-nek nevezett hálózat az amerikai védelmi minisztérium felügyelete alatt jött létre, ezért eleinte csak katonai létesítmények összekapcsolását szolgálta (és ez volt az eredeti célja is, olyan redundáns hálózat létrehozása, mely egy esetleges szovjet atomtámadás esetén is működőképes marad). A hálózatban rejlő lehetőségeket az egyesült államokbeli Nemzeti Tudományos Alapítvány

(NSF) is felismerte, de mivel nem kapott lehetőséget az ARPANET-be való bekapcsolódásra, ezért úgy döntött hogy saját hálózatot épít, melynek segítségével az államokban található 6 szuperszámítógépet működtető egyetemet akarta összekötni, ezzel is könnyítve a kutatási eredmények megosztását. Ezzel megszületett az NSFNET, mely a mai napig részét képezi az USA-beli országos gerinchálózatnak. Fontos megemlíteni, hogy az NSFNET is a TCP/IP-t használta adatátviteli protokollként, így képes volt együttműködni a már meglévő ARPANET-tel. 1983-ban le is választják MILNET néven a katonai szegmenst az ARPANET-ről és összekapcsolják az NSFNET-tel. Az igazi áttörést 1988 jelenti, amikor engedélyezik a hálózat kereskedelmi célú felhasználását, tehát bármilyen legális célra felhasználható lett az Internet. A telekommunikációs céges sorra kapcsolódtak be a már meglévő hálózatba, illetve egészítették ki azt saját hálózatukkal. Az Internet mind a mai napig folyamatosan terjed, egyre újabb és újabb részhálózatokkal bővül.

Az Internet lehetővé teszi, hogy különböző protokollokon keresztül végezzünk adatcserét, így FTP-n keresztül küldhetünk fájlokat, SMTP és POP segítségével e-maileket küldhetünk és fogadhatunk, SSH segítségével titkosított csatornán kommunikálhatunk, IRC keresztül cseveghetünk, TELNET-en vagy RPC-vel távolról elérhetünk egy másik gépet.

Már a 80-as évek végén létezett egy GOPHER-nek nevezett protokoll, mely segítségével online tartalmakat lehetett nézegetni, és közöttük a linkekhez hasonló módon menük segítségével navigálhattunk. Ez az eljárás a Web elődjének tekinthető. Ezt váltja fel a HTML nyelv, melyet 1990-ben a svájci CERN kutatóintézetben egy angol tudós, Tim Berners-Lee publikál. Ezzel megszületik a WWW, mely nem más, mint hipertext linkek segítségével összekapcsolt szöveges dokumentumok hálózata, amik az Interneten keresztül érhetőek el HTTP protokoll segítségével. Ugyan a köznyelvben az Internet és a Web egymás szinonimái, de igazából a Web csak egy szolgáltatás amit elérhetünk az Interneten keresztül. A weboldalak egy speciális ún. jelölő nyelv segítségével vannak leírva, ezt hívjuk HTML-nek hívunk. Ez pedig nem más mint egy strukturált dokumentum, melynek szövegét különböző html-elemek segítségével formázhatunk, linkeket adhatunk meg benne, képeket, hangot, mozgóképet ágyazhatunk bele, és bizonyos fokig szemantikus információkat is tárol az adott szövegről. Egy weboldalt reprezentáló HTML dokumentumot egy böngésző képes megjeleníteni (renderelni), tehát felismeri a különböző HTML elemeket és ezeknek megfelelően formázva jeleníti az adott oldalt.

Az első időkben a Web gyakorlatilag csak ennyiről szólt: statikus HTML oldalak voltak webszervereken elhelyezve, melyeket URL-ek segítségével lehetett elérni, a használt böngésző letöltötte a HTML oldalt és megjelenítette a képernyőn. Az oldalak statikusak voltak abban az értelemben, hogy csak úgy lehetett őket megváltoztatni, ha a szerveren lévő dokumentumot módosítottuk, semmilyen módon sem lehetett a kliens oldalról hatással lenni az oldal tartalmára. Ezt hívják Web 1.0-nak.

Természetesen a fenti megoldás tökéletesen kiszolgálta azt amire kitalálták: egy adott kutatási eredményt vagy dolgozatot begépelve, a HTML elemekkel gyorsan és könnyedén lehetett megformázni, aztán publikálni a Web-en keresztül és ott hagyni őket akár az idők végezetéig, nem kellett utólag módosítani. Azonban az Internet terjedését az üzleti szektor megjelenése gyorsította fel igazán, ezért a gazdasági élet a maga kívánságai szerint alakította tovább a Web-et: megjelent a webes grafika, a különböző céges honlapok, a maguk cégtábláival, logóival és hirdetéseivel. Továbbá megjelentek már dinamikus megoldások is, ahol egy kis szerveroldali program az oldal lekérésének időpontjában generálta le a HTML dokumentumot, melynek adatait egy adatbázisból vette, ezzel elérve, hogy akár minden letöltésnél máshogy nézzen ki az oldal, képes volt egy regisztrált felhasználói kör adatait tárolni, és egy olyan funkcionalitást nyújtani amely két oldalletöltés között is megőrizte tartalmát. Ezzel a lehetőségek egy igen széles tárháza nyílt meg, és a webes alkalmazásfejlesztés virágzásnak indult. Eleinte csak C-ben vagy egy Unix-os bash szkriptnyelven megírt program állt rendelkezésre a szerveren, ennek minden előnyével és hátrányával. Web lehetőségeinek bővülésével és az egyre nagyobb üzleti igényeknek megfelelően a technológiák is egyre jobban fejlődtek, megjelentek a Java appletek, a flash animációk vagy a PHP, majd egész szerveroldali keretrendszerek mint az ASP vagy a JSP. A mai weboldalak nagy része ilyen megoldással működik, ezt hívjuk Web 1.5-nek.

Azonban a fejlődés nem állt meg, napjainkban azonban már nem csak új technológiák látnak napvilágot, hanem a fejlesztők szemléletmódja is nagyban megváltozott. Olyan új szolgáltatásokat akarnak bevezetni, melyek segítségével a nagyközönség is kifejezheti önmagát. Megjelentek a közösségi oldalak, blogok, kép- és videómegosztó portálok, szociális hálózatokat építhetünk ki, és saját híroldalt vezethetünk. Ezzel olyan nem technikai jellegű változások jelentek meg, melyeket egy egyre bővülő közösség alakít, mely egy új minőséget képes létrehozni. Természetesen vannak technikai jellegű újítások is: megjelent az RSS, mely segítségével könnyen áttekinthetjük egy oldal bejegyzéseit, terjed a szemantikus web, mely a különbö-

ző keresőrobotok és értelmező programok számára próbálja meg szemantikusan is feldolgozhatóvá tenni az oldalak tartalmát. Előtérbe került az oldalak akadálymentesítése, azaz csökkent képességűek számára is elérhetővé tenni a tartalmakat, egyre jobban betartják a szabványokat, XHTML formátumot használnak, CSS leírókkal formázzák az oldalakat, illetve létrehozták az AJAX technológiát mely aszinkron adatátvitelt tesz lehetővé a kliensoldali böngésző és a szerveroldali webalkalmazás között, tehát a teljes oldal újratöltése nélkül képes frissíteni a tartalmát. Ez már mind a Web 2.0, amely úgy néz ki a webalkalmazások irányába fejlődik majd tovább, így egyre inkább csak egy böngészőre van szükségünk a gépen, és ebben fut minden alkalmazás, legyen szó levelezésről, szövegszerkesztésről vagy egy vállalat irányításáról. Ezzel gyakorlatilag központosul az alkalmazások karbantartása, hiszen csak egy webszerverre kell feltenni ezeket a szoftvereket, és a kliensgépeken csak egy böngésző telepítésére van szükség, ezáltal könnyebben karbantartható és védhető az adott alkalmazás.

2 A szoftverfejlesztés folyamata

Minden terméknek van életciklusa, mely a rá vonatkozó igények felmerülésétől a termék megvalósításán és használatán túl egészen a használatból való kivonásáig tart. A szoftverfolyamat tevékenységek és kapcsolódó eredmények sora, amelyek egy szoftvertermék előállításához vezetnek. A szoftverfolyamat szellemi folyamat, így mint minden ilyen, összetett és nagyban függ az emberi tényezőktől és döntésektől. Éppen ezért, mert emberi beavatkozást és kreativitást igényelnek, a folyamatok automatizálása csak kevésbé érvényesült: bizonyos résztevékenységeket már elvégezhetőek programok segítségével, de nincs reális lehetőség nagyobb mértékű automatizációra az elkövetkezendő évek során azokon a területeken, ahol a szoftverfolyamatban résztvevő tervezők kreativitására van szükség.

2.1 Szoftverfejlesztés

A szoftverfejlesztés tehát olyan folyamat, amely során egy adott igényt kielégítő szoftver első példányát létrehozzuk. A folyamat indulhat "nulláról", azaz úgy, hogy nincs jelentős előzménye, nincs meglévő, hasonló szolgáltatású rendszer a birtokunkban; de indulhat úgy is, hogy egy meglévő szoftver továbbfejlesztését határozzuk el. Fejlesztésről csak akkor beszélünk, ha a folyamat során jelentős újdonságokat hozunk létre.

A bevezetésben már említettem, hogy a fejlesztés folyamatának négy jellegzetes lépését különíthetjük el: a specifikációt, a tervezést és implementációt, a validációt és végül a szoftver evolúcióját.

A fejlesztés jelentős része gondolati síkon, azaz a modellek síkján folyik. A valóságos dolgokról bennünk élő képeket nevezzük modelleknek, ennek a képnek a kialakítását pedig modellezésnek. Egy modell sohasem tükrözi a valóság minden apró részletét, hiszen mindig valamilyen céllal alkotjuk. A modellben azokra a tulajdonságokra koncentrálunk, amelyek a cél elérése szempontjából fontosak. Ezért a modell szükségképpen absztrakt. Milyen összefüggés áll fenn a modell és a valóságos dolgok között? Ugyanarról a valóságos dologról több modellt is alkothatunk, amelyek mindegyike más-más szempontból emel ki lényeges tulajdonságokat. Fordított irányban: egy modellnek több valóságos dolog is megfelel. Ezek a valóságos dolgok a modellben figyelmen kívül hagyott (lényegtelennek tartott) tulajdonságaikban térnek el egymástól.

A szoftverfejlesztés során valamely speciális feladat megoldására univerzális eszközöket használunk, azaz speciális rendszerként viselkedő számítógépes rendszert hozunk létre. Eközben mind a speciális rendszerről kialakított modellre, mind a felhasználható számítástechnikai eszközökről kialakított modellre szükségünk van, a cél eléréséhez pontosan ennek a két modellnek az elemeit kell megfeleltetnünk egymásnak.

A fejlesztés kezdetekor általában az adott probléma szakterületének fogalmaival kifejezett igényekből, vagyis a fogalmi modell egy változatából indulunk ki, és a követelményeket kielégítő valóságos rendszerhez kell eljutnunk. A legtöbb gondot ennek során az okozza, hogy a szóban forgó rendszerek bonyolultak, amiből számos probléma származik. Rögtön elsőként említhetjük, hogy a bonyolult modellek áttekinthetetlenek, megértésük, kezelésük nehézkes. Nincs olyan zseniális tervező, aki egy rendszernek, amelynek programja több tízezer forrássorból áll, és sok (mondjuk tíznél több) processzoron fut, valamennyi részletét egyidejűleg fejben tudná tartani, illetve egy-egy tervezői döntés minden kihatását átlátná. A bonyolult rendszerek modelljeit és a modellek közötti megfeleltetéseket csak több lépésben tudjuk kidolgozni. Minden lépés hibalehetőségeket rejt magában, mégpedig minél bonyolultabb a rendszer, annál nagyobb a hibázás esélye. Az áttekintés nehézségein túl komoly probléma a résztvevők információcseréje is. A rendszerrel szemben támasztott követelményeket általában nem a fejlesztők, hanem a felhasználók (megrendelők) fogalmazzák meg, maga a fejlesztés pedig általában csoportmunka. Igen jó lenne, ha a folyamat minden résztvevője pontosan

ugyanazt a gondolati képet tudná kialakítani önmagában az amúgy igen bonyolult rendszerről. Sajnos ezt nagyon nehéz elérni, nagy a félreértés veszélye. (Tapasztalatok szerint a probléma egyszemélyes változata is létezik: igen jó lenne, ha egy fejlesztő néhány hét elteltével fel tudná idézni ugyanazt a képet, amit korábban kialakított magában a rendszerről.)

Ezen a bonyolultságon valahogyan uralkodni kell.

A bonyolultságot általában úgy érzékeljük, hogy nagyon sok mindent kellene egyszerre fejben tartanunk, és ez nem sikerül. Figyelmünket hol az egyik, hol a másik részletre koncentráljuk, és a váltások közben elfelejtjük az előzőleg tanulmányozott részleteket. A bonyolultság uralása érdekében olyan modelleket kellene alkotnunk, amelyek lehetővé teszik, hogy az egyidejűleg fejben tartandó információ mennyiségét csökkenthessük, a tervezés közben hozott döntések kihatását pedig az éppen áttekintett körre korlátozzuk. Erre két alapvető eszköz áll rendelkezésünkre, a részletek eltakarása (absztrakció), illetve a probléma (rendszer) egyszerűbb, egymástól minél kevésbé függő részekre bontása (dekompozíció). Ezzel a két gondolkodási technikával olyan struktúrákat hozhatunk létre, amelyeken mozogva figyelmünket a rendszer különböző részeire, különböző nézeteire irányíthatjuk.

2.2 Fejlesztési modellek

Tehát a bonyolultság ellenőrzéséhez absztrakt modelleket kell készítnünk, ezek segítségével a probléma egy olyan magasszintű leírását hozhatjuk létre, amely megérthető, és tovább finomítható, az egyes részek tovább részletezhetőek a kellő mélységig. Ezen leírások összessége adja a teljes megvalósítandó rendszer modelljét.

Látható, hogy maga a fejlesztés is bonyolult folyamat, így a szoftverfolyamat is modellezhető, amely nem más mint a folyamat egy absztrakt reprezentációja. Minden egyes folyamatmodell különböző speciális nézőpontból reprezentál egy folyamatot, de ilyen módon csak részleges információkkal szolgálhat magáról a folyamatról.

A továbbiakban szeretnék néhány folyamatmodellt ismertetni. Ezek olyan általános leírások lesznek, amelyek nem a szoftverfolyamat pontos leírásai, inkább csak hasznos absztrakciók, amelyeket a szoftverfejlesztés különböző megközelítési módjainak megértéséhez lehet felhasználni. Ezeket a gyakorlatban nem kizárólagos módon használják, hanem keverik őket, így az egyes előnyöket kiemelhetik és csökkenthetik az alkalmazásukkal járó hátrányokat. Megtehető az, hogy egy nagy rendszer esetében a különböző alrendszerek fejlesztése különböző folyamatmodellek alapján történik, attól függően, hogy az egyes részek milyen fej-

lesztési ütemet és áttekinthetőséget kívánnak. Továbbá mindenki aki alkalmazza őket, olyan módon alakítja az egyes folyamatokat, hogy a számára a lehető legjobban megfelelje, illetve tapasztalatai alapján új részekkel egészíti ki, vagy ötvözi más technikákkal.

2.3 A vízesésmodell

A szoftverfejlesztés problémáinak felismerése után a projekt folyamatának leírására kialakult legrégebbi modell a vízesésmodell, amelyet fázismodellnek is nevezünk. Nevét a szemléltető ábra jellegzetes alakjáról kapta, mivel az egyes fázisok lépcsősen kapcsolódnak egymáshoz. A modell a termékfejlesztésre koncentrál, azaz az első működő példány előállításáig terjedő szakasz lefolyását ábrázolja. Ma általában a változtatásokat is kezelő ciklikus modellekbe ágyazva használják, önmagában legfeljebb nagy egyedi szoftverek esetén fordul elő.

A modell alapvető szakaszai az alábbi fejlesztési tevékenységekre képezhetőek le:

1. Követelmények elemzése és meghatározása: a rendszer szolgáltatásai, megszorításai és céljai a rendszer felhasználóival történő konzultáció alapján alakulnak ki, ezeket később részletesen kifejtik, és ezek szolgáltatják a rendszer-specifikációt.
2. Rendszer- és szoftvertervezés: a rendszer tervezési folyamatában választódnak szét a hardver- és szoftverkövetelmények, itt kell kialakítani a rendszer átfogó architektúráját. A szoftver tervezése alapvető szoftverrendszer-absztrakciók, illetve a közöttük lévő kapcsolatok azonosítását és leírását is magában foglalja.
3. Implementáció és egységteszt: ebben a szakaszban a szoftverterv programok, illetve programegységek halmazaként realizálódik. Az egységteszt azt ellenőrzi, hogy minden egyes egység megfelel-e a specifikációjának.
4. Integráció és rendszerteszt: megtörténik a különálló programegységek, illetve programok integrálása és teljes rendszerként történő tesztelése, hogy a rendszer megfelel-e a követelményeknek, a tesztelés után a szoftverrendszer átadható.
5. Működtetés és karbantartás: általában ez a szoftver életciklusának leghosszabb fázisa (bár ez nem szükségszerűen igaz). Megtörtént a telepítés és a rendszer gyakorlati használatbavétele. A karbantartásba beletartozik az olyan hibák kijavítása, amelyekre nem derült fény az életciklus korábbi szakaszaiban, a rendszeregységek implementációjának továbbfejlesztése, valamint a rendszer szolgáltatásainak továbbfejlesztése a felmerülő új követelményeknek megfelelően.

Minden fázis eredménye egy dokumentum, melyet jóváhagytak és elfogadtak. A következő fázis csak ezután indulhat el, tehát látható, hogy az egyes fázisok mereven követik egymást, azonban a valóságban sokszor átfedik egymást, és információt szolgáltatnak az összes többi fázishoz. A fejlesztés semmiképp nem lehet teljesen lineáris, hiszen minden fázisban találhatunk hibákat a modellben, és akkor a kellő mélységben vissza kell lépnünk a hiba forrásához, és kijavítani őket, így inkább a fejlesztési lépések iterációjának tekinthető a modell.

A modell előnye, hogy a készülő rendszer jól definiált lesz és részletes dokumentáció készül minden fázisról, mely növeli a karbantarthatóságot. Továbbá a rendszer átgondolt volta miatt robosztus lesz, nincsenek benne feleslegesen bonyolult vagy redundáns részek. Hátránya, hogy hiba esetén akár a legelső fázisig vissza kell menni és kijavítani a problémát, ami maga után vonja a többi később fázis újrajátszását is. Ez rendkívül költségessé teszi a folyamatot, így sokszor néhány iteráció után már nem is lépnek vissza, hanem elfogadják a hibát és a későbbi lépésekben megpróbálják kikerülni azt. Ezzel viszont csak azt érik el, hogy egy bizonyos küszöb után, már nem azt csinálja a szoftver amire eredetileg a megrendelő szánta, rontja a szoftver struktúráját, hiszen implementációs fogásokkal kényszerül a tervezési hibák kivédésére.

A modell hátránya abból adódik, hogy az egyes fázisok túlságosan szeparáltak, így már a fejlesztés korai szakaszában el kell köteleznie magukat a tervezőknek így a későbbi módosításokra nincs lehetőség. Ezért ez a modell csak akkor alkalmazható eredményesen, ha a követelmények már jól ismertek, és nem várható hogy változás történik bennük.

2.4 Inkrementális fejlesztés

A vízesésmodell igen jó szolgálatot tett annak tudatosításában, hogy a szoftverfejlesztés nem azonos a programozással. A modell így önmagában a nagy, egyedi rendszerek fejlesztési folyamatának kezelésére alkalmas. Korlátai akkor jelentkeztek, amikor kiderült, hogy a szoftver változási hajlama még az egyedi rendszerek esetén sem szorítható merev fázishatárok közé. A leggondosabb tervezés ellenére is csaknem minden rendszer átadásakor, vagy próbaüzeme során felmerülnek olyan módosítási igények, amelyek még a követelményekre is visszahatnak. Az általános célú, nagypéldányszámú szoftverek esetén a módosítási igények pedig folyamatosan jelentkeznek.

A gyakorlatban a legtöbb szoftver életének nagyobbik részét teszi ki – és a költségek nagyobb részét is igényli – az egyre újabb változatok, verziók előállítása, a módosítások vég-

rehajtása. Ezt a tevékenységet karbantartás (maintenance) néven szokás említeni. Az új szoftver fejlesztése ebben a folyamatban mindössze egy speciális esetet képvisel, az első változat előállítását. Ekkor a "semmiből" hozzuk létre az első verziót, ezt követően pedig a már meglévő változatok módosításával állítjuk elő az újabbakat.

Tekintettel arra, hogy egy változat – főleg az első – önmagában is elég nagy lehet ahhoz, hogy évekig készüljön, a fejlesztés közben megszerzett ismeretek fényében szükségessé válhat a követelményelemzés a tervezés és az implementáció átdolgozása. Habár a vízesésmodell előnye, hogy egyszerűen kezelhető, és hogy különválasztja a tervezési és implementációs fázisokat, azok végeredményként olyan robusztus rendszerekhez vezetnek, amelyek alkalmatlanok az esetleges változtatásokra. Ezzel szemben viszont az evolúciós megközelítési mód megengedi, hogy elhalasszuk a követelményekkel és a tervezésekkel kapcsolatos döntéseket, de ezek gyengén strukturált és nehezen megérthető és karbantartható rendszerekhez vezetnek.

2.5 Spirális modell

A szoftverfolyamat spirális modellje manapság nagyon széles körben ismert és alkalmazott modell. Az előző modellektől eltérően, a spirális megközelítés a szoftverfolyamatot nem tevékenységek és a közöttük található esetleges visszalépések sorozataként tekinti, hanem egy spirál mentén reprezentálja azokat. A spirál minden egyes körben a szoftverfolyamat egy-egy fázisát írja le. Így a legbelső kör a megvalósíthatósággal foglalkozik, a következő a rendszer követelményeinek meghatározásával, a következő pedig a rendszer tervezésével és így tovább.

A spirál négy síknegyedre osztható fel az alábbiak szerint:

1. Célok kijelölése: az adott fázis által kitűzött célok meghatározása, ekkor kell azonosítani a folyamat megszorításait, a terméket, fel kell vázolni a kapcsolódó menedzselése tervet. Fel kell ismerni a felmerülő kockázati tényezőket, és ezektől függően alternatív stratégiákat kell tervezni, amennyiben ez lehetséges.
2. Kockázat becslése és csökkentése: minden az előző lépésben felismert kockázati tényezőt részletes elemzés alá kell vetni, és kidolgozni a megfelelő lépéseket, amelyek segítségével az adott kockázat minimalizálható.
3. Fejlesztés és validálás: ebben a részben választunk egy fejlesztési modellt, amely a lehető legjobban figyelembe veszi az előző lépésben feltárt kockázatokat, pl. ha

az alrendszerek integrálásában rejlenek a legfőbb kockázatok, akkor a vízesésmodellt érdemes alkalmazni, míg egy felhasználói felületek szempontjából kritikus alkalmazást prototípuskészítéssel érdemes elkészíteni.

4. Tervezés: az előző lépések fényében el kell dönteni, hogy érdemes-e folytatni a folyamatot egy következő ciklussal vagy sem. Ha a folytatás mellett döntünk, akkor meg kell tervezni a folyamat következő fázisát.

Nagyon fontos különbség az előző modellekhez képest, hogy ez explicit módon számol a fejlesztés során fellépő kockázati tényezőkkel, és megpróbálja azokat a lehető legjobban szem előtt tartani, és minimalizálni, ezzel is elősegítve a projekt sikerességét. Mivel a kockázatok azt eredményezhetik, hogy egy projekt túllépi a neki szánt idő vagy költségkeretet, ezért a menedzsment számára nagyon fontos tevékenység ezeknek a kockázatoknak a csökkentése.

2.6 A választás

A példa alkalmazás kis méretéből adódóan a fejlesztéshez szükséges módszertannak kis elég egyszerűnek és átláthatónak kell lennie, hiszen ahogy a hiányos és módszertelen fejlesztés a projekt kudarcát okozhatja, ugyanúgy a feleslegesen körülményes és túlméretezett megközelítés is rossz hatással lehet a kivitelezésre. Azonban ez nem jelenti azt, hogy ebben az esetben nincs is szükség odafigyelni a projekt menetére és felépítésére, csak meg kell találni a megfelelő egyensúlyt.

A példa alkalmazás megvalósításához általam választott módszer nem a fentiek közül való, hanem azok egy megfelelő keveréke. Az ún. Q-módszer alapjaiból táplálkozik, ugyanis ez egy olyan agilis módszertan, amely kifejezetten kisméretű projektek kivitelezéséhez dolgoztak ki. Csak a szükséges dokumentációkat készítettem el, de ezekben nem tértem el lényegesen a szabványok által meghatározott felépítéstől. Törekedtem az egyszerűségre és az átláthatóságra, hogy ne vesszen el a lényeg a sok formaiság között. Természetesen ezeket az elkészített dokumentumokat a következő fejezetekben a megfelelő helyen szeretném szemléltetésként is felhasználni, illusztrálni velük az elmélet és a gyakorlat összefüggéseit.

3 Követelménytervezés

Egy alkalmazás, legyen bármilyen kicsi is, egy rendszert alkot, melynek vannak összetevői, ezek a összetevők valamilyen módon összehangolt munkát végeznek, és ezen együttműködés eredményeképpen az alkalmazás szolgáltat valamilyen funkcionalitást, amit a fel-

használók igénybe vesznek. Ahogyan a mérnöki világban egy rendszert az elkészítése előtt érdemes megtervezni, úgy egy szoftvert is tervezni kell, azelőtt hogy a programozók hada leülne és implementálná a problémát. Ehhez a tervezéshez viszont pontosan kell tudnunk, hogy mit szeretnénk megvalósítani. Tehát tisztában kell lennünk azzal, hogy mik a készülő szoftverrel szemben támasztott elvárások, mik a rendszer követelményei. Ezeket a követelményeket is tervezni kell, kellő részletességgel szükséges feltárni őket, és megfelelő pontossággal rögzíteni különböző dokumentumok formájában, hogy az implementáláskor felhasználhatóak legyenek.

Tehát a követelményeket is tervezni kell, mely folyamat során összegyűjtjük a rendszer leírásához szükséges követelményeket, és összefoglaljuk formális módon egy (vagy több) dokumentumban. Mindez négy nagy lépésből tevődik össze. Először is el kell készíteni egy megvalósíthatósági tanulmányt, amely felméri, hogy az elkészítendő rendszer tényleg hasznára válik majd üzleti szempontból a megrendelője számára. Ha a tanulmány pozitív eredményt ad, akkor fel kell tárni a követelményeket, elemezni kell a problémát, majd ezután ezeket a követelményeket valamilyen szabványos formára kell hozni, azaz specifikálni kell őket. Az utolsó lépésben a leírt követelményeket validálni kell, azaz meg kell állapítani, hogy a feltárt követelmények tényleg azt a rendszert írják le, amelyre a megrendelőnek szüksége van, tehát egy olyan rendszert definiál a specifikáció, amely tényleg a megrendelő kívánságai szerint fog működni.

3.1 A vízió

Egy szoftver fejlesztése akkor kezdődik, amikor valaki (általában egy cég) egy új elképzelést szeretne megvalósítani, vagy már egy meglévő folyamatot szeretne új, automatizált keretbe foglalni. Vegyük a mi hipotetikus csónakkölcsönző cégünket, ami már jó ideje működik, végzik a mindennapi üzleti tevékenységüket: motorcsónakokat adnak kölcsönbe egy általuk nyilvántartott ügyfélkörnek. Lehet kölcsönözni néhány napra, egy hétre vagy akár egy hónapra is. Többféle vízi járműből választhatunk, vannak kisebb kenuk, ladikok, motoros csónakok, és hatalmas jachtok is. A cég nyilvántartja az ügyfelei azon adatait, amelyeket a kölcsönzési szerződésben szerepeltetni kell, ahogy azt is nyilvántartja, hogy adott időben melyik jármű hol van, melyeket lehet még kikölcsönözni és melyek vannak már kikölcsönözve, ezek várhatóan mikor jönnek vissza, és lesznek újra kölcsönözhetőek. Továbbá azt is tudni kell a cég dolgozóinak, hogy az egyes csónakok milyen állapotban vannak, mikor mennyit

voltak kinn, mert bizonyos időszakonként szervizelni kell őket, a biztonsági előírásoknak megfelelően adott időközönként egy megfelelő átvizsgáláson kell átesniük.

Összefoglalva van egy már működő cégünk a maga bevett üzleti folyamataival és jól bevált eljárásaival. Azonban a világ fejlődik, és a piacon kemény verseny folyik az ügyfelek kegyeiért, hiszen nagyrészt azon múlik a cég fennmaradása, hogy mennyi potenciális vevőt tud magához vonzani, és kölcsönzésre ösztönözni. Manapság pedig a Web és az online tartalmak előretörését láthatjuk: egyre több cég hirdeti magát különböző webes oldalon, és rendelkezik saját honlappal is. Ezek egyszerűbb esetben csak tájékoztató jelleggel működnek, azonban egyre több oldalon már igénybe is vehetünk valamilyen formában az adott cég által nyújtott szolgáltatásokat, pl. online foglalásokat végezhetünk, szétnézhetünk a cég kínálatában, percre pontos információkat kaphatunk az árukészlet alakulásáról, és intézhetjük a céggel kapcsolatos ügyeinket is.

Éppen ezért a mi képzeletbeli cégünk vezetősége a fennmaradás érdekében úgy dönt, hogy egy saját webes portált készíttet egy szoftverfejlesztő vállalattal, melyen lehetőséget nyújt ügyfeleinek, hogy megtekinthessék a kölcsönözhető csónakok palettáját és az oldalon keresztül előfoglalásokat rögzíthessenek. Továbbá a vezetőség szeretné, ha alkalmazottai a cég nyilvántartásait is ugyanezen a rendszeren keresztül végeznék, a kölcsönzések regisztrálásától kezdve a karbantartás felügyeletéig.

Ezzel gyakorlatilag vázoltunk egy szokásos esetet: egy kis vagy közepes méretű cég, viszonylag kevés ráfordítással szeretné eddigi üzleti folyamatait részben automatizálni. A készülő alkalmazás vízióját csak ennyi alkotja, a megrendelő hozzávetőleges elképzelése, hogy milyen alkalmazást szeretne, de az esetek nagy részében nem is várhatunk többet. Nagyon kevés az olyan megrendelő, aki pontosan és tételesen tudja, mit is szeretne, mit vár el pontosan a készülő alkalmazástól. Azonban pontosan specifikált követelmények nélkül nem lehet jó programot írni abban az értelemben, hogy az elkészült szoftver tényleg hasznos lesz a felhasználói számára, és ténylegesen azt teszi, amit a megrendelő elvár tőle. Emiatt van szükség követelményelemzésre, melynek során fel kell tárni a rendszer által nyújtandó funkciókat, a rendszer határait és működési megszorításait.

A továbbiakban a követelménytervezés ezen aspektusait szolgáló főbb lépésekkel és a hozzájuk kapcsolódó dokumentumokat részletezem.

3.2 Követelmények elemzése

A követelménytervezés első lépése a megvalósíthatósági tanulmány elkészítése, azonban ez csak egy tömör leírás arról, hogy a megvalósítandó rendszer tényleg hozzájárul-e az üzleti tevékenység céljainak megvalósításához, és rendszer létrehozása a reá szánt idő és költségkeretben elvégezhető-e. Ezért ezzel nem foglalkozom bővebben.

Ezután jöhet a követelmények elemzése. Ennek első lépéseként a szükséges lehet, hogy a vízióban felvázolt rendszer általános jellemzőit egy dokumentumban rögzítsük, melyet Áttekintésnek is hívnak. Ez nem más, mint egy informális leírása a felhasználói követelményeknek, azaz természetes nyelven leírt kijelentések sorozata arról, hogy milyen funkcionalitást és szolgáltatásokat várunk el az elkészítendő rendszertől, és ezek mellett milyen megszorításoknak kell megfelelnie. A dokumentumot esetleg illusztráló ábrák és magyarázó diagramok egészíthetik ki. Az áttekintés informális jellegéből adódóan, könnyen érthető olyanok számára is, akik nem rendelkeznek a technikai háttér megfelelő ismeretével, hiszen nincsenek benne informatikai szakkifejezések, formális jelölések, vagy a rendszer konkrét megvalósítására vonatkozó kijelentések. Csupán körvonalakban kell leírnia a rendszer főbb funkcionalitását, és a vele szemben támasztott követelményeket.

Ennek a leírási módnak az a hátránya, hogy nem egyértelmű, hiszen hétköznapi nyelven sokszor nehéz lehet úgy megfogalmazni egy kijelentést, hogy azt mindenki ugyanúgy értelmezze. Azonban nem szabad körülményes módon, bekezdések hosszú során át magyarázni valamit, mert akkor elveszlik a kijelentés lényegi mondandója, ezért érdemes kerülni a túlságosan bőbeszédű körülírásokat. Továbbá az áttekintésben keverednek egymással a funkcionális és nem funkcionális követelmények, tehát míg bizonyos kijelentések a rendszer viselkedéséről és szolgáltatásairól szólnak, addig mások a rendszerhez kapcsolódó megszorításokra és korlátokra vonatkoznak, mi több, ezek a követelmények egy kijelentésen belül is megjelenhetnek keveredve egymással.

Mindezek ellenére erre a dokumentumra mindenképpen szükség van, hiszen ezen keresztül képes mindkét fél megérteni a problémát és annak megvalósítási módját. Sorra kell venni az áttekintés kijelentéseit, megvizsgálni azokat, szétválasztani a funkcionális és nem funkcionális követelményeket, a nyelvi pontatlanságokból eredő többértelműséget további kérdésekkel kell tisztázni, majd tételesen leírni az ilyen módon feltárt követelményeket, esetleg a felmerülő új követelmények megoldására javaslatot tenni. Fontos szem előtt tartani, hogy a megrendelőnek sokszor fogalma sincs arról mit szeretne, mit is vár el pontosan a ké-

szülő szoftvertől, ezért nagyban megkönnyíthetjük a munkát, ha egy kérdéssel kapcsolatban már egy vagy több javaslattal állunk elé, így tud választani közülük, melyiket szeretné.

A mi megrendelésünkkel sincs ez másképp, az áttekintést végigolvasva több példát is találunk a fentiekre.

1. példa: Áttekintés

a) A cég honlapját meglátogató ügyfélnek lehetősége kell legyen a kölcsönző által nyújtott csónakok adatait és fényképét megtekinteni, keresni azok között típus vagy más adat alapján történő lekérdezésekkel.
b) A csónakokat csak előre definiált ideig lehet kikölcsönözni. Ezt a kölcsönzési időt a kölcsönző ügyfél egy alkalommal meghosszabbíthatja a cég honlapján keresztül.
*c) A szoftverfejlesztés minden lépése az **ISO IEC 90003 2004 Software Standard**-nek megfelelően kell történnie, melynek következetes betartásából származó dokumentumok a cég vezetősége számára hozzáférhetőek kell legyenek.*

A fenti két idézet az Áttekintésből származik. Az a) idézet egy funkcionális követelmény, mely megmondja, hogy a készülő webalkalmazásnak lehetővé kell tennie, hogy a honlap látogatója böngészhessen a cég motorcsónakjai között. Azonban nem definiálja tisztán, hogy milyen módon lehet keresni közöttük, melyek pontosan azok a „más adatok" amelyek alapján kereshetünk. A b) részben szintén egy funkcionális követelményt láthatunk: látható, hogy a megrendelőnek már van egy kialakult gyakorlata a kölcsönzési idők meghatározásával kapcsolatban, azonban ezt még a későbbiekben tisztázni kell, hiszen itt nincs pontosan meghatározva. A c) idézet pedig egy nem funkcionális követelményre ad példát, hiszen egy megszorítást jelent, méghozzá nem is magára a szoftverre, hanem annak elkészítési folyamatát korlátozza, miszerint az említett ISO szabványnak kell megfelelnie a fejlesztés menetének, így a készülő tervezési és fejlesztési dokumentumok is meg kell hogy feleljenek ennek a szabványnak.

Az említett dokumentum végén a konkrét rendszerkövetelmények is megjelennek, melyek megszabják a használt operációs rendszert, melyen az alkalmazásnak futnia kell, az implementációs nyelvet vagy technológiát, a hardvermegszorításokat illetve meghatározhatja, hogy milyen egyéb szoftverekkel kell tudnia együttműködni (pl. adatbázis-szerver, alkalmazás-szerver stb.) Ezek a követelmények alapul szolgálnak a rendszer megvalósítási szerződéséhez, hiszen működési megszorításokat adnak. Azonban vigyázni kell, hogy csak olyan részletesen fejtsük ki a dolgot, hogy ne menjen át tervezésbe vagy implementációba ez a dokumentum.

3.3 Interjúk

Az áttekintés elkészítése után már van egy dokumentált alap, amelyen elindulhat a követelményelemzés. Az áttekintés körvonalazza azt a feladatkört amelyet az elkészítendő rendszernek el kell látnia. Azonban a rendszer implementálásához több információra van szükség, és alaposabb megfogalmazásra, tisztán kell látnia tervezőknek a rendszer működését. Mivel a legtöbb esetben (ahogyan esetünkben is) már egy létező és működő folyamat automatizálásáról van szó, így a rendszer működésének részleteit az üzleti folyamatban résztvevő ún. kulcsfiguráktól szerezhetjük meg. Kulcsfigurának nevezünk minden olyan személyt vagy csoportot, akiket a rendszer közvetve vagy közvetlenül érint. A kulcsfigurák közé számítanak a végfelhasználók, akik majd felhasználják a rendszer szolgáltatásait, és mindenki más is a cégen belül, akikre hatással lesz valamilyen formában az új szoftver telepítése. Bővebb értelemben ide tartoznak még a rendszert karbantartó szoftvermérnökök, a kapcsolódó rendszereket fejlesztő szakemberek, üzleti vezetők, szakterületi szakértők stb.

Tehát a cég működésekor végbemenő szakterületi folyamatok egyes lépéseit és feltételeit az említett kulcsfiguráktól tudhatjuk meg interjúk keretében. Az interjú kétféle lehet, vagy zárt, amikor egy előre meghatározott kérdéssorra kell válaszolnia az interjú alanyának, vagy lehet nyitott, amikor nincs előre meghatározva a beszélgetés menete, hanem bizonyos feladatköröket és problémákat érintenek, és hagyják hogy az interjúalany kifejtse a tudását és követelményeit az adott feladattal kapcsolatban. Természetesen ritkán alkalmazzák tisztán ezt a két típust, a valóságban ezek elegyével dolgoznak: mivel nem célravezető dolog azt mondani egy kulcsfigurának, hogy beszéljen arról, milyen rendszert szeretne, ezért általában néhány indító kérdéssel kezdenek, majd az ezek megválaszolása során felmerülő újabb kérdések nyomán haladnak tovább, egy megfelelő mederben tartva az interjú menetét. Ezekkel az interjúkkal elérhetjük, hogy átfogóbb képet alkothassunk arról, hogy az egyes kulcsfiguráknak mi a munkája, hogyan fognak érintkezni az új rendszerrel, és ha egy korábbi rendszer leváltása a cél, akkor a jelenlegi rendszernek milyen hátrányai és nehézségei vannak.

Annak ellenére, hogy a kulcsfigurák általában szívesen beszélnek arról, mit csinálnak munkájuk során, ezek az interjúk nem megfelelőek arra, hogy az alkalmazás szakterületi követelményeit megfelelően megértsük. Egyrészt azért, mert a kulcsfigurák a saját szakterületük zsargonját és terminológiáját használják, hiszen a folyamatok ismertetése lehetetlen vagy túlságosan körülményes lenne ezek nélkül a kifejezések nélkül. Ezért lesz majd szükség a következő részben ismertetett fogalomszótárra. Másrészt pedig bizonyos szakterületi ismeretek az

egyes kulcsfigurák számára olyannyira magától értetődő és természetes, hogy vagy nehezen tudják őket elmagyarázni, vagy annyira alapvetőnek tartják, hogy nem is említik meg ezeket. Azonban ezek a dolgok egy kívülálló (elemző) számára nem biztos, hogy ugyanilyen magától értetődőek, így ezeket nem is fogják figyelembe venni a követelmények leírásánál.

Azonban ezekből az interjúkból nyert információk jól kiegészítik a dokumentumokból és megfigyelésekből származó információkat, és néha az is előfordulhat, hogy a szakterületi dokumentumokon túl ez az egyetlen információforrás, mely segítségével megismerhetőek és megérthetőek a folyamatok. Viszont önmagukban az interjúk nem elegendőek, fontos információk maradhatnak ki, ezért mindig más követelményelemző technikák eredményeinek kiegészítésére szabad csak alkalmazni.

Végül a mi rendszerünk esetében kulcsfigurák lehetnek a cég alkalmazottjai, akik nap mint nap szolgálják ki a kölcsönző ügyfeleit, ők adhatnak betekintést a kölcsönzés és az előjegyzés folyamatába, továbbá a motorcsónakok szervízelésével kapcsolatos tevékenységek részleteit is feltárhatják. A vezetőség valószínűleg az alkalmazottak nyilvántartásával kapcsolatos folyamatokkal szemben támasztanak követelményeket, és a cég működésének felügyeletében szeretnének nagyobb befolyást. Egy harmadik csoportot jelenthetnek a kölcsönző ügyfelei, akik a rendszer webes felületének elsődleges felhasználói lesznek, azonban a megszólításuk talán a legnehezebb. Az általuk támasztandó követelményeket valószínűleg a vezetőség fogja vázolni, és az alkalmazásuk után fog kiderülni, hogy ezek az igények mennyiben váltak be vagy sem.

Ha az előző fejezet példáiból indulunk ki, akkor például velük kapcsolatban tisztázni kell az interjúk során, hogy milyen jellemzőit kell rögzíteni az egyes csónakoknak, melyek azok amelyek a felhasználói keresésben is részt kell hogy vegyenek, milyen technikai jellegű további adatokat tárolnak a csónakokról (pl. szervizelési idő). Továbbá hogyan történik egy csónak kikölcsönzése, milyen üzleti szabályokat kell figyelembe venni, ehhez mit kell a rendszernek nyilvántartani stb. Ezekre a kérdésekre az interjúk során kaphatunk választ, mert az üzleti folyamatok menetét azok az alkalmazottak ismerik a legjobban akik nap mint nap ezt végzik a munkájuk során.

Természetesen az én alkalmazásom esetében nem voltak tényleges interjúk, hiszen nem volt kit megkérdeznem, de a kezdeti ötletek leírása után el kellett gondolkoznom, hogyan szeretném részleteiben megvalósítani a rendszert, és szükséges volt többek között a fenti kérdések tisztázása is.

3.4 Fogalomszótár

Korábban említést tettem arról, hogy az interjúk egyik problémája, hogy a különböző szakterületi szakértők az adott terület zsargonját használják, mely nehézkessé teszi a folyamatok megértését a kívülálló követelményelemzők és tervezők számára. Ezért a követelményelemzés következő lépéseként a fogalomszótárat kell összeállítani. Erre azért van szükség, mert így az alkalmazási szakterületen előforduló egyes szakkifejezések kifejtésre kerülnek, melyek nem ismertek az üzleti elemzők és a tervezők számára. Tehát a megrendelőnek meg kell magyaráznia, hogy a szakterület egyes fogalmai pontosan mit jelentenek, milyen a rendszer szempontjából fontos adatokkal rendelkeznek, illetve mely üzleti folyamatokkal vannak kapcsolatban. A későbbiekben a dokumentumban szereplő fogalmakat lehet használni egyéb leírásokban és magyarázatokban, a fogalomszótár biztosítja, hogy mindenki ugyanazt értse a fogalomszótárban szereplő kifejezések alatt.

Ebben a dokumentumban is megjelennek a rendszer kulcsfigurái. Mivel ők támasztanak követelményeket a rendszerrel szemben, ezért fontos a definiálásuk, és annak a tisztázása, hogy hogyan akarják felhasználni a rendszert, és milyen módon tudnak majd hatással lenni a rendszer működésére. Ezen túl a rendszerrel kapcsolatos főbb adattípusokba (mint a jelen esetben pl. regisztrált ügyfél, csónaktípus, kölcsönzési szerződés, foglaló stb.) is betekintést nyújt ez a dokumentum, hiszen a kulcsfigurák érintkezése a rendszerrel mindenképpen adatokat továbbít a rendszer felé, és onnan kifelé is, és ezeknek az adatoknak az áramlása valamilyen adatszerkezeten keresztül lehetséges. Ezentúl már itt felszínre jönnek, hogy milyen folyamatok vannak a háttérben, a fogalmak meghatározása hozzákapcsolja őket azokhoz a tevékenységekhez, amelyekben részt vesznek az említett kulcsfigurák, de ezek leírása nem erre a dokumentumra tartozik.

A tárgyalt rendszer esetében a fogalomszótár megmagyarázza, hogy kik a cég ügyfelei, a rendszer szempontjából hogyan kategorizáljuk őket (regisztrált és nem regisztrált ügyfél), ezekhez a kategóriákhoz milyen jogkörök tartoznak, mi a kölcsönzés, a kölcsönzési szerződés, hogyan kell számítani a foglalót, mennyi lehet a kölcsönzési idő, milyen típusú és állapotú motorcsónakok vannak, mi a késedelmi díj, a szervízkorlát és a körlevél. Látható, hogy ezek között lehetnek olyan dolgok, amelyek teljesen ismeretlenek, míg mások egyértelműnek tűnnek, de közelebbi vizsgálatukkor kiderül, hogy mennyi mindent tisztázni kell velük kapcsolatban.

2. példa: Fogalomszótár

Nem regisztrált ügyfél
Olyan személy, aki nem rendelkezik regisztrációval, és a cég honlapját meglátogatva kereshet a kölcsönözhető csónakok között, ezek adatait megtekintheti, és regisztrálhat.

Regisztrált ügyfél
A regisztrált ügyfélnek van azonosítója, neve, lakcíme (ország, város, irányítószám, utca, házszám, ajtó, emelet), számlázási címe, személyigazolvány száma, bankkártya száma, adószáma, email-címe, telefonszáma. A regisztrált ügyfél kereshet a kölcsönözhető csónakok között, ezek adatait megtekintheti, és a honlapon való azonosítás (belépés) után a kiválasztott csónakokra előjegyzést tehet, illetve megtekintheti korábbi kölcsönzéseinek listáját. A regisztrált ügyfél önmaga is megszüntetheti regisztrációját, azonban ha egy ügyfél adatai egyszer a cég adatbázisába kerültek, a cég azt mindvégig megőrzi, azok csak az ügyfél hivatalos kérése esetén távolíthatóak el az adatbázisból.

Kölcsönzési szerződés
Olyan szerződés, mely kimondja, hogy az ügyfél az általa választott csónak használati jogát a kölcsönzési idő által meghatározott időtartamra megkapja. Amennyiben a csónak kölcsönzés közben meghibásodik, az ügyfélnek csak a visszahozatal napjáig kell kifizetni a kölcsönzési díjat.

Szervízkorlát
Az az utolsó szervizelés óta eltelt időtartam, ami után a csónakot szervízbe kell küldeni: minden századik kölcsönzés vagy negyven nap kölcsönzés után.

Körlevél
A regisztrált ügyfelek bizonyos időszakonként e-mailben értesítést kapnak a cég aktuális híreiről, akcióiról. A körlevél összeállítását és kiküldésének meghatározását az alkalmazott végzi.

3.5 Szakterületi folyamatok és kapcsolatok

A fogalomszótárat kiegészítő dokumentum a szakterületi folyamatok és kapcsolatok leírását tartalmazó dokumentum. Ebben vannak rögzítve a rendszer alkalmazási szakterületéhez köthető követelmények, melyek lehetnek funkcionális követelmények, és az ezekre vonatkozó megszorítások, illetve az egyes funkcionalitásokhoz kapcsolódó műveletsorok, számítások leírásai. Gyakorlatilag a szakterületen fellelhető, a rendszer működéséhez szükséges folyamatok absztrakt leírása, mely az előző részben már említett folyamatok és fogalmak közötti kapcsolatot teremti meg, hiszen ez konkrétan leírja, hogy az egyes folyamatokban mely kulcsfigurák vesznek részt, a folyamatok hogyan zajlanak, milyen lépésekből tevődnek össze, az egyes lépéseknek milyen előfeltételeik vannak, és hogyan hatnak a rendszer bizonyos részeire.

3. példa: Szakterületi folyamatok

Előjegyzés
A regisztrál ügyfélnek lehetősége van egy kölcsönözhető csónakot előre lefoglalni. Az előjegyzésnek van kezdete és előrelátható időtartama. Egy előfoglalt csónakot kikölcsönözni a lefoglaló ügyfélnek van elsőbbsége. Amennyiben a lefoglaláskor megadott napon nem kölcsönzik ki a csónakot, az előfoglalás érvényét veszti.

Kikölcsönzés
Tagsági szerződéssel rendelkező ügyfél legfeljebb 3 csónakra köthet bérleti szerződést. Az ügyfél ügyfélszámának megadását követően megjelennek az ügyfél adatai. A kölcsönözendő csónakok és kölcsönzési időtartam megadását követően a rendszer kijelzi a befizetendő összeget. A befizetést követően a kölcsönzés elkezdődik.

Visszahozatal
Az ügyfél a kikölcsönzött csónakok visszahozásakor az alkalmazottnak megadja az ügyfél-azonosítóját és hogy mely csónakokat hozta vissza. A rendszer megvizsgálja, hogy szükséges-e késedelmi díjat fizetni, ha igen, akkor ennek összegét megjeleníti. A befizetendő késedelmi díj kiszámítása a kikölcsönzéskor befizetett foglaló figyelembevételével történik. Ha nem kell késedelmi díjat fizetni, akkor az alkalmazott visszaadja a befizetett foglalót. Ha a csónak meghibásodott, akkor ezt az alkalmazott szervízelendőnek nyilvánítja a motorcsónakot.

A fenti példában látható, hogy a kölcsönzés folyamatában megjelenik az ügyfél, a csónak és kölcsönzés fogalma, illetve, hogy ezek hogyan függnek össze a kikölcsönzés folyamatában. Látható, hogy a függéseket is, hogy az ügyfélnek tagsági szerződéssel kell rendelkeznie, hogy az ügyféladatok az ügyfélszáma alapján kereshetőek, vagy hogy a kölcsönzés megkezdéséhez a kölcsönzendő csónakokat és a kölcsönzés időszakot kell megadni.

3.6 Forgatókönyvek

A fentebb ismertetett szakterületi folyamatok leírásával az a probléma, hogy nehéz megfogalmazni őket, ugyanis az egyes emberek, kulcsfigurák könnyebben kezelik a valós életbeli problémákat, mint az absztrakt leírásokat. Éppen ezért a követelmények feltárásakor sokszor alkalmazzák azt a technikát, hogy forgatókönyveket készítenek, melyekben leírják, hogy egy adott probléma megoldása milyen konkrét lépéseken keresztül végezhető el. Ezen leírások alapján a kulcsfigurák képesek megérteni, hogy az adott szituáció megvalósítása közben hogyan érintkeznek majd a rendszerrel, és emiatt a felmerülő problémáikat és kritikáikat is meg tudják fogalmazni. Az így felmerülő információkat a követelménytervezők nagyon jól tudják hasznosítani a rendszerkövetelmények finomításakor. Tehát a forgatókönyvek segítenek további részletekkel kiegészíteni a már eddig körvonalazott és dokumentált követelményeket.

A forgatókönyv tehát nem más, mint egy leírt interakció-sorozat, mely rögzíti, hogy a forgatókönyvben szereplő kulcsfigurák milyen lépéseket tesznek, és ezeknek milyen hatásuk van a rendszerre. Minden forgatókönyv egy vagy több lehetséges interakciót ír le. Az elkészítése ezen interakciók körvonalazásával kezdődik, a feltárás alatt további részletekkel bővítjük, hogy végül az interakció teljes leírása megszülessen.

A részletezés egy egyszerűbb módja, ha minden egyes használati esethez megadunk egy vagy több forgatókönyvet, amelyben felsoroljuk, hogy a funkció milyen, az adott kulcsfigura és az alkalmazás között lezajló párbeszédet igényel. A forgatókönyvek megadásakor célszerű a párbeszédre, az interakciókra helyezni a hangsúlyt. A forgatókönyvek így olyan pontokból állnak, amelyek vagy a rendszer felé történő adatszolgáltatást, vagy a rendszer adatszolgáltatását, például eredményközlését írja le. A „center-out" elv alapján először a normálnak, tipikusnak tekintett forgatókönyveket vegyük fel, majd ezután térjünk ki a speciális, például a hibás esetek párbeszédére.

Egy forgatókönyv az alábbiakat tartalmazhatja:

1. egy előfeltétel leírása, hogy az interakció milyen feltételek mellett indul (ennek segítségével megtehetjük azt hogy az egyes feltételeket egymásba ágyazzuk, így nem lesznek túlságosan terjengősek az egyes forgatókönyvek)
2. az interakció-sorozat eseményeinek leírása normális esetben
3. leírás a kivételes eseményekről, mit lehet az adott eseménysorozat során elrontani, ezekre hogyan kell reagálni
4. a rendszer állapotának leírása a forgatókönyv végrehajtása után.

A forgatókönyvek nagy előnye, hogy informális leírások, így könnyen érthetőek a kulcsfigurák számára, illetve elkészítésük során a követelményelemzők együtt dolgozhatnak ezekkel a kulcsfigurákkal. Így könnyen azonosíthatók a szükséges forgatókönyvek, és azok részletei. Ezek a dokumentumok általában szövegesek, de szükség esetén kiegészíthetőek diagramokkal, képernyőképekkel és magyarázó folyamatábrákkal.

4. példa: regisztráció forgatókönyve

Regisztráció
Kiinduló feltétel: *a felhasználó a böngészőjében betöltötte a honlap főoldalát.*

Normális működés*: A felhasználó a regisztráció linkre kattintva az ügyfél regisztrációja oldalra kerül. Ott kitölti a lapon található mezőket, majd a lap alján található regisztráció gombra kattinthat a regisztrációhoz, vagy a Mégse gombra a művelet visszavonásához, ezzel a főoldalra jut vissza. Ha a felhasználó a regisztrációra kattintott, akkor a rendszer ellenőrzi a megadott adatok helyességét, és amennyiben az adatok rendben vannak, akkor rögzíti az adatbázisban, majd kiküldi a szükséges megerősítő e-mailt a megadott címre. A felhasználónak megjelenít egy üzenetet, mely megerősíti, hogy a regisztrációja megtörtént, és hogy a megerősítést el kell végeznie az elküldött e-mail utasításai szerint. A felhasználó egy link segítségével a főoldalra juthat vissza.*

Hibás esetek*: a felhasználónak ki kell töltenie minden kötelező mezőt, ha ez nem történt meg, akkor a rendszer hibaüzenettel tér vissza a regisztrációs oldalra. A megadott e-mail címnek érvényesnek kell lennie, amennyiben ez nem teljesül, szintén hibaüzenettel tér vissza. A megadott jelszónak és a megerősítő mező tartalmának egyeznie kell. A megadott felhasználónévvel vagy e-mail címmel még nem lehet nyilvántartva egyetlen felhasználó sem. Rendszerszintű hiba lehet, ha a megerősítő e-mailt nem lehet elküldeni, vagy ha az adatbázis nem érhető el. Ezekben az esetekben egy általános hibaoldalra jut a felhasználó.*

Rendszerállapot: *a műveletsor elvégzése után a felhasználó a főoldalon van, az adatbázisban egy új regisztráció található, amely „nincs megerősítve” állapotban van. Továbbá ugyanehhez a regisztrációhoz él egy megerősítést azonosító kódsor, amely alapján a kiküldött e-mail azonosítható lesz.*

A fenti példában jól látható a forgatókönyv egy használati esetre vonatkozó leírása: megfogalmazza a szükséges előfeltételt, ezzel elkerülhetjük, hogy nagyon körülményes legyen az egyes esetek leírása (ld. a későbbi példában), majd egy műveletsort fogalmaz meg. Ezek nagy része felhasználói interakció, melyek arról szólnak, hogy a felhasználónak hova kell kattintania, és miket kell kitöltenie, ezek mellett pedig néhány megjegyzés is található azzal kapcsolatban, hogy a felhasználói interakciókra hogyan fog reagálni a rendszer. Az első leírásban feltételezzük, hogy minden rendben történik, ugyanis először mindig érdemes a normális működést tisztázni, majd a későbbiekben kitérni azokra a pontokra, ahol hibás módon térhet el a végrehajtástól a rendszer. Ezeket az eseteket taglalja a harmadik rész. Végül érdemes megjelölni, hogy a művelet sikeres végrehajtása esetén milyen változások állnak be a rendszerben.

5. példa: előjegyzés forgatókönyve

Csónak előjegyzése
Kiinduló feltétel: *a regisztrált felhasználó be van jelentkezve az oldalra, és egy csónaktípus adatlapját töltötte be a böngészőjébe.*

Normális működés: *a csónaktípus adatlapján a felhasználó az előjegyzés linkre kattint. Ekkor az oldal alján megjelenik az előjegyzéshez szükséges panel, amelyen meg kell adnia, hogy mely dátumok között akarja kikölcsönözni egy csónakot az adott típusból. A rendszer ellenőrzi, hogy a felhasználónak lehetséges-e újabb csónakot előjegyezni, illetve a megadott időszakra van-e olyan csónak, amelyre még nincs előjegyzés. Ha van ilyen, akkor megjelenik az előjegyzés rögzítése gomb. A felhasználó erre kattintva rögzítheti az előjegyzést, a rendszer nyugtázásként az előjegyzések oldalra viszi tovább, ahol látható, hogy az előjegyzés tényleg sikeresen megtörtént.*

Hibás esetek: *a felhasználónak már van 3 előjegyzése, ekkor nem rögzíthet újabb előjegyzést, a megadott időpontok nem megfelelőek, tehát a kezdőidőpont később van, mint a végidőpont, illetve nincs olyan csónak, amelyet előjegyezhetnénk az adott időszakra. Ezekben az esetekben a rendszer a megfelelő hibaüzenettel adja tudtára a felhasználónak az előjegyzés sikertelenségének okát.*

Rendszerállapot*: a sikeres művelet hatására a rendszer rögzíti az adott típusból az egyik szabad csónakra az előjegyzést, és azt megjeleníti a felhasználó számára.*

3.7 Használati esetek

A használati esetek a rendszer funkcionális követelményeinek feltárásának forgatókönyv-alapú eszközei. Egy grafikus eszköz a, mely leírja a folyamat elsődleges résztvevői, mint az interakció elindítói és a rendszer közötti interakciót, az egyes lépések sorozatának ábrázolásával. A folyamatok résztvevőit aktoroknak nevezi, melyeket pálcikaemberkékkel reprezentál, az interakciókat pedig ellipszisek jelölik, melyek az interakció nevével vannak címkézve. A használati esetek halmaza minden, a rendszerkövetelményekben megjelenítendő interakciót reprezentál.

Az aktorok a már korábban feltárt kulcsfigurák közül kerülnek ki, és az egyes használati esetek egy vagy több forgatókönyvet írnak le, de mindig leírják egy esemény teljes lefolyását az résztvevő aktor szemszögéből nézve. Így a használati esetek lehetnek egészen egyszerűek is, melyen csak egy aktor és a főbb tevékenységek vannak feltüntetve, míg egészen bonyolultak is, melyben korábbi használati eseteket fejthetünk ki. Ezeket az interakciókat érdemes úgy létrehozni, hogy egy adott tevékenységgel kapcsolatos aktorokat határozzuk meg először, aztán pedig felsoroljuk az aktor által elvégezhető fontosabb tevékenységek összefoglaló neve-

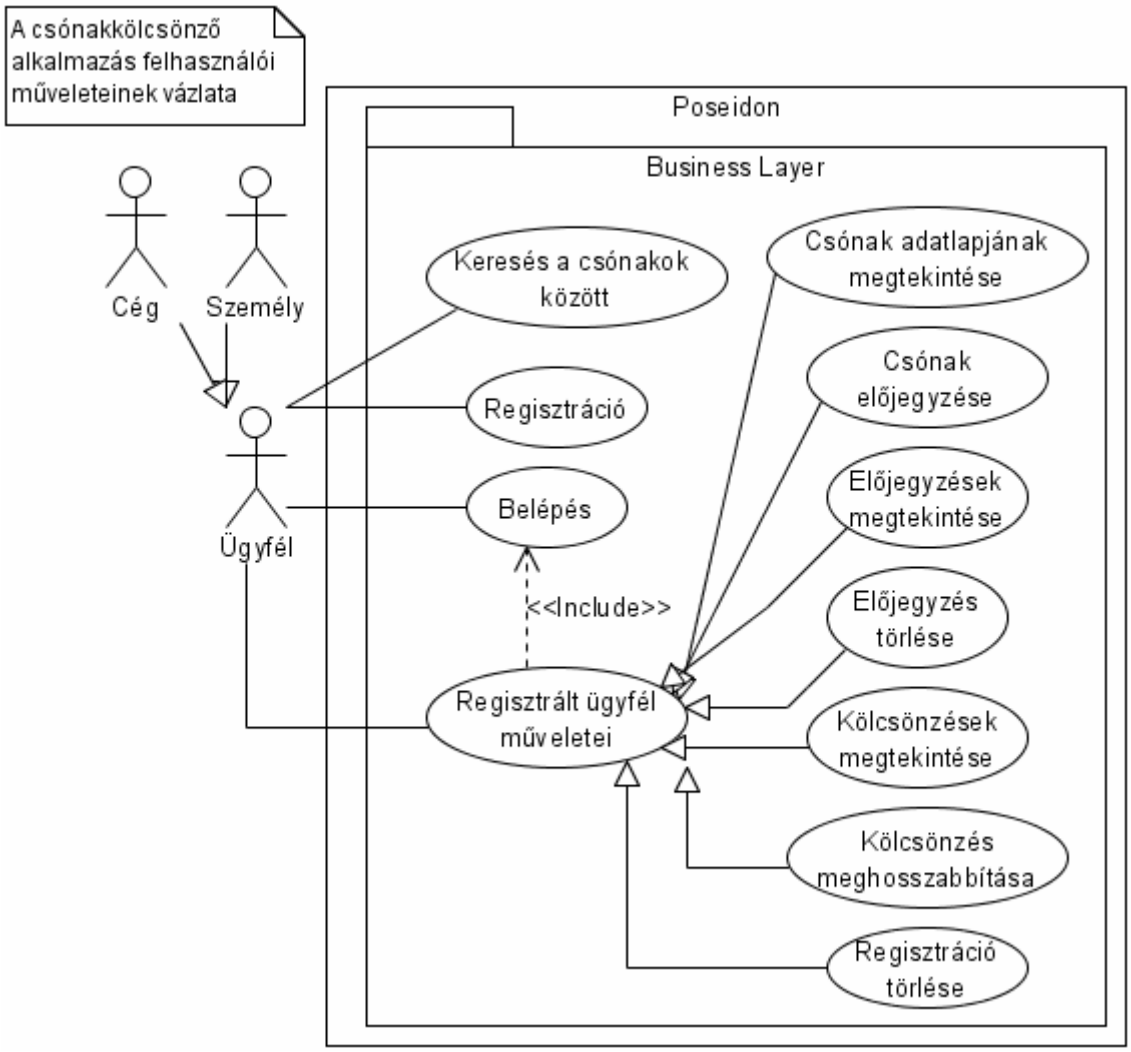
it. Majd ez után, az egyes „buborékokat” részletezzük tovább újabb használati eset diagramok segítségével, melyeken már feltüntetünk köztes lépéseket is, és az egyes lépéseket addig finomítjuk amíg szükséges. Továbbá ésszerű először csak a normális esetekkel foglalkozni, amelyben feltételezzük, hogy semmilyen hiba nem történik sem az aktor, sem a rendszer részéről, ezeket a hibás eseteket külön diagramon érdemes jelölni részletesebben.

A diagramokon megjelenő elemek között különböző kapcsolatok jelenhetnek meg, mint az asszociáció, a generalizáció, a tartalmazási és kiterjesztési kapcsolat. Ezek jelentését és a kapcsolatok kialakításának mechanizmusát itt nem részletezem, ezek az UML szabvány részei, mely szabvány pontosan leírja ezen relációk jelentését és alkalmazási körüket. A diagramokon opcionálisan megjelenhetnek olyan elemek is amelyek a rendszer határát hivatottak jelölni, illetve a rendszeren belül valamilyen logika szerint csoportosítja az egyes használati eseteket. Ez általában funkcionális szempont szerinti csoportosítás szokott lenni, mely már az ilyen kezdeti időkben is tükrözi a rendszer leendő architekturális tagoltságát. Mivel a használati eset diagramok nem tükrözik tisztán az egyes tevékenységek közötti időbeli összefüggéseket, ezért sokszor ezek kiegészítéseként ún. szekvencia diagramokat szoktak alkalmazni, melyek kezdetleges módon feltüntetik az aktort és az interakcióban résztvevő rendszerobjektumokat, és jelölik hogy ezek milyen módon érintkeznek időrendi sorrendben elhelyezve, illetve milyen műveletek kapcsolhatóak az egyes interakciókhoz. Természetesen ezek még csak elemzési szinten, körvonalazott szinten írják le ezeket az objektumokat és műveleteket, ezek csak a tervezési fázisban fognak konkretizálódni.

A használati esetek előnye, hogy velük hatékonyan tárhatóak fel az aktor-nézőpontok olyan követelményei, amelyekben mindegyik fajta interakció ábrázolható használati esetként. Továbbá grafikus megjelenésükből következően nyelvtől független leírások, így könnyű információt cserélni különböző anyanyelvű résztvevők között is, illetve a grafikus jellegből következően egy követelményelemzéshez nem értő személy is könnyen megértheti és áttekinteti az egyes rendszerfunkciók működését, hiszen a diagramok megértéséhez minimális magyarázat szükséges. Így az elemzés ezen fázisába is könnyen bevonatóak a kulcsfigurák. A használati esetek a tervezők számára pedig betekintést engednek a különböző üzleti folyamatok közötti összefüggésekbe, láthatják, hogy az egyes részek hogyan hatnak majd egymásra, milyen alternatív interakciók vannak az egyes útvonalakon, így növelhetik a rendszer robosztusságát. A rendszer felhasználói felületének tervezői is felhasználhatják ezeket a leírásokat, hiszen az egyes használati esetek leírják, hogy milyen lépésekre, interakciókra van szükség, és

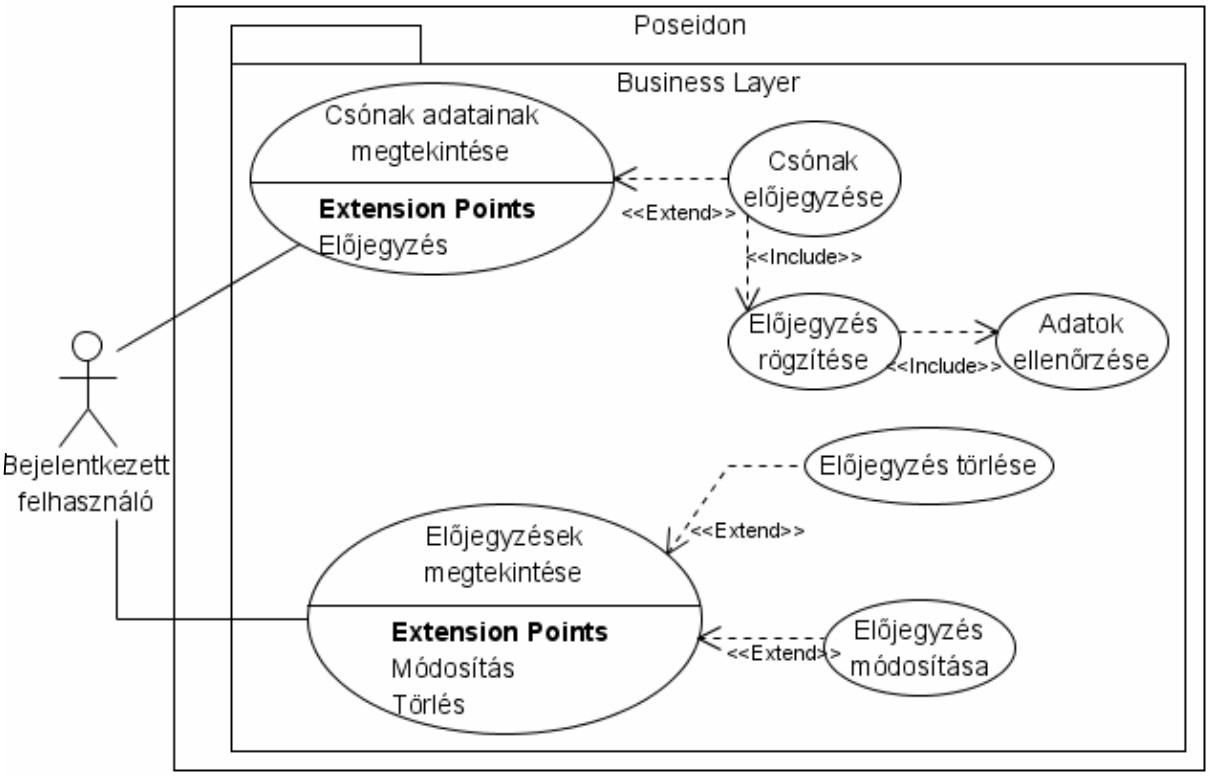
ezek az interakciók nagy valószínűséggel valamilyen felhasználói felületen keresztül fognak végbemenni.

6. példa: használati eset diagram (vázlatos)



A fenti példában jól látható, hogy egy egyszerű használati eset diagram is milyen kifejező lehet, hiszen a fenti diagram csak egy vázlatos felsorolása annak, hogy a felhasználók milyen műveleteket végezhetnek el az alkalmazás segítségével. A kezdetek során ebből is kiindulhatunk, és finomíthatjuk tovább az egyes esetek (buborékok) részletezésével, a köztük rejlő összefüggések feltárásával. Ilyen például, hogy az előjegyzések megtekintése átmehet törlésbe is, a kölcsönzések megtekintése átmehet meghosszabbításba stb. A következő példában az előjegyzésekkel kapcsolatos használati eseteket láthatjuk.

7. példa: előjegyzés használati esete



Azonban a használati esetek használhatósága nagyban függ attól, aki készítette a diagramot, a leírások elkészítése nem jelenti automatikusan a tiszta megfogalmazás követelményének teljesítését. Továbbá mivel ezek a diagramok az interakciókra koncentrálnak, nem hatékonyak a közvetett nézőpontokból származó megszorítások vagy magas szintű üzleti és nemfunkcionális követelmények feltárására, valamint a szakterületi követelmények felderítésére.

3.8 Működési leírások

A forgatókönyvek segítségével olyan használati eseteket írhatunk le, amelyekben a vezérlést alapvetően a felhasználói interakciók határozzák meg. Így nem alkalmasak olyan funkciók ábrázolására, amelyek viszonylag kevés interakcióra, ugyanakkor több és összetettebb tevékenységsorozatra épülnek.

A több, önálló lépésben lezajló működések a tevékenységsorozatok leírásával vázolhatók. A tevékenységsorozat egyben egy időbeli egymásutániságot is jelent, így megadhatjuk, hogy melyek az érvényes sorrendek. A működési leírások különösen akkor hasznosak, ha a használati eset valami olyan folyamatot indít be, amely az aktortól részben független.

A működési leírás hasonlít forgatókönyvekre, mert itt is az egyes pontokban tevékenységeket (vagy más néven: műveleteket) adhatunk meg. Az összetett résztevékenységeket vagy a más feltételek mellett lezajló eltérő működést strukturálással vagy külön működési leírással jelezhetjük. Először a normális működéseket adjuk meg, majd ezután a speciális, például a hibás eseteket.

A leírás során célszerű a tevékenységeknek csak a vázolására szorítkozni. Minél kevesebb műveletet adjunk meg, kizárólag azokat, amelyek valóban lényegesek a felhasználó szempontjából. Fontos, hogy a kulcsfigurák fogalmait használjuk és ne az informatikai nyelvezetet.

3.9 Felhasználói felületek

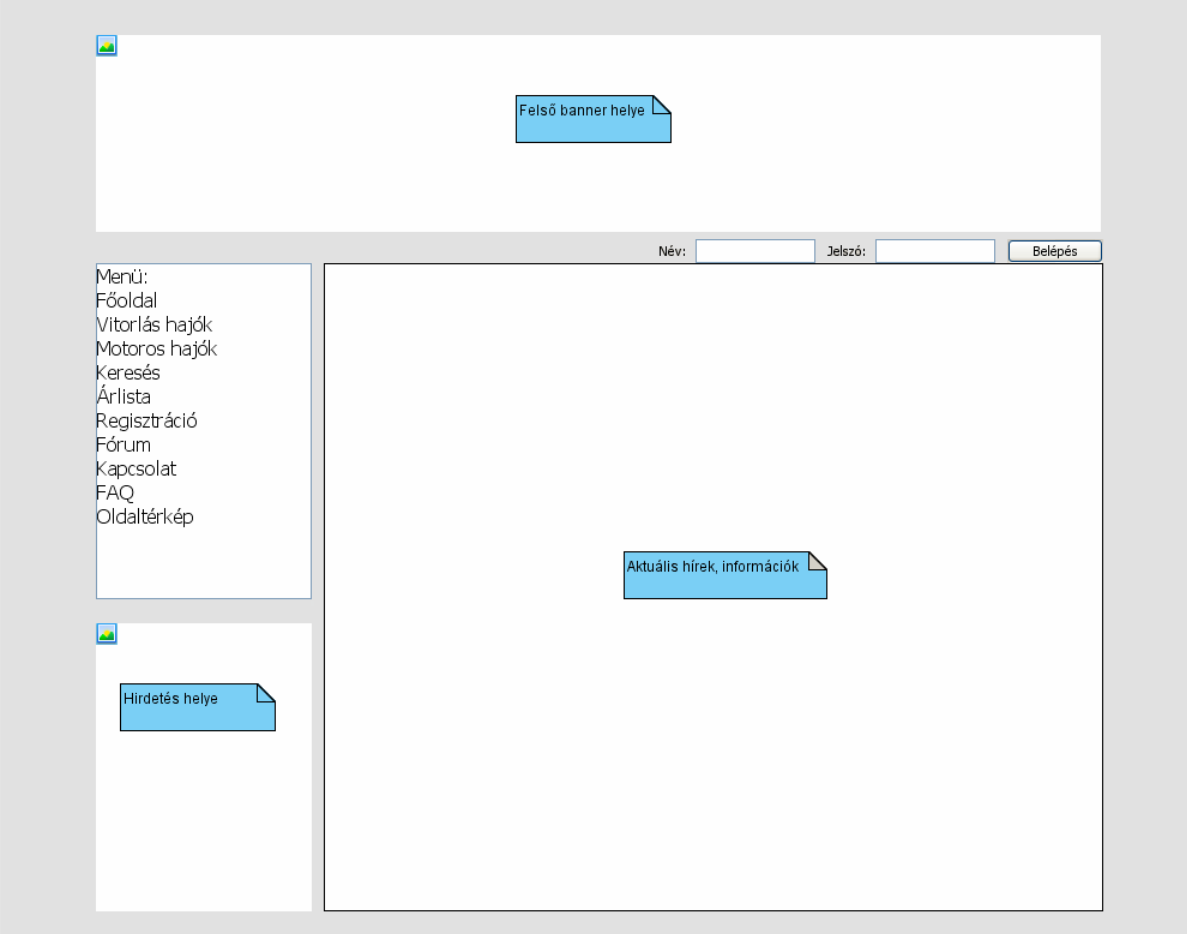
A forgatókönyvekkel, illetve a működési leírásokkal pontosíthatjuk az egyes használati esetek lépéseit, azaz a felhasználóval történő párbeszédet vagy a végrehajtandó tevékenységsorozatokat. A forgatókönyvek egyes elemei az aktorként megjelenő felhasználóval vagy külső rendszerrel történő információátadást írnak le. A használati esetek pontosításának következő lépéseként célszerű megtervezni az információátadás eszközeit, a felhasználói felületeket.

A „felhasználói felület” (user interface) kifejezést általános értelemben kell érteni, azaz a külső rendszerek is a rendszer felhasználói, melyekkel az adatcsere speciális módon, például adott szerkezetű fájlok felületén keresztül is történhet. Azonban világos, hogy ezen felületek nagyobb részét (bizonyos speciális eseteket kivéve) valamilyen ember által használt vizuális interfész fogja kitenni, hiszen a rendszerek működtetését főképp emberek végzik, illetve emberi felhasználásra készítik őket.

A követelmények elemzésekor természetesen csak a felhasználói felületek vázlatairól van szó, hiszen azok tényleges megvalósítása már a tervezés és az implementálás feladata. Azonban láthattuk, hogy az eddigi dokumentumok nagy része is a rendszer kulcsfiguráival való interakciók leírásán alapult. Egyértelmű, hogy ezen interakciók javarésze valamilyen felhasználói felületen keresztül fog végbemenni, és az esetek többségében ez grafikus megjelenítést igényel. Így célszerű az egyes használati esetekben és a forgatókönyvekben is megfogalmazott interakciókhoz egy felületvázlatot készíteni. Ezen csak annyit kell feltüntetni, hogy milyen felületelemek tűnnek fel az egyes képernyőkön, ezek milyen elrendezésben tárják a felhasználó elé a rendszerből jövő adatokat, és azok hogyan fognak megjelenni, hogy az egyes

beviteli adatokat milyen felületelemeken, mezőkön keresztül tudjuk megadni, illetve milyen gombokra, linkekre tudunk majd kattintani, hogy egy következő felületre jussunk.

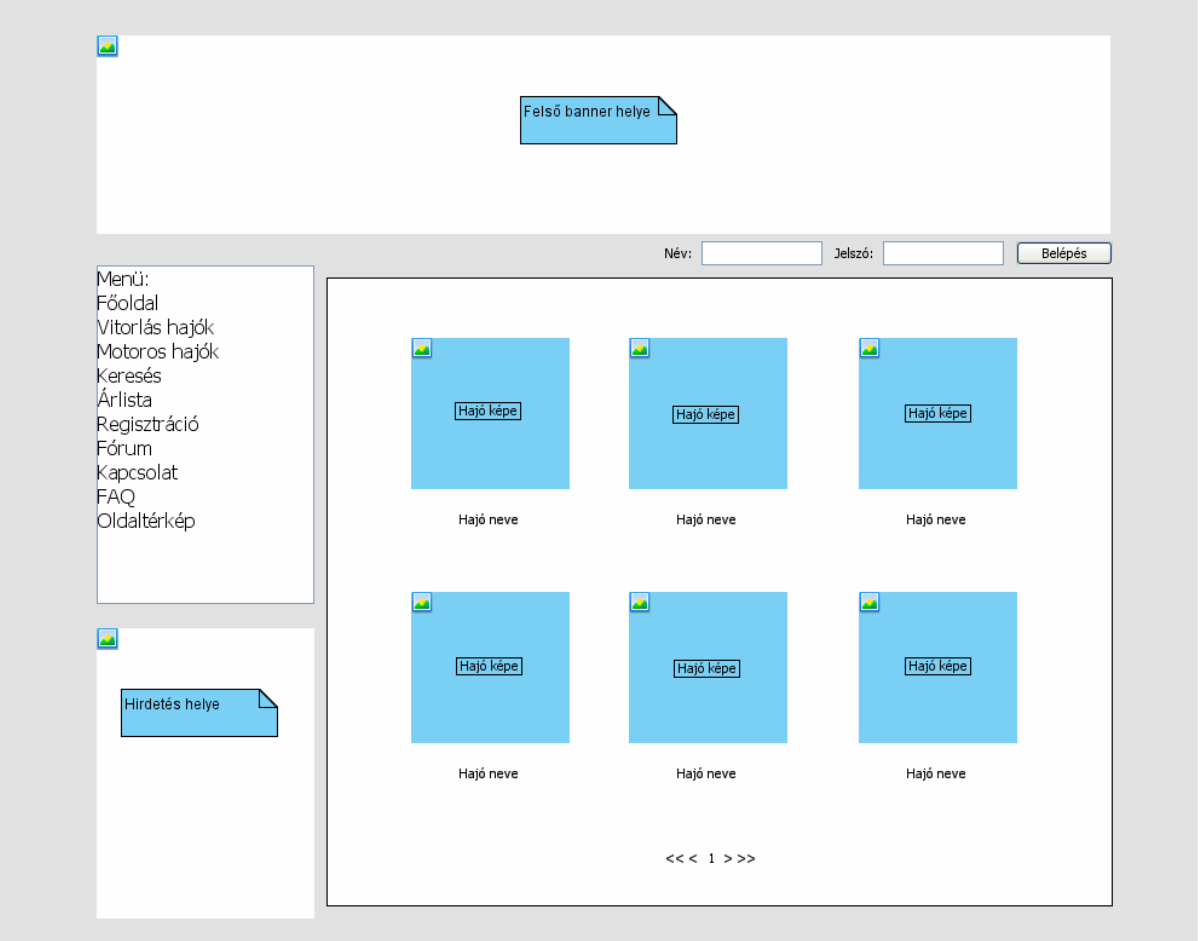
8. példa: Főoldal (felületvázlat)



Mindennek ott jelenik meg a haszna, hogy sokszor bizonyos követelmények csak akkor jönnek elő, amikor látjuk, hogy bizonyos interakciókhoz további dolgok lennének szükségesek, másrészt pedig nem csupán a vázlatok elkészítése a lényeg, hanem az, hogy egy összefoglaló diagramon azt is feltüntetik, hogy az egyes felületek hogyan kapcsolódnak egymáshoz, mikor milyen gombra tudunk kattintani, és azok milyen új felületekre visz tovább. Mindezeknek tükröződnie kell a vonatkozó forgatókönyvek leírásában is, hiszen ezek is az egyes interakciók egymásutániságát írják le. Mindezeken túl itt is megjelennek bizonyos üzleti logikához köthető követelmények, mint például hogy egy felhasználónak milyen hosszú kell legyen a felhasználóneve és a jelszava, milyen karaktereket kell tartalmaznia, mi jelenik meg egy sikertelen bejelentkezési kísérlet után stb.

Látható, hogy egy webes alkalmazás esetén a felületek elemzése létfontosságú, hiszen a rendszer felhasználó a cég ügyfelei ezzel a felülettel fognak elsődlegesen találkozni, ezen keresztül fogják igénybe venni a szolgáltatásokat, és ez alapján fogják megállapítani az adott cégről a véleményüket. Továbbá a laikus megrendelő számára sokszor ez az elsődleges, hogy lássa, miképpen lesz felhasználható a készülő rendszer, hiszen minél hamarabb eredményt szeretne látni. Egy webes alkalmazás fejlesztése során könnyen megtehetjük azt, hogy a felületek elemzéséhez statikus HTML oldalakat gyártunk, erre rengeteg üzleti szoftver létezik már a piacon, melynek segítségével kevés erőfeszítéssel „összekattintgathatunk" felületeket, és azok összelinkelésével akár még szimulálhatjuk is annak működését, melyet ugyan design nélkül de megmutathatunk a megrendelőknek, korlátozott módon, de kipróbálhatja és elmondhatja véleményét azzal kapcsolatban. Természetesen egy asztali alkalmazás esetén is kevés erőfeszítés árán lehet ilyen vázlatokat készíteni, hiszen a mai piaci tervező szoftverek támogatják a különböző felületek készítését, bár azok csak bemutatásra készülnek, működni nem fognak.

9. példa: csónakok böngészése (felületvázlat)



3.10 Szoftverkövetelmények dokumentálása

Az elvégzett követelményelemzés eredményeit a későbbi felhasználhatóságuk érdekében pontosan dokumentálni kell. A rendszer megvalósításához egyetlen nagy dokumentumban össze kell foglalni a szoftver követelményeit, amelyet a tervezők fognak felhasználni, és mely alapján közvetett módon a szoftverfejlesztők is fognak dolgozni. Ezt a dokumentumot szokták a szoftverkövetelmények dokumentumának hívni, vagy nevezik még szoftverkövetelmény-specifikációnak is. Ez egy hivatalos és formális leírása a rendszernek, melynek tartalmaznia kell a rendszer felhasználói követelményeit – tehát a funkcionális és nemfunkcionális követelményeket – és a rendszerkövetelményeket.

Ez a leírás sokszor összefoglalható egyetlen dokumentumban is, de ez csak kisebb rendszerek esetében tehető meg. Látható, hogy az elkészítendő leírások száma nagy és ezeket mind magában kell foglalnia valamilyen formában a szoftverkövetelmények dokumentumának, ezért az áttekinthetőség érdekében egy nagy rendszer esetében érdemes több dokumentumra szétszedni ezt a terjedelmes leírást.

Ez a leírás kulcsszerepet játszik a fejlesztés további lépéseiben, több okból is: egyrészt ezt kapja meg a megrendelő, ez alapján fogja eldönteni, hogy az elkészítendő rendszer kielégíti-e majd az igényeit, és megvalósítja-e azokat a kívánságait, amelyekre az adott szakterületi környezetben használni kívánja. Másrészt ha az elvárásainak megfelel a dokumentum, akkor ezt fogja aláírni, és a későbbiekben erre hivatkozhat mindkét fél, tárgyalási alapként: a megrendelő, ha szerinte valamit nem valósítottak meg, amit kért, és a fejlesztők is, ha valami olyat követel rajtuk a megrendelő, amiben nem egyeztek meg. Az utóbbi azért is lényeges, mert a szoftver elkészítése pénzbe és időbe kerül, melyet még a projekt elején rögzítenek, és ezek erősen befolyásolják, hogy mit lehet megvalósítani és mit nem. Tehát valószínűleg nincs sem idő, sem pénz olyan funkcionalitások megvalósítására, amelyek nem voltak előre rögzítve. Természetesen amennyiben az ügyfél ragaszkodik bizonyos módosításokhoz, akkor azok megvalósítása megtörténhet, de az már további összegek ráfordítását igényli a részéről.

A másik nagyon fontos indok, hogy a rendszer tervezői ezen dokumentum alapján ismerkednek meg a feladattal, értik meg, hogy mit kell megvalósítani a rendszer keretén belül, és tervezik meg annak implementálását, illetve lehetséges kiterjesztési pontok meghatározásához ad lehetőséget, tehát már a rendszer tervezésekor felkészülhetünk arra, hogy bizonyos újabb funkciókat kell majd bevezetni a későbbiekben, egy másik projekt keretében. Ez az előrelátó gondolkodás elengedhetetlen és nagyban megkönnyíti a későbbiekben a szoftver evolú-

cióját. Végül pedig a rendszer tesztelői is ebből a leírásból fognak kiindulni, hiszen ezen követelmények alapján kellesz majd verifikálni az elkészült szoftvert, tehát azt megállapítani, hogy az alkalmazás tényleg azt csinálja, amit a megrendelő elvár tőle.

Ezen dokumentum formai megszorításaira különböző szabványok léteznek, az Amerikai Védelmi Minisztérium és az IEEE is definiált ilyen szabványos dokumentumvázakat (pl. IEEE/ANSI 830-1998-as szabvány), melyek megadják ezen leírások szerkezetét. Ezekre a szabványokra támaszkodva mindenki kialakíthatja a maga formátumát, általában nagyobb cégek esetében a sokéves tapasztalatoknak megfelelően adnak hozzá vagy vesznek el részeket, illetve magától a megoldandó feladattól is függ, hogy hogyan néznek ki ezek a dokumentumok, hiszen bizonyos részek lehetséges hogy értelmetlenek vagy feleslegesek lennének, míg mások további felbontást és részletességet kívánnak, a probléma specialitásából következően.

Az elkészítéskor azt is figyelembe kell vennünk, hogy ezen dokumentum felhasználói változatosak, ezt láthattuk, hiszen a megrendelő cég vezetőitől és elemzőitől kezdve a szoftver fejlesztéséért felelős tervezőkig többen is olvassák ezt a dokumentumot, és mindannyiuknak meg kell érteniük ezt a leírást. Ez bizonyos kompromisszumot jelent: tájékoztatnia kell a megrendelőket érthető formában a rendszer követelményeiről, de eléggé szabatosnak és részletesnek kell lennie ahhoz, hogy azt a tervezők és a tesztelők felhasználhassák a szoftver elkészítésekor.

Bárhogyan is valósítjuk meg ezt a leírást, mindenképp szem előtt kell tartanunk, hogy a pontos dokumentáción alapszik a teljes rendszer, így egy nem megfelelően alapos dokumentálás akár a projekt kudarcát is okozhatja. Mindez fokozottan érvényes nagyméretű szoftverfejlesztő cégek esetében, ahol a projektet megvalósító projekt tagok több különböző telephelyen vannak, illetve ha több vállalkozó fejleszti egy nagyobb rendszer kisebb alrendszereit.

3.11 Követelmények validálása és kezelése

A követelmények validálása annyit jelent, hogy az elkészített dokumentumokat megvizsgáljuk, hogy tényleg azokat a funkcionalitásokat fedi le, amelyeket a megrendelő kíván majd az elkészített szoftvertől. A validálási folyamat teljes egészében átfedi az elemzést, mert elsődleges célja az, hogy megtalálja a követelményekkel kapcsolatos problémákat, így folyamatos ellenőrzés szükséges. Bár sokan elhanyagolhatónak tarják ezt a részfolyamatot, egy valódi szoftverfejlesztési folyamatban igenis szükséges a dolog. Gondoljunk bele, ha hibás

követelmények alapján implementálnak egy részfunkciót a rendszerben, majd ezt csak akkor veszik észre amikor már az adott programrész már működik, és elvileg kész. Ekkor egészen a tervekig kell visszanyúlni, és megváltoztatni azokat. Ez a kis változtatás felgyűrűzhet egészen a tesztelésig: a funkcionalitást akár teljesen át kell írni, az ezt igénybe vevő más programrészeket is módosítani kell, és a teszteseteket is újra kell tervezni és létrehozni, hiszen eredetileg egy hibás követelményt teszteltek. Látható, hogy egy kis probléma milyen nagy bajjal jár, és az implementálási rész végrehajtása közben ilyen jellegű változtatásokat eszközölni sokkal költségesebb, mint amikor még csak papíron létezett az egész szoftver.

Tehát milyen módon lehet ellenőrizni, hogy a követelmények megfelelőek? Egyrészt ellenőrizni kell az egyes funkcionalitásokat, és azt is átgondolni, hogy ezek nem vetnek-e fel további megvalósítandó funkciókat. Mivel több fajta kulcsfigurával kell dolgozni, akiknek mind van valamilyen igényük, ezért természetesen olyan kompromisszumra kell jutnunk, amely mindenkinek megfelelő, kellő mértékben. Át kell vizsgálni a leírást, hogy ne legyenek benne ellentmondások, az egyes részek ne zárják ki kölcsönösen egymást. Továbbá figyelni kell arra, hogy a rendszer teljes, azaz minden szükséges szolgáltatást és megszorítást tartalmaz, amelyet a megrendelő elvár a rendszertől. Emellett pedig a rendszernek megvalósíthatónak kell lennie, tehát az ismert és felhasználható technológiák lehetővé tegyék a szoftver implementálását, a szerződésben megadott idő- és költségkereten belül. Végül a rendszernek verifikálhatónak kell lennie, tehát az egyes rendszerkövetelmények ténylegesen ellenőrizhetőek kell legyenek, tehát a megrendelő számára egyértelműen fel tudjunk mutatni olyan teszteket amelyek mindkét fél számára bizonyítják, hogy az alkalmazás az szoftverkövetelmények dokumentumában rögzítettek szerint megfelelően működik, és teljesíti az összes előírt követelményt.

Tudjuk, hogy a leírt követelmények egy adott állapotot rögzítenek. Azonban megfigyelhető a nagyobb rendszerek esetében, hogy ezek a követelmények az idő előrehaladtával folyamatosan változnak. Ez abból adódik, hogy minden igyekezetünk ellenére is, nem tudunk elsőre átlátni minden problémát, így nem is lehet őket tökéletesen definiálni. A szoftver készítése során az egyes kulcsfigurák egyre jobban megértik annak működését, így új igények merülhetnek fel, ezért a követelményeknek is szükségszerű módon fejlődniük kell, hogy mindig a megfelelő képet tükrözzék. A követelménydokumentum tárgyalása során már volt szó erről, amikor említettem, hogy a követelmények meghatározásakor figyelni kell arra, hogy bővítési pontokat kell meghatározni, és ezeknek megfelelően kell majd a rendszert megtervezni. Ezek

is azt a célt szolgálják, hogy a rendszer megfeleljen a változó követelményeknek is, és minél kevesebb erőfeszítéssel lehessen ezeket is megvalósítani, az eredeti rendszer lehető legkisebb módosításával. Sajnos a rendszer felhasználói és alkalmazói csak akkor fogják látni annak a szervezetre gyakorolt hatásait, ha már elkezdték a használatát, így az alkalmazás során keletkező új tapasztalatok hatására óhatatlanul is merülnek fel új igények.

A követelmények kezelése tehát nem más, mint a rendszerkövetelményekben bekövetkező változások megértése és vezérlése. Fel kell tárni az egyes követelmények közötti összefüggéseket, hogy felmérhessük az esetleges változatás hatásait. Meg kell alapozni egy olyan formális folyamatot, amellyel javaslatot tehetünk a változtatásokra, és ezeket a követelményekhez köthetjük.

4 Tervezés

A szoftverfolyamat egyik nagy fejezete, a követelményelemzés már lezárult, a rendszer megvalósításához szükségesnek vélt követelményeket feltártuk, és rögzítettük a megfelelő dokumentumokba. Ha ezek a lépések megfelelően történtek, akkor ennek alapján már hozzákezdhetünk a rendszer megtervezéséhez. A szoftvertervezés lényege a szoftver logikai szerkezetére vonatkozó döntések meghozatala, melyet valamilyen módon le kell írnunk, hogy megvalósítható legyen a fejlesztők számára. Ez a leírás történhet valamilyen logikai modell segítségével, mint az UML, vagy egy informális jelölésrendszer segítségével is reprezentálhatjuk a rendszer terveit.

4.1 Felépítés

Értelemszerűen, az átláthatóság és a karbantarthatóság érdekében egy nagy rendszert érdemes kisebb egységekre bontani, melyek alrendszereket alkotnak. Ezek az alrendszerek egymás közötti kommunikáció segítségével biztosítják a teljes rendszer funkcionalitását. Ezeknek a részeknek az azonosítása és kommunikáció és vezérlés kialakításának feladata a tervezési folyamatra hárul. A követelmény- és architekturális tervezés folyamatai között jelentős az átfedés. Ideális esetben a rendszer specifikációjának nem szabadna tervezési információkat tartalmaznia, a valóságban azonban ez nem így szokott történni.

A rendszert tehát alrendszerekre kell bontani, első körben meg kell határozni milyen módon osztjuk fel azt. Az alrendszerek tervezése tulajdonképpen a rendszer durva szemcsézettségű komponensekre való bontása, ahol ezek a komponensek lehetnek akár önálló

rendszerek is. Annak eldöntése, hogy a rendszert hogyan kell részekre bontani, igen nehéz probléma. Természetesen a fő szempontot a rendszerkövetelmények jelentik, és olyan tervet kell készíteni, amelyben a követelmények és az alrendszerek igen közel állnak egymáshoz. Ekkor ugyanis a követelmény megváltozása esetén a változás lokalizálva van, nem pedig több alrendszer között oszlik meg.

A tervezés természetesen kreatív folyamat, ahol megpróbáljuk előállítani egy rendszer szerkezetét, amely megfelel a funkcionális és nemfunkcionális követelményeknek. Mivel ez kreatív folyamat, az elvégzendő tevékenységek nagyban függnek a kifejlesztés alatt álló rendszer típusától, a rendszert építő személyek tudásától és tapasztalataitól, valamint magától a rendszer követelményeitől. Éppen ezért az architektúra megtervezésének folyamatára tekinthetünk úgy is mint egy döntési folyamatra, melynek során olyan alapvető döntéseket kell hoznunk, mely kihat a rendszer és a fejlesztés folyamatának egészére.

Ugyan minden szoftver egyedi a maga módján, azonban ha ugyanazon szakterületen alkalmazandó rendszerekről van szó, akkor azok rendelkeznek bizonyos közös vonásokkal, és ugyanazon szakterület fogalmait használják. Ezért a szoftverek felépítése is mutat bizonyos közös jellemzőket, ezek tervezése alapulhat előre megadott modelleken és tervezési stílusokon. Ezen modellek és stílusok alkalmazásakor gyakorlatilag egy korábban elkészített mintát követünk, azt szabjuk át a megtervezendő rendszer igényeinek megfelelően. Természetesen erre szükség is van, hiszen egyetlen modell sem tökéletes, sőt vannak bizonyos erősségeik és gyengeségeik. Ezen a problémán úgy segíthetünk, ha nem egy adott mintát követünk, hanem több ismerete alapján ötvözzük a megfelelőeket, ezzel erősítve a különböző erősségeiket, és kiküszöbölve a gyengeségeiket.

Tehát szükséges, hogy már a tervezés korai szakaszában eldöntsük, milyen lesz a rendszer szerkezeti modellje. Ezt a döntést az alrendszerek közvetlenül is tükrözhetik, de gyakran előfordul, hogy az alrendszer modellje részletesebb a szerkezeti modellnél, így ez nem minden esetben jelent egyszerű megfeleltetést. A helyzetet az sem könnyíti, hogy többféle szerkezeti modell közül választhatunk. Ezek közül a modellek közül csak kettőre térnék ki, amely a saját alkalmazásom szempontjából érdekes, és általános értelemben a webes alkalmazásokban is sokszor megjelenik.

Egy webes alkalmazás során egy tervező számára nem kérdéses, hogy a kliens-szerver modell, ami adott és biztosan alkalmazni kell. Ez nem jelent mást, mint azt, hogy a rendszer résztvevői két nagy csoportba oszthatóak:

- Szerverek: ezek szolgáltatásokat nyújtanak a többi komponens számára, így egymás számára is, itt találhatóak meg az adatbázisok, alkalmazás-szerverek, webszerverek, egyéb tároló rendszerek a különböző médiaformátumok számára, lehetnek még nyomtatók, állományszerverek és fordítószerverek.
- Kliensek: ezek azok a komponensek, akik hozzáférhetnek és igénybe vehetik a szerverek által nyújtott szolgáltatásokat, ezek lehetnek más szervereken futó alrendszerek, és a egyéb a szoftvertől független alkalmazások (pl. egy böngésző-program)

Ezt a két fő komponenstípust egy hálózat választja el, ami lehet intranet, tehát egy vállalat belső hálózata, és maga az Internet is. A klienseknek csupán a szerverek neveit kell tudniuk, és azt hogy egy adott szolgáltatásuk, hogyan vehető igénybe, és ezek ismeretében távoli eljáráshívások segítségével tudják igénybe venni az adott funkcionalitást. Az hívások továbbítását valamilyen kérés-válasz alapú protokollon keresztül lehet megvalósítani, klasszikusan erre használható a WWW alapját képező HTTP.

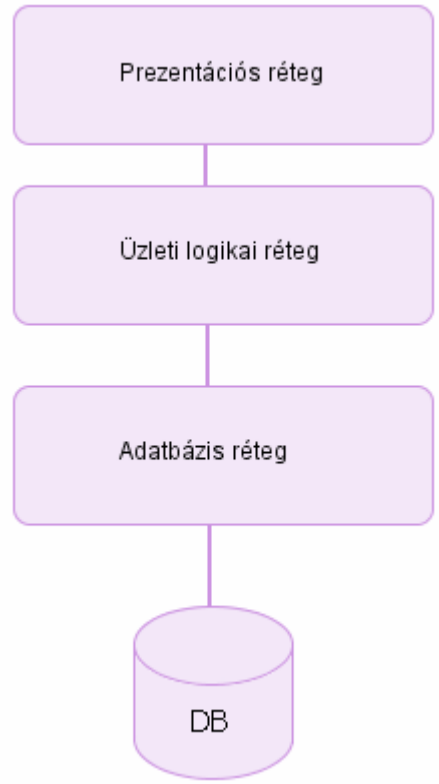
Ezen modell legnagyobb előnye, hogy egy osztott architektúrát ír le, ezért hatékonyan alkalmazható olyan szoftverek megvalósítása esetében, ahol sok, elosztott feldolgozó egységet kell alkalmazni, és fontos szempont a rendszer skálázhatósága és rendelkezésre állása. Ugyanis a rendszer terhelése eloszthatóa különböző szerverek között, illetve szükség esetén könnyen új szerver illeszthető be, illetve meghibásodás esetén kivehető, a teljes rendszer leállítása nélkül. Ha az egyes komponensek kommunikációját valamilyen univerzális alapon valósítjuk meg, például HTTP-n keresztüli XML alapú kommunikációt végzünk, akkor a rendszer könnyen bővíthető újabb komponensekkel, anélkül, hogy az üzenetváltási módszert nagyban változtatni kellene.

A fenti modell hátránya, hogy sokszor a kétféle komponensnek megfelelően csak két réteget alkalmaznak a megvalósításakor. Tehát vesznek adattároló szervereket, és kiszolgáló webszervereket. Az üzleti logika megvalósítása során az egyik réteget jelenti a kiszolgálókon működő valamilyen webtechnológián alapuló megvalósítás, mely magában foglalja a megjelenítési és feldolgozási funkcionalitást is, azaz ebben a rétegben van az üzleti logika. Az oldal működése során keletkező perzisztens adatok tárolása az adatbázis-szerverekre hárul, melyek

irányítását szintén az előző réteg végzi, valamilyen adatbázis kapcsolaton keresztüli sima SQL utasítások végrehajtásával. Egyszerű és kis rendszerek esetében ez az eljárás elfogadott volt, és megvalósítása kevés tervezés után lehetővé vált. A probléma abban rejlik, hogy a különböző szolgáltatások megvalósítása gyakorlatilag egyetlen rétegben összpontosul. Ennek hatására a karbantarthatóság és a bővíthetőség erősen romlik, a rendszer robosztussága alacsony, és a hibák kijavítása ismeretlen módon hathat a rendszer többi részére.

Ennek kiküszöbölése érdekében érdemes ötvözni ezt a modellt egy rétegezett modellel, melynek lényege, hogy szolgáltatások szempontjából különálló rétegekbe szervezzük a teljes rendszert. Ezek a rétegek természetesen egymásra épülnek, és igénybe veszik a közvetlenül alattuk lévő réteg szolgáltatásait, és szolgáltatásokat nyújtanak a felettük lévő réteg számára. Ez a megvalósítás ismerős lehet például a hálózati megvalósításokban jártas embereknek, emlékezzünk csak vissza az OSI modellre. Általában 3 réteget szoktak alkalmazni, de mivel ez problémafüggő, ezért tetszőlegesen bővíthető további rétegekkel.

10. példa: rétegelt architektúra



Az említett három réteg legalsó szereplője, az adatbázis réteg, melyet perzisztencia rétegnek is szoktak nevezni. Ennek feladata minden adatbázis közeli funkció megvalósítása, a rendszer által használt adatszerkezetek és objektumok konzisztens tárolása az adatbázisban, és

eredeti állapotukban való visszanyerése onnan a későbbi felhasználáshoz. Ennek a rétegnek megfelelő módon kell elfednie az adatbázis jelenlétét a felette lévő rétegek elől. Ez azért is fontos, mert a felsőbb rétegek számára transzparens módon kell viselkednie, tehát mindegy hogy mi az adatbázis: lehet hogy az ténylegesen egy adatbázis-kezelő rendszer, lehet hogy csak egy pár szöveges állomány vagy Excel-táblázat. Illetve ezen túl az sem elképzelhetetlen, hogy nem csak egyetlen adatbázis van, tehát a fenti megoldások keverednek, több adatforrásból származnak a rendszer által felhasznált adatok.

A középső réteget üzleti logikai rétegnek szokták nevezni, melynek feladata a funkcionális követelményekben foglalt üzleti logika megvalósítása. Az adatbázis réteg felől kér és afelé továbbit adatokat, azokat feldolgozza, bizonyos transzformációkat végez rajta, majd ezek eredményét továbbadja a felette lévő réteg számára. Itt kell megvalósítani majd minden logikai tevékenységet, bizonyos események bekövetkeztének előfeltételeit vizsgálni, megfelelően reagálni a felhasználói interakciókra, gyakorlatilag itt helyezkedik el a rendszer irányító központja.

A legfelső réteg a megjelenítésért felelős, így megjelenítési vagy prezentációs rétegnek is nevezik. A felhasználó közvetlenül ezzel a réteggel áll kapcsolatban, hiszen a szoftver felhasználói felületén keresztül végez interakciót a rendszerrel. Minden megjelenítési és a megjelenítéssel kapcsolatos egyszerű validálási folyamatok ebben a rétegben zajlanak, itt van meghatározva, hogy az egyes kattintások hova visznek, hogy a felületelemek hogyan és milyen elrendezésben jelennek meg, hogyan néznek ki stb. Minden más végrehajtásához ez a réteg igénybe veszi az üzleti logikai réteg szolgáltatásait, és rajta keresztül közvetett módon az adatbázis réteg szolgáltatásait is.

Ezzel a háromrétegű modell segítségével gyakorlatilag egy tervezési mintát valósítunk meg, amelyet Model-View-Control (MVC) modellnek neveznek. Ez a minta három rétegbe osztja fel a rendszert, az egyes rétegek rendre a modellt (adatbázis réteg) a nézetet (megjelenítési réteg) és a vezérlést (üzleti logikai réteg) valósítják meg. Ezt a tervezési modellt leggyakrabban webes alkalmazásoknál használják, hiszen ott a megjelenítési réteg elkülönülése megvan a HTML alapú megjelenítés miatt, így egy másik rétegbe kényszerülnek azok a alkalmazás komponensek, amelyek dinamikusan állítják elő ezeknek az oldalaknak a tartalmát, és az adattároló réteg elkülönítése innen már csak egy lépés. Fontos eleme, hogy a különböző rétegek között jól definiált interfészeket alkalmaz, mely lehetővé teszi az egységes nézőpontot a réteg funkcionalitását felhasználó réteg számára, továbbá elősegíti, hogy a rétegek esetleges

cseréje könnyen elvégezhető legyen. Az interfészek használatából fakadóan egy réteg nem látja az alatta lévő réteg megvalósítását, így az tetszőleges módon újraimplementálható, anélkül hogy a felettes rétegekben bármilyen módosítást is kellene végezni. Ezen előnye mellett ez a megközelítés támogatja az inkrementális fejlesztést is, hiszen ha implementálunk egy réteget, akkor annak funkcionalitása elérhetővé válik a felettes réteg számára, és így a szolgáltatások egy része is elérhetővé válik az azt felhasználók számára.

A modell hátránya, hogy a struktúrát a rétegek meg is bonyolíthatják, főleg, ha több réteg van, sokszor egy legfelső réteg számára olyan funkcionalitásokra is szüksége lehet, amelyet csak egy alsóbb rétegtől kaphat meg, ekkor több rétegen keresztül tudja csak azt elérni, és ez megbontja a modell felépítését abban az értelemben, hogy minden réteg csak az alatta lévőtől függ. Továbbá értelemszerűen romlik a teljesítmény, egy adott felhasználói interakcióra való reagáláshoz minél több rétegen kell átlépnie a megadott parancsnak annál rosszabb a válaszidő. Ahhoz hogy a megfelelő reakcióidő biztosítható legyen, bizonyos szempontból újra meg kell bontani a modell elképzeléseit, és lehetővé tenni, hogy bizonyos rétegek, több réteget átlépve is kommunikálhassanak egymással.

Mint ahogy említettem is webalkalmazások esetén ez az architektúra az elfogadott, ezért én is ezt a megközelítést használtam a rendszer megvalósításakor, melynek során az említett három rétegbe soroltam a nyújtandó szolgáltatásokat, meghatározva ezzel az egyes rétegek felelősségi körét, és a megvalósítandó funkcionalitásokat, és az ezekhez szükséges interfészek specifikációját.

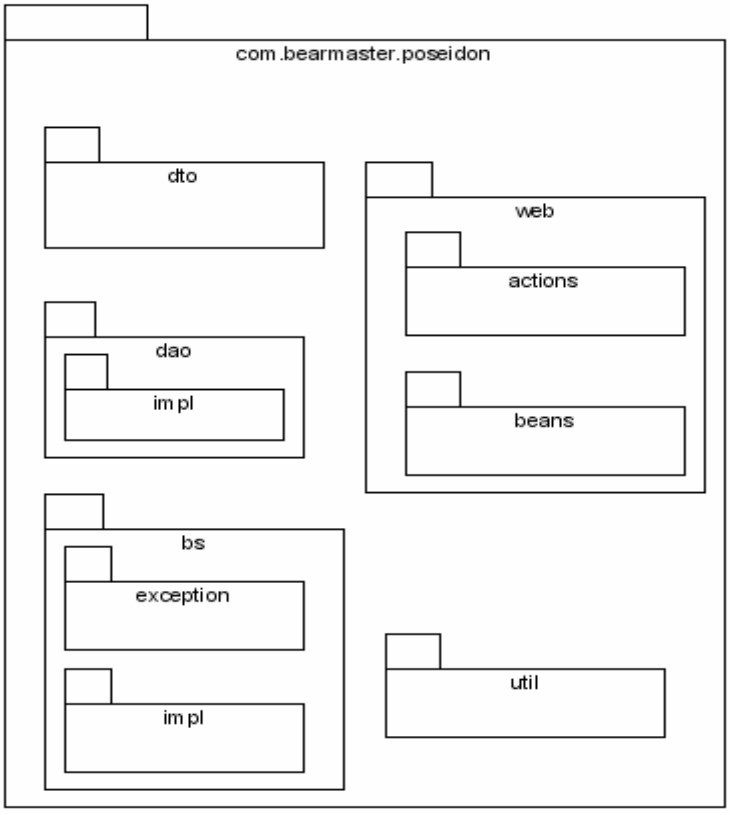
4.2 Modulokra bontás

Meghatároztuk a teljes rendszer architektúráját, ezzel megközelítőleg felvázoltuk, hogy milyen alrendszerek fogják felépíteni az alkalmazásunkat. Azonban ezek az alrendszerek még túlságosan nagyok, hogy pontosan definiálhatóak legyenek, ezért egy alrendszert tovább kell bontani megfelelő méretű és számú modulokra. Míg egy alrendszer egy önálló rendszernek tekinthető, mely bizonyos szolgáltatásokat nyújt a külvilág felé, és ezeket egy rögzített interfészen keresztül biztosítja a többi alrendszer számára, addig az alrendszereket felépítő kisebb modulok nem képesek „önálló életet élni”, bár ők is szolgáltatásokat nyújtanak más modulok számára, mégsem tekinthetőek alrendszernek, mert jóval kisebbek és egyszerűbbek annál, és csak több modul együttese ad teljes funkcionalitást.

A modulokra bontás két fő módon történhet: funkció-orientált megközelítéssel, amikor úgy fogjuk fel az egyes modulokat, mint a bemenő adatokat megfelelő kimenő adatokká konvertáló funkcionális részegységeket, és objektum-orientált felbontás szerint, ahol az egyes alrendszereket egymással kommunikáló objektumok halmazára bontjuk szét. Természetesen az alkalmazott nyelvnek és technológiáknak megfelelően az objektum-orientált felbontást célszerű választani, ezért a továbbiakban ezt a módszert részletezem.

Tehát az egyes alrendszereket lazán kapcsolódó, jól definiált interfészekkel rendelkező objektumok halmazára kell felbontani. Ezek az objektumok egymásra épülnek, felhasználják a többi objektum által nyújtott szolgáltatásokat, és keretbe foglaló alrendszer interfészén keresztül biztosítják az alrendszer funkcionalitását. Az objektum pedig nem más, mint saját állapottal és önálló viselkedéssel rendelkező entitás. A belső állapotát bezárja a külvilág elől, ez kifelé nem látszik, és közvetlenül nem is érhető el, azonban biztosít megfelelő műveleteket, metódusokat, amelyek segítségével más objektumok befolyással lehetnek az állapotára, képesek megváltoztatni azt.

11. példa: csomagdiagram



Az objektum-orientált tervezés tehát nem más, mint az alrendszer működéséhez szükséges objektumosztályok felderítése és a köztük lévő kommunikációs kapcsolatok megtervezé-

se. Ez a folyamat három különböző fázisra bontható fel: első lépésként elemeznünk kell a szakterület alkalmazás modelljét, azonosítani a szükséges főbb objektumosztályokat, és meghatározni az esetleges segédobjektumok osztályait. Ezen osztályok tükrözik a követelményelemzés során feltárt objektumokat, metódusaik pedig a végrehajtható műveleteket. Ez után jöhet a tervezés, mely során az első lépésben feltárt osztályok segítségével egy megfelelő modellt alakítunk ki, meghatározzuk, hogy az egyes osztályok hogyan függnek egymástól, melyiknek mi a felelőssége, mit kell tudnia végrehajtania, és mely funkcionalitásokhoz kell igénybe vennie más osztályok objektumait. Ezen fázisban további segédobjektumok válhatnak szükségessé. Az utolsó részben pedig a tervet meg kell valósítani, valamely objektumorientált nyelven implementálni kell a terveket, hogy megkapjuk a működő rendszert. Természetesen ez a tevékenység már átnyúlik a szoftverfolyamat implementációs részébe.

A lépések közötti átmenet ideális esetben észrevehetetlen, ami maga után vonja, hogy mindegyik lépés kompatibilis jelölésrendszerrel rendelkező leírási módot alkalmaz. Minden lépésben egyre finomodnak az előző szinten már létező objektumosztályok, ezeket egyre jobban részletezzük, újabb funkcionalitást biztosító osztályokat adunk a rendszerhez, és ezt addig iteráljuk amíg el nem érjük a kívánt részletességet. Tekintve, hogy az objektumorientált paradigma szerint az objektumok megvalósítják az egységbe zárást, így a tényleges adatreprezentáció a külvilág számára nem ismert, ezért ennek tényleges eldöntése egészen az implementációig kitolható. Bizonyos esetekben az objektumok elosztása, illetve annak eldöntése, hogy az objektumaink szekvenciálisan vagy konkurensen fognak-e működni szintén elhalaszthatóak. Megjegyzésként megemlítem, hogy a modellvezérelt architektúra ezen alapszik, ez a módszer a tervezést két szinten képzeli el: egy implementáció-független szinten, melyen a rendszer egy absztrakt modelljét alkotjuk meg, és egy implementáció-függő szinten, amelyre leképezzük az előző szint modelljét, mely alapján majd a kódgenerálás is megvalósítható.

Az osztályok elemzésekor természetesen támaszkodnunk kell a követelményelemzésben létrehozott fogalomszótárra és szakterületi folyamatok leírásaira. Ezekben megtalálhatjuk a kulcsfontosságú osztályokat (a mi esetünkben például a csónak csónaktípus, ügyfél stb.) illetve ezekből a leírásokból megtudhatjuk azokat a jellemzőket amelyek feltétlenül szükségesen egy ilyen objektum leírásához. Ezen objektumok nagy része valamilyen módon utazni fognak a különböző rétegek között, ún. transzfer-objektumokba csomagolva találhatjuk meg majd őket, illetve valamilyen a perzisztencia réteg által meghatározott módon le kell őket képezni adattároló egységekre, adatbázis táblákra, XML formátumú állományokra. Ezeknek az objek-

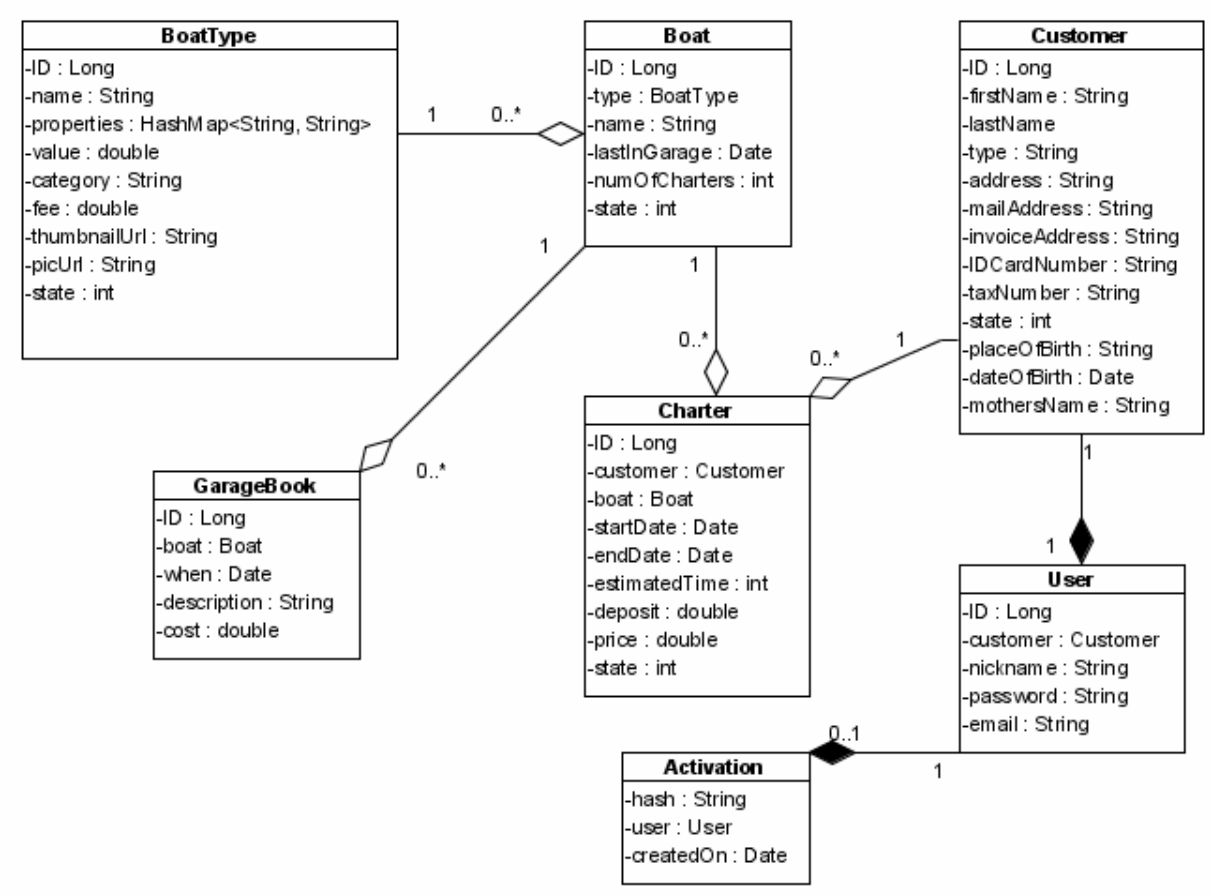
tumoknak a viselkedése csak annyiból fog állni, hogy a különböző jellemzőkhöz lekérdező metódusokat biztosít, illetve a módosítható jellemzőkhöz tartozik majd egy beállító metódus.

A szakterületi folyamatokból azonosított tevékenységek és funkcionalitások megvalósítását más rétegobjektumok fogják implementálni: a réteg interfészeket implementáló „kiszolgáló objektumok" (service objects) jelennek meg, amelyek a transzferobjektumokon operálnak, és ezen műveletek eredményeképpen más rétegbeli szolgáltatásokat hívnak meg, és ezek számára további transzfer-objektumokba csomagolt információkat továbbítanak. Fontos, hogy megfelelően képezzük le a szükséges funkcionalitásokat, hogy jól definiált interfészeket tudjunk készíteni, mert ezek utólagos módosítása végiggyűrűzhet az összes felettes rétegen, amely rengeteg munkát jelent. Egy jól definiált interfész elkészítése után a különböző programozói csoportok egymástól függetlenül dolgozhatnak a különböző rétegeken, anélkül, hogy tudniuk kellene akármit is a többi rétegről. A saját kódjaik teszteléséhez elegendő lesz csupán ún. minta implementációkat (mock implementation) készíteni, melyek a teszteknek megfelelő minimális funkciókat valósítják meg az kapcsolódó rétegekből.

Ahhoz, hogy a kód megfelelően karbantartható legyen, oda kell figyelni, hogy ne legyenek benne felesleges kód duplikációk. Ehhez csak azt kell megvizsgálni, hogy bizonyos metódusok, funkciócsoportok hol ismétlődnek meg, az ilyen metódusokkal rendelkező objektumok valószínűleg szorosabb kapcsolatban állnak egymással mint korábban gondoltuk. A közös funkcionalitást generalizáció segítségével kiemelhetjük egy közös ősbe, így egy helyen lesz majd elkészítve a szükséges kód, melyet a leszármazott osztályok objektumai felhasználhatnak, és igény szerint újradefiniálhatják ha mégis erre lenne szükség. Ugyanez igaz az interfészekre, mivel a különböző nyelvek támogatják az interfészek közötti öröklődési hierarchiát is, ezért már ezen a szinten jelezhetjük, hol lesznek ilyen jellegű összefüggések az osztályhierarchiában.

Az ilyen módon összegyűjtött információkat természetesen rögzíteni is kell, melyre többfajta eszköz is van, a legelterjedtebb módszert itt is az UML eszközei biztosítják. Méghozzá osztálydiagramok segítségével írhatjuk le a felfedezett interfészeket és osztályokat, jelölhetjük a közöttük lévő asszociációs és generalizációs kapcsolatokat. A rétegek megtervezésénél előnyös lehet különböző tervezési minták alkalmazása, melyek biztosítják, hogy bizonyos problémák megoldására már kipróbált és jól működő megoldásokat alkalmazzunk, azonban ezekkel is óvatosan kell bánni, csak abban az esetben érdemes őket használni, ha tényleg indokolt és megértett a működésük.

12. példa: osztálydiagram (részlet)



Az osztályok felderítésén túl nagyobb léptékben is kell gondolkodnunk a rendszer felépítésekor: a különböző osztályok csomagokba szervezhetőek a jobb átláthatóság érdekében, és ezek a csomagok is külön hierarchiát alkotnak. Általános elv nincs a csomagszerkezet szervezésére, általában a csomagok nevei a céget azonosító egyedi elérési úttal kezdődik (pl.: com.mycompany.myappname), ezen belül érdemes rétegenként külön csomagot biztosítani. Ezen túl az UML lehetősége ad az alrendszerek jelölésére is, így elkészíthetünk egy olyan diagramot is amely segítségével áttekinthető a teljes rendszer felépítése, látszanak rajta az egyes alrendszerek illetve az őket felépítő főbb csomagok és osztályok.

4.3 Objektumok együttműködése

Elkészítettük a csomag- és osztálydiagramokat, melyek együttesen az alrendszer modelleket adják, azonban ezek statikus modellek, melyek nem mutatják, hogy ténylegesen hogyan fognak együttműködni a rendszert felépítő objektumok. Azonban a tervezéskor fontos azt is tudni, hogy hogyan fognak ezek az objektumok viselkedni a tényleges alkalmazásukkor, tehát szükséges valamilyen dinamikus modell alkalmazása, melyből ez is kiderül. Erre használható

az UML szekvencia- vagy együttműködési diagramja. Ez olyan dinamikus modell, amely az interakciók minden lehetséges módjára leírja, hogy ezek milyen sorrendben következnek be.

Ezen a diagramon látható, hogy egy adott interakcióban milyen objektumok vesznek részt, ezek a felső részen vízszintesen elrendezve találhatóak. Mindegyik objektumhoz tartozik egy függőleges szaggatott vonal, az ún. életvonal. Ezek a függőleges vonalak jelzik az időt, erről lesz leolvasható, hogy időben az egyes objektumok hogyan jönnek létre és élnek az adott interakció végrehajtása során, illetve a műveletek sorrendje is látható. Az objektumok közötti interakciókat vízszintes nyilak ábrázolják, melyek a függőleges életvonalakat kötik össze, ezeket a nyilakat általában megcímkézik az adott művelet nevével. Fontos, hogy ezek a nyilak nem adatáramlást jelölnek, hanem az interakció számára alapvető üzeneteket és eseményeket. Végül az életvonalakon vékony függőleges téglalapok helyezkednek el, melyek azt az időszakot jelöli, amíg az adott objektum a rendszer vezérlőobjektuma, tehát amikor időben eléri a téglalap tetejét, akkor kapja meg a vezérlést, és a téglalap aljához érve átadja a vezérlést egy másik objektumnak.

Megállapítható, hogy a diagram első objektuma általában egy aktor, aki elindítja az adott interakciót, és bizonyos esetekben az utolsó objektum szintén egy aktor, amelyben befejeződik az interakció (pl. egy adatbázis, amelyben mentésre kerülnek bizonyos adatok).

A terv dokumentálása során minden lényeges interakcióra el kell készíteni egy ilyen szekvencia modellt. Mivel készítettünk használati eset modelleket is, ezért érdemes, hogy minden ilyen diagramhoz tartozzon egy szekvencia diagram is.

4.4 Az objektumok aktivitása és állapota

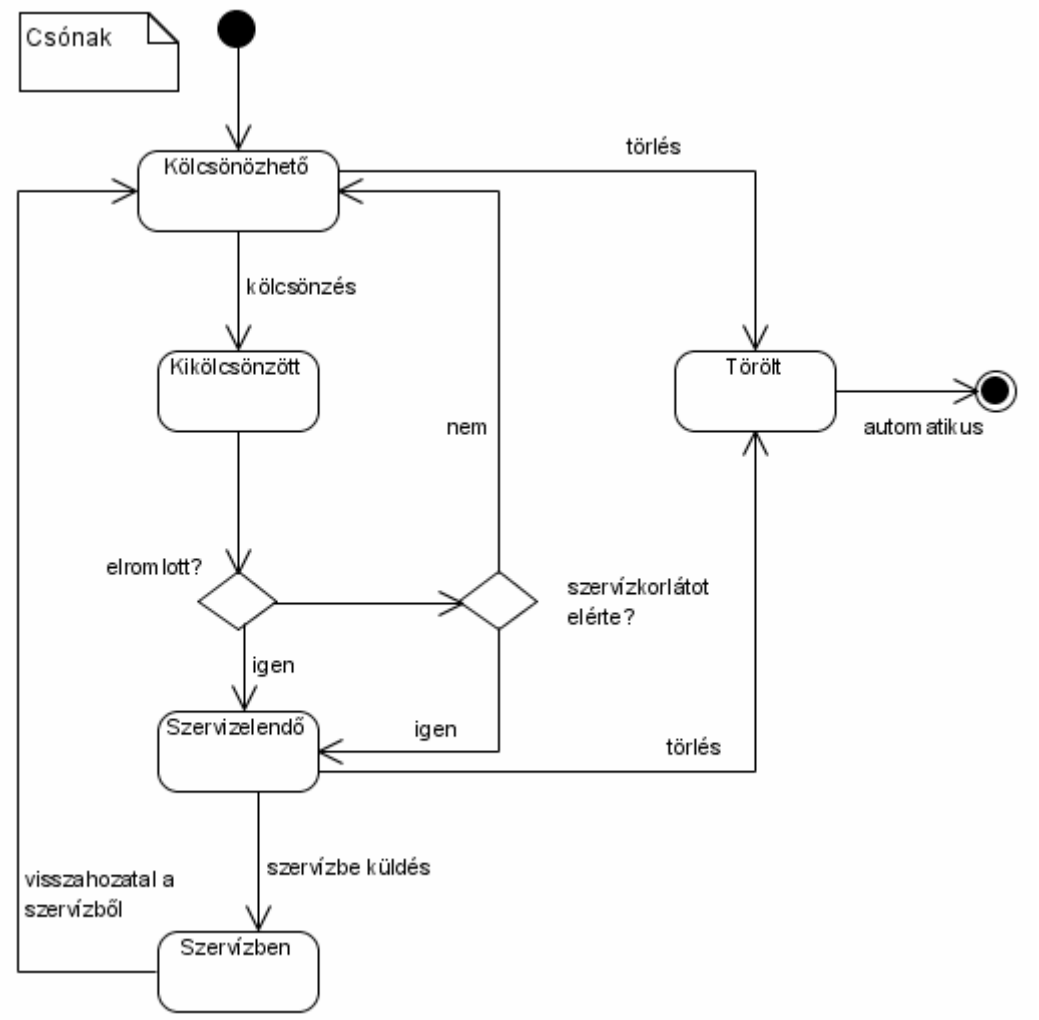
A szekvencia diagramokat objektumok egy csoportjának együttes viselkedésének modellezésére használják, de szükség lehet arra, hogy összefoglaljuk az önálló objektumok általuk feldolgozható különböző üzenetekre válaszul adott viselkedését is. Ennek leírására használhatjuk az állapotgép modellt vagy rövidebb nevén az állapotdiagramot. Ez a diagram megmutatja, hogy az egyes objektumpéldányok hogyan változnak az időben az általuk kapott üzenetek hatására.

Amikor egy objektum fogad egy üzenetet vagy hívhatjuk akár eseménynek is (ha a rendszer vezérlése nem központosított, hanem esemény alapú), akkor az arra adott válasza függ az magától az üzenettől és az objektum pillanatnyi állapotától. Az objektum erre az üzenetre valamilyen válasz formájában reagálni fog, és ezen válasz része lehet akár az is, hogy

megváltoztatja az állapotát. Ezt nevezzük állapotváltásnak vagy állapotátmenetnek. Az állapotdiagram egy olyan gráf, amelynek csomópontjai nem mások mint az objektum egyes állapotai, amelyeket az életciklusa során felvehet, élei pedig az egyes események hatására bekövetkező átmenetek. A diagramon az állapotot lekerekített téglalappal jelöljük, melyben feltüntetjük az állapotot, egy átmenetet egy irányított él jelöl, mely leképezi, hogy az átmenethez tartozó esemény hatására az objektum melyik állapotból melyikbe kerül át. A nyilat az esemény nevével vagy esetlegesen a bekövetkező metódushívás nevével címkézzük. Továbbá a diagramon szerepelhetnek feltételes elágazást jelző rombusz alakú elemek is. Ezeken jelöljük, hogy az adott állapot milyen attribútumát vizsgáljuk, illetve az egyes attribútum értékek esetén merre haladunk tovább a gráf mentén. Fontos hogy az objektum életciklusának van kezdete és vége, így ennek meg kell jelennie az állapotdiagramon is: a kezdőállapotot egy tömör kör jelöli, míg a végállapotot egy külső körben elhelyezett tömör kör. Egy adott állapotból kivezető különböző átmenetek különböző eseményekhez tartoznak.

Az állapotdiagram leírja az események hatására létrejövő állapotok sorrendjét, az állapotgép működését. Az objektum valamely állapotából – az első olyan esemény hatására, amelyhez tartozik átmenet, - az objektum egy következő állapotba kerül. Ha az objektum adott állapotában valamely esemény nem értelmezett, akkor annak hatására nem történik állapotváltás, azaz az eseményt figyelmen kívül hagyjuk. Előfordulhat olyan esemény is amelynek hatására lejátszódó állapotváltás alkalmával az objektum következő állapota megegyezik az aktuálisan fennálló állapottal. Egy eseménysorozat az állapotdiagramon egy útvonal bejárásának felel meg. Valamely állapotgép egy adott időpillanatban csakis és kizárólag egyetlen állapotban lehet. A kezdőállapotból csak egyetlen kiinduló él vezethet valamely állapotba, míg a végállapotba csak befelé vezethetnek élek, ebből bármennyi lehet, azonban külön végállapotokkal szokták jelölni azt, ha valamilyen okból érdemes megkülönböztetni, hogy az egyes objektumoknak máshogyan ér véget az életciklusa (pl. normálisan érte el a végállapotot vagy valamilyen bekövetkező hiba hatására).

13. példa: csónak állapotdiagramja



Egy állapotdiagram mindig az egész osztály viselkedését írja le. Mivel az adott osztály valamennyi példánya egyformán viselkedik, valamennyi ugyanazon attribútumokkal és állapotdiagrammal rendelkezik. Mivel mindegyik példány önálló, a többitől függetlenül létezik, ezért az egyes példányok pillanatnyi állapotát a konkrétan őket ért hatások határozzák meg. Azaz a különböző példányok különböző állapotban lehetnek.

Érdemes megemlíteni, hogy a fentiekhez hozzá tartozik az is, hogy önmagában nem csak egy esemény határozhatja meg egy objektum állapotváltását, hanem az átmenetek függhetnek az események paramétereitől is, ilyenkor a gráf élein nem csak az események neveit hanem a szükséges paramétereket is megjelölik. Így gyakorlatilag az is mondhatjuk, hogy két esemény különbözőnek számit ha más jellemzőkkel rendelkezik.

Általában nem szükséges minden objektumhoz állapotdiagramot készíteni. Sok objektum annyira egyszerű, hogy az állapotgép modell nem sokat segítene az implementáció során,

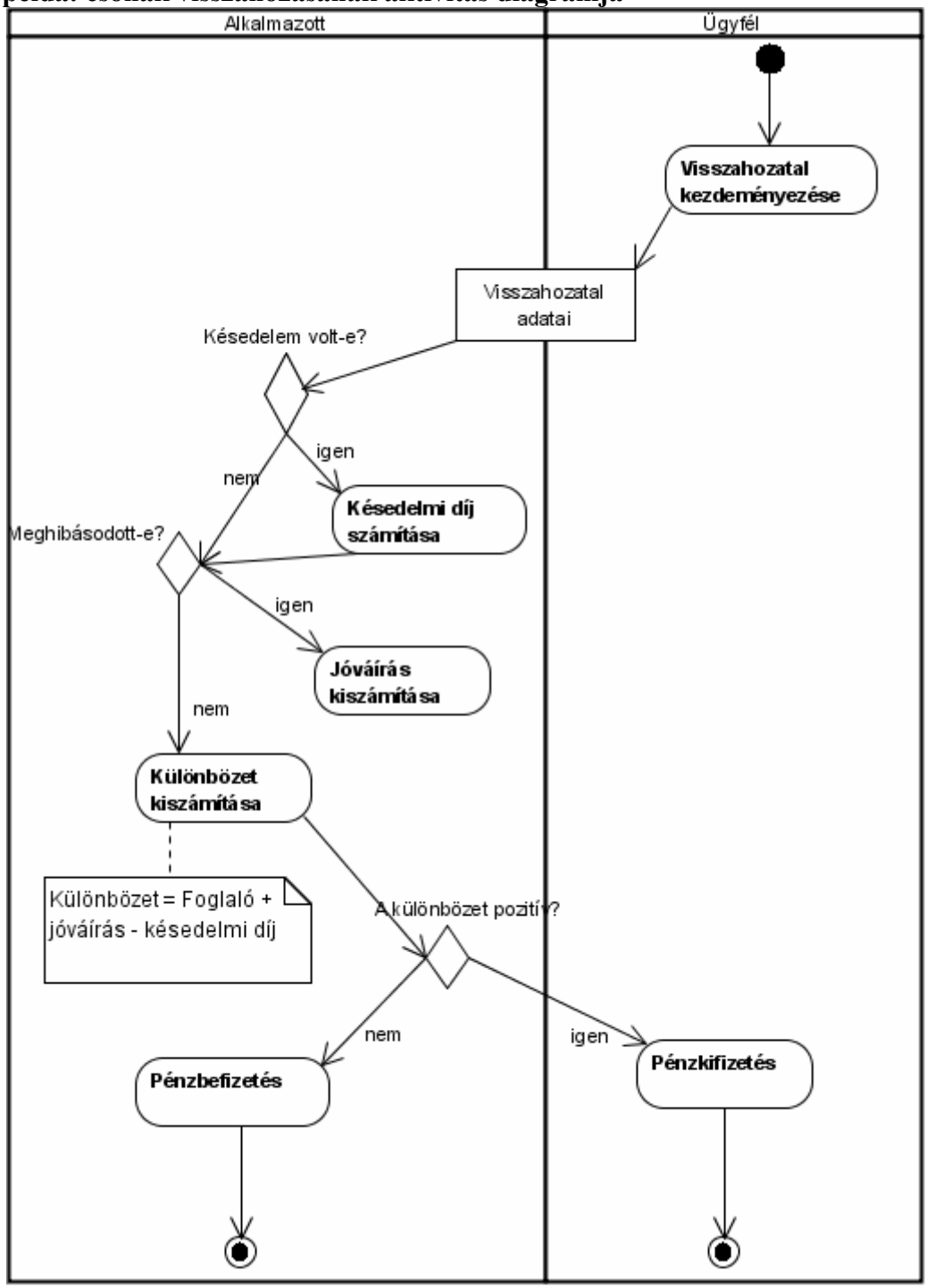
a programozók nem értenék meg jobban az objektumok működését, ezért elkészítésük ebben az esetben felesleges.

Az állapotok leírásán túl szükséges lehet a rendszert az egyes változások, az alkalmazás dinamikájának szemszögéből is vizsgálni. Abban az esetben ha különböző események, tevékenységek sorrendjét, a folyamatok vezérlését szeretnénk modellezni, akkor aktivitás diagramot alkalmazunk. Ez a diagram akár úgy is tekinthető mint a szekvencia diagram és az állapotdiagram ötvözete, mert ezek az állapotváltozásokat az egyes interakciók segítségével írják le. Mint már említettem az aktivitás diagram segít modellezni a folyamatokat, leírja azokat a dolgokat, hogy milyen események szükségesek egy adott cél eléréséhez és ezeknek milyen sorrendben kell bekövetkezniük, megjeleníti az összetett algoritmusok végrehajtási sorrendjét és modellezhetőek vele a párhuzamosan futó folyamatok is. Ettől függetlenül ez a diagram nem válthatja ki a már említett szekvencia és állapotdiagramokat, mert nem részletezi azt hogy az egyes objektumok hogyan viselkednek és hogyan működnek együtt.

Nevéből következően a diagramon aktivitásokat ábrázolunk, amely valamely nagyobb léptékű tevékenység részét képezi, az aktivitás lehet egy metódushívás vagy egy felhasználói interakció egyaránt. Ezeket szintén lekerekített téglalapokkal jelöljük, melybe az aktivitás nevét írjuk. Ezeket az aktivitásokat szintén átmenetekkel kapcsoljuk össze, mely kifejezi, hogy az egyik aktivitás befejeződött és az átmenet segítségével megkezdődhet az újabb aktivitás végrehajtása. Ezzel gyakorlatilag az egyes aktivitások között időbeli sorrendet határozunk meg. Az átmenetet irányított élek jelölik a diagramon. Itt is megengedett feltételes elágazások használata, melyek segítségével jelölhetjük, hogy egy átmenet csak akkor következik be ha a megfogalmazott feltétel teljesül, egyébként egy másfajta átmenetet produkál a rendszer, melyet szintén rombusszal jelölünk. Továbbá megjelenik egy új elem is, amely segítségével kezelhetjük a párhuzamos tevékenységeket, ez pedig a szinkronizációs vonal. Ez egy vízszintes vastag vonal, amelybe több gráf él vezethet, és egyetlen él hagyja el. Jelentése, hogy a bemenő élekhez tartozó aktivitások párhuzamosan is történhetnek, és a hamarabb befejeződő tevékenységek a szinkronizációs vonalhoz érve bevárják a többi tevékenység végrehajtását, és a következő tevékenység csak akkor kezdődhet ha minden kapcsolódó aktivitás befejezte működését. Ezzel lehetővé tesszük, hogy a tevékenységek párhuzamosan kerüljenek végrehajtásra, de nem követeljük meg, a hangsúly a szinkronizáción van. Ha tényleges párhuzamos végrehajtást szeretnénk jelölni, abban az esetben is ezt az elemet alkalmazzuk úgy, hogy egy szinkronizációs vonalba csak egy él érkezik és több ágon megy tovább ezzel jelölve, hogy az

adott ponttól a végrehajtás párhuzamos szálakon történik. Végül ezen a diagramon is szerepel egy kezdőállapot, amelyből csak egy kiinduló él van, ezt tömör körrel jelöljük, míg több végállapot is szerepelhet, ezeket egy körben szereplő tömör körrel tüntethetjük fel, és ezekbe csak érkezhet él, de nem indulhat onnan ki.

14. példa: csónak visszahozásának aktivitás diagramja



4.5 Felhasználói felületek

Mindezidáig csak a rendszer felépítésének megtervezésével foglalkoztam, holott egy felhasználó ezekkel a részekkel nem fog közvetlenül találkozni, a rendszer működése a háttérben meghúzódik, és csak ennek az eredményei fognak megjelenni. Természetesen a program végrehajtása során a feldolgozás eredményét valamilyen formában prezentálni kell, tehát szükség van egy felhasználói felületre, amelyen keresztül a felhasználó érintkezni tud az alkalmazással, annak utasításokat tud adni, és amelyről le tudja olvasni a kiadott utasítások eredményét.

Mivel a felhasználói felület (Graphical User Interface – GUI) szerves részét képezi az alkalmazásnak, ezért a tervezésére is nagy figyelmet érdemes fordítani. Ahhoz, hogy a szoftverrendszer teljesen kihasználhassa a benne rejlő lehetőségeit, lényeges, hogy a kialakítandó felhasználói felület illeszkedjen a reménybeli felhasználó szakértelméhez, tapasztalatához és elvárásaihoz. A jó felhasználói felület terve kritikus a rendszer megbízhatósága szempontjából. Sok úgynevezett „felhasználói hibát" az okoz, hogy a felhasználói felületek nem veszik figyelembe a valódi felhasználók képességeit és munkakörnyezetét. Egy rosszul tervezett felület azt eredményezheti, hogy a felhasználó esetleg nem fér hozzá a rendszer bizonyos számára kulcsfontosságú jellemzőihez, hibázik, és úgy érzi, hogy a rendszer segítségnyújtás helyett csak hátráltatja a munkájában, és így nem tudja kielégítő módon használni a szoftvert, amely előbb vagy utóbb elégedetlenséget szül benne. És ugye ki szeretne egy olyan környezetben dolgozni, amely számára csak kényelmetlen, nehézkes a használata és teljességgel átláthatatlan…

A felhasználói felületek tervezési döntései során figyelembe kell venni a szoftvert használó emberek fizikai és mentális képességeit. Figyelni kell arra, hogy az ember rövid távú memóriája véges kapacitású, ezért egy oldalon nem szabad elárasztani túl sok információval, mert lehetséges hogy nem fogja tudni mind befogadni. Ez ahhoz vezet, hogy többet hibázik a felhasználó, ami hibaüzenetekhez vezet, ez pedig csak tovább növeli a felhasználóban a feszültséget, és a hibázás esélyét. Az emberek fizikai képességei is nagyban eltérnek: míg egyesek jobban látnak és hallanak, addig mások lehetnek színvakok, vagy nagyothallók, megint másoknak jobb a kézügyességük, ezért a felület tervezésekor érdemes figyelembe venni ezeket a tényezőket, és nem csak a saját elvárásaink szerint megtervezni a felületet. Továbbá a kedvelt interakciós módok is különböznek az embereknél: egyesek képekkel míg mások szövegekkel szeretnek dolgozni, megint mások a közvetlen manipuláció hívei, ellentétben azok-

kal akik inkább az olyan interakciókat részesítik előnyben, ahol konkrét parancsokat adhatnak a rendszernek.

A tervezésnél érdemes bizonyos irányelveket figyelembe venni, mint a felhasználói jártasság elve, mely szerint a felületeken olyan információkat kell megjeleníteni, amelyek az őket felhasználó emberek számára fontosak és érthetőek, olyan fogalmakat és kifejezéseket kell használnia, amelyeket ismer a felhasználó.

A felületnek konzisztensnek kell lennie, tehát a rendszer parancsainak és menüinek ugyanazzal a formátummal kell rendelkeznie, a paramétereket ugyanazon parancsokhoz mindig ugyanúgy kell megadni. Ezzel lerövidül a betanulási idő, és csökken a hibázás lehetősége, illetve megkönnyíti az esetlegesen új alrendszerek működtetését is. Ez az elv vonatkozhat arra is, hogy a különböző szoftverek között is érdemes megtartani egy bizonyos fokú konzisztenciát, tehát ha egy adott billentyűkombináció beilleszti a vágólapon lévő objektumok az aktív felületelembe, akkor érdemes a mi alkalmazásunkban is ugyanazt a billentyűkombinációt alkalmazni. Ebben segíthetnek azok a szabványjellegű szabályok, amelyeket a többi nagy szoftverfejlesztő cég alkalmaz, így valószínűleg eléggé elterjedtek, így nekünk is érdemes ezekhez alkalmazkodni.

A minimális meglepetés elve szerint a rendszernek bizonyos műveletsorok után mindig ugyanúgy érdemes működnie. Ugyanis a felhasználó felépít magában egy ok-okozati összefüggésrendszert, amely segítségével megérti a rendszer belső működését. Ha egy bizonyos interakció a felhasználó részéről egy adott változást okoz a rendszerben, akkor ésszerű, hogy egy másik környezetben elvégzett ugyanazon interakció ugyanazt a változást idézze elő. Ha mégsem így történik, akkor a felhasználó meglepetté és zavarttá válik, nem érti, miért nem úgy történnek a dolgok, ahogy azt a kiadott utasításai alapján gondolta, hogy történni fognak. Éppen ezért figyelni kell arra, hogy az összehasonlítható tevékenységek összehasonlítható eredményhez vezessenek.

A visszaállíthatóság elve azért fontos, mert a felhasználók a rendszer használata közben elkerülhetetlenül hibáznak. A felület tervének minimalizálni kell ennek a lehetőségét, azonban ez teljes mértékben úgysem fog sikerülni. Ezért biztosítani kell a felhasználó számára, hogy ha valamilyen hibás utasítást adott ki, akkor bizonyos keretek között visszavonhassa annak hatását, vagy a kritikus tevékenységek előtt a rendszer megerősítést kérjen a felhasználótól, ezzel esélyt adva neki, hogy észrevegye ha hibázott. Ezek mellett lehetővé teheti a rendszer ún. ellenőrzőpontok felállítását, amely során a rendszer adott időpontbeli konzisztens állapota

elmenthető, és hiba esetén az ellenőrzőpont utáni műveletek hatását semmissé téve visszatérhetünk arra a korábbi állapotra, amelyet az ellenőrzőpont tárol.

A felhasználót természetesen támogatni kell, a felületekhez készíteni kell beépített segítő rendszert, ún. helyzetérzékeny súgót, amely az elakadt és tanácstalan felhasználónak bizonyos szintű segítséget vagy útmutatást nyújt. Ez lehet csupán minimális információ egy gombhoz tartozó funkcióról, vagy egy mező kitöltésének szabályairól, egészen egy kimerítő leírásig a rendszer lehetőségeiről és funkcionalitásáról. Az ilyen fajta súgót strukturálni kell, nem szabad a felhasználót elárasztani túl sok információval. Azonban meg kell jegyezni, hogy ez a súgórendszer nem válthatja ki a rendszer működéséről szóló alapos dokumentációt, amely leírja a rendszer működését és felhasználását a felhasználók szemszögéből.

Végül pedig érdemes figyelembe venni a felhasználók sokféleségét: míg egyes felhasználók majd csak néha tévednek az oldalunkra, azok így igen kevés jártassággal fognak rendelkezni, ezért számukra olyan felület megfelelő, amely kellő útmutatást ad számukra a szükséges lépésekről és a lehetőségeiről. Ezzel szemben egy olyan felhasználó, akinek rendszeresen használnia kell az alkalmazást, már jól ismeri a rendszer működését, ezért számára a gyors előrehaladás a mérvadó, tehát számára mondjuk a gyorsbillentyűk alkalmazása lehet kívánatos, hogy a szükséges interakciókat minél gyorsabban el tudja végezni. Itt kell megemlíteni, hogy lehetnek olyan felhasználók is, akik valamilyen fogyatékossággal rendelkeznek, ezért a felületeket érdemes olyan képességekkel felruházni. amely számukra is segítséget nyújt, és a képességeikhez mérten ők is jól fel tudják használni az adott alkalmazást.

Ezzel gyakorlatilag csak érintettük a szükséges követelményeket, amelyek sokszor ellentmondásba kerülnek egymással, így a tervezők feladata dönteni, hogyan teremtik meg a szükséges egyensúlyt a különböző irányelvek között. Ehhez elemezni kell a rendszer felhasználóit, ismerni az igényeiket és jártasságukat a különböző rendszerek használatában, illetve tudni kell, hogy az információk megjelenítése milyen formában a legmegfelelőbb számukra. Ezek fényében kell kialakítani a különböző felületeket, megtervezni, hogy hogyan lesz képes a felhasználó interakciót végezni a felülettel, tehát hogyan tud parancsot adni annak. Ezt megteheti menük, űrlapok, vagy közvetlen a felületelemek manipulációjával, alkalmazhat valamilyen parancsnyelvet, amely akár egy természetes emberi nyelv is lehet. Meg kell határozni ezeknek a felületelemeknek a harmonikus elrendezését, úgy hogy az a legkényelmesebb legyen a felhasználó számára, azok a dolgok legyenek a szeme előtt amire figyelnie kell.

Továbbá az információkat valahogyan meg is kell jeleníteni, ehhez a feladathoz is rendkívül sok probléma tartozik. Megjeleníthetjük a dolgokat szövegesen, grafikusan, alkalmazhatunk különböző diagramokat, jelző elemeket. Döntenünk kell arról, hogy a megjelenítendő információ számára melyik mód a legcélravezetőbb, melyikből derül ki az olyan jellegű információ, amelyet a felhasználónak ismernie kell a munkája elvégzéséhez. Figyelni kell a megjelenő elemek stílusán, melybe az elemek elrendezése és színe is tartozik. A színek sokszor segítenek a felhasználónak, irányítja a figyelmét, és jelzi a lényeges információt tartalmazó részeket. Azonban ügyelnünk kell milyen színeket alkalmazunk, és hogy hogyan alkalmazzuk őket, mert amennyire segíthet annyira zavaró is lehet alkalmazásuk.

4.6 Tesztesetek tervezése

Mint ahogy minden munkát, amelyet az ember végez, úgy az elkészített szoftvereket is ellenőrizni, tesztelni kell, hogy megfelelőek-e a felállított követelményeknek. Természetesen nem érdemes „ad hoc” módon tesztelni, mert az nincs kellőképpen ellenőrizve, és az emberi lustaság hamar gondokat okozna ebben az esetben. Másrészt pedig már a készülő tervet is folyamatosan ellenőrizni kell, és lehetőleg minél hamarabb kiszűrni belőle a logikai ellentmondásokat és hibákat, mert annál kevesebb költséggel lehet a szükséges módosításokat végrehajtani. Ezért elvárható, hogy az ellenőrzési folyamatok a szoftverfejlesztés minden lépésében jelen legyenek. A követelmények vizsgálatával indul, és a tervezés áttekintésén valamint a kódvizsgálaton keresztül egészen a termék teszteléséig tart.

Ebben a részben magának a kódnak és az elkészült alrendszereknek a tesztelését szeretném kifejteni. Ezt a tevékenységet verifikációnak nevezzük, ami azt jelenti, hogy az elkészített program tényleg jól működik-e. Nem keverendő össze a validációval, amely az előzővel szemben azt ellenőrzi, hogy tényleg azt a szoftvert készítettük el, amit a megrendelő kért.

A verifikáció során ellenőrizni kell, hogy a követelményelemzéskor létrejött szoftverspecifikációnak megfelel-e a szoftver, teljesíti-e a benne megfogalmazott funkcionális és nemfunkcionális követelményeket. Ennek az ellenőrzésnek a tervezését már az elején el kell kezdeni, még akkor is ha tudjuk, hogy az ezzel járó költségek akár a teljes költség felét is kitehetik. Azonban egy rendszer minél kritikusabb annál több figyelmet kell fordítani a tesztteltelésre.

A tesztek létrehozása több forrásból lehetséges: a követelmények meghatározása után és az elkészített rendszerspecifikáció alapján elkészíthetőek azok a tesztterv, amelyeket majd

alkalmazhatunk az átadáskori tesztelésnél, a rendszer egészének tervezésekor felállíthatjuk azokat a teszteseteket, amelyek a helyes integrációt fogják ellenőrizni, illetve az egyes komponensek és alrendszerek részletes tervezésénél minden egyes egység számára meghatározhatunk teszteseteket, amelyekkel biztosítható, hogy a megfelelő funkcionalitást nyújtják. Ezentúl a tesztek tervezésének keretében olyan szabványos folyamatokat kell alkotnunk, amelyek segítségével az erre szánt erőforrások megfelelően kioszthatóak, becsülhetővé válik az egyes tesztelési ütemek erőforrásigénye, illetve segítenie kell a tervezésben és a tesztelésben résztvevőknek abban, hogy jobban el tudják képzelni az adott helyzetet és környezetet, amelyben a rendszert ténylegesen fogják használni.

A tesztek tervezésekor tudnunk kell, hogy mely követelményeket szeretnénk egy teszten belül ellenőrizni, tehát mely folyamatokat fogjuk vizsgálat alá vetni a teszt során, pontosan a rendszer mely részegységeit érinti a tesztelés, hogyan fog időben végbemenni a tesztelés, az egyes bemenő adatok esetén milyen kimenő adatokat várunk és meghatározhatjuk, hogy milyen hardvert és szoftvert igényel a teszt végrehajtása. Természetesen ezeket megfelelően dokumentálni kell, hogy később felhasználhatóak legyenek, és ezek alapján a megfelelő fejlesztők teszteseteket generálhassanak. Egy kis rendszer esetében ezek a tervek persze kevésbé formálisak, illetve megeshet az is, hogy nem is tudjuk őket élesen elválasztani a fejlesztéstől. A teszteket is lehet inkrementálisan végezni, általánosan elmondható, hogy a tesztek együtt fejlődnek a szoftverrel, hiszen számos befolyásoló tényezőt kell figyelembe venni: egy olyan komponens amely még nincs kész, az teljes egészében nem tesztelhető, ezért úgy kell módosítani a teszteket, hogy az csak a meglévő funkcionalitást tesztelje.

A tesztelés egyik fajtája az ún. száraztesztelés vagy más néven a szoftver átvizsgálása. Ekkor annyit teszünk, hogy a meglévő dokumentációkat, tervet vagy kódot átnézzük, abban hibák és hiányosságok után kutatva. Ehhez a fajta teszteléshez rendkívül sok tapasztalatra és széleskörű szakmai tudásra van szükség a tervvel, az alkalmazott programozási technológiákkal és magával a nyelvvel kapcsolatban. Mivel ez többnyire a meglévő kód statikus átnézését jelenti, több előnnyel is bír: a normál tesztelés során bizonyos hibák elfedhetnek másokat, így egy hiba kijavítása után nem tudni, hogy az újabb hiba tényleg új, vagy csak a javítás egy mellékhatása. Azonban száraztesztelésnél ezzel nem kell foglalkoznunk, egymás után több hibát is felfedezhetünk és kijavíthatunk. Ezzel a tesztelési módszerrel félig kész kódot is ellenőrizhetünk, hiszen nincs szükség speciális tesztelési környezetre, amely figyelembe veszi a szoftver hiányos voltát. Továbbá vannak egyszerű teszteléssel nem ellenőrizhető dolgok is,

mint az alkalmazni kívánt szabványok betartásának ellenőrzése, a kód hordozhatósága és karbantarthatósága. Ezeket a jellemzőket többnyire csak a kód átvizsgálásával tudjuk ellenőrizni, bár ma már léteznek bizonyos módszerek melyek segítenek ezek automatizálásában (mint például a kód stílusának ellenőrzése stb.) Látható, hogy számos előnye van a száraztesztelésnek azonban a normál tesztelést nem váltja ki teljesen, de a két technika egymás mellett való alkalmazásával kihasználhatjuk mindkét módszer előnyeit.

A másik fő típus a kód tényleges automatizált tesztelése, amelyhez valamilyen eszköz segítségével teszteket tudunk generálni. Ezen tesztesetek generálásához van szükség a tesztek tervezésére, amely feltárja a szükséges bemenő adatok halmazát és azt a halmazt amire a bemeneti halmaz leképeződik mint eredményhalmaz. A tesztek segítségével sok mindent tudunk ellenőrizni: elemezhetjük a vezérlés folyamatát, ennek segítségével feltárhatjuk azokat az útvonalakat, amelyeket a vezérlés ténylegesen bejárhat, így felderíthetjük azokat a kódrészleteket, amelyekre sosem jut el a vezérlés. Megtudhatjuk, hogy megfelelően vannak-e felhasználva a különböző változók, abból a szempontból, hogy melyek azok, amelyek nem kapnak kezdőértéket, vagy éppen deklarálva vannak, de sehol sem használjuk fel őket. Ellenőrizhetjük a különböző eljárások és függvények felhasználási módját, azaz hogy a hívások megfelelnek az adott függvény specifikációjának, nem hívunk-e olyan függvényt, amely nincs deklarálva (megjegyzendő, hogy ennek az elvégzése csak olyan gyengén típusos nyelvekben szükséges, mint a C vagy a FORTRAN; egyéb esetben a fordító elvégzi ezt az ellenőrzést). Továbbá felfedezhetjük az összes lehetséges végrehajtási útvonalat, és ezekhez olyan teszteseteket készíthetünk, amelyek lefedik az összes lehetséges útvonalat, vagy legalább azok nagy százalékát, ezzel biztosítva, hogy a lehető legtöbb lehetséges esetet ellenőriztük.

Ezen tesztek végrehajtásához már léteznek szabványos eszközök, mint például a jUnit, mely segítségével produktív módon írhatunk teszteseteket a különböző osztályaink számára, anélkül, hogy különösebb figyelmet vagy erőfeszítést kellene szentelnünk a tesztesetek automatizált végrehajtásának biztosításához. Így ténylegesen csak azzal kell foglalkoznunk, hogy a teszttervekben foglalt eseteket lefedjük, megadjuk valamilyen formában az ellenőrizendő bemenő adatokat és megadjuk, hogy ezekhez milyen eredményt várunk el. A feladat oroszlánrészét a már megírt keretrendszer elvégzi, így ha jól megírtuk a teszteseteinket, akkor addig tesztelhetünk majd a fejlesztés során, amíg minden teszteseten át nem megy az adott kód.

Azonban nem csak osztályokat kell tesztelni, hanem magasabb szinten is: a különböző komponensek funkcionalitását, illetve hogy megfelelnek-e a specifikált interfészeknek, a kü-

lönböző, egyedülállóan tesztelt komponensek és alrendszerek képesek-e együttműködni, és az elvárt viselkedést biztosítani. Ezek egy része szintén automatizálható, míg mások emberi beavatkozást igényelnek. Például ma már akár egy adott grafikus felület, akár egy weboldal funkcionalitása is ellenőrizhető ún. smoke-tesztek segítségével. Ennek során a tesztelő egy szoftver segítségével rögzíti a rendszerben végzett interakcióit, tehát hogy mikor melyik felületelemre kattintott és hogy utána mi történt. Ezek alapján összeállíthatóak különböző műveletsorok, és meghatározható hogy annak végrehajtása után a fejlesztőnek mit kell látnia a képernyőn. Ezzel automatizálható és többször végrehajtható lesz a teszt.

5 Implementálás

Sok minden történt azóta, hogy néhány embernek egy ötlet fogalmazódott meg a fejében, hogy elkészítsenek egy szoftvert. Formába öntöttük a kezdeti ötletet, tisztáztuk hogy milyen követelményeknek kell megfelelnie majd az alkalmazásnak, pontosan végiggondoltuk hogyan akarjuk megvalósítani majd azt, miként bontjuk fel részeire a teljes problémát, milyen ütemezés mellett akarjuk megvalósítani és milyen technológiákat akarunk felhasználni mindeközben. Ezután azt is megterveztük és leírtuk, hogy hogyan akarjuk majd kipróbálni a programot – ha már elkészült – milyen teszteket kell elvégeznünk, hogy megbizonyosodjunk róla, tényleg jól működik, amit elkészítettünk.

Ezzel gyakorlatilag eljutottunk ahhoz a fázishoz, amikor mindazt, ami eddig csak papíron tervekben létezett meg is valósítsuk. Mindenünk adott hozzá, ha megfelelő volt a dokumentáció, akkor minden félelem nélkül belekezdhetünk a dologba. Ugyan a munka egyik fele már készen van, a másik fele még hátra van: el kell készíteni magát a szűkebb értelemben vett szoftvert, a tényleges programot, amely majd elvégzi a rá bízott feladatokat, és kiszolgálja a megrendelő igényeit.

A megvalósításhoz is módszerek tömkelegéből választhatunk, egyre-másra jönnek létre az újabb szoftverfejlesztési módszertanok és megközelítési módok. Nem szeretném mindegyiket teljes egészében felsorolni, inkább csak érintőlegesen szólni róluk néhány szót. Érdemes ezeket tanulmányozni, de valószínűleg teljes egészében egyiket sem fogjuk alkalmazni, hiszen a használata során rájövünk melyek az előnyei és hátrányai számunkra, így egy idő után úgyis kialakul mindenkinek a saját módszertana, amely segítségével képes átlátni a folyamatot és megfelelő ütemben elvégezni a kitűzött feladatot.

Az utóbbi egy-két évtizedben világossá vált, hogy a fejlesztés egyik legszűkösebb erőforrása az idő. Egyre több vállalat modernizálta a működését, és ebben többnyire a számítógépesítés is nagy szerepet kapott. Azonban a gépeket programok vezérlik, és azokat valakiknek ki kell fejleszteni. A piaci versenyben pedig sokszor az lehet majd a győztes, aki hamarabb tud reagálni a világ változásaira, tehát minél előbb van kész a vállalat működését segítő szoftver, annál előbb képes a vállalat még hatékonyabban és pontosabban végezni a feladatát. Emellett az is megfigyelhető, hogy a piac túl gyorsan változik: mire elkészül egy programrendszer, amely egy adott feladatkört elvégez, addigra teljesen új igények merülnek fel, így kezdődhet elölről az egész fejlesztés, amely hamar a vég nélküliség látszatát kelti.

Ennek megfelelően alakultak ki a különböző gyors szoftverfejlesztési megközelítések, melyek azon alapulnak, hogy a fejlesztési fázisok nem szorosan egymás után történnek. A követelmények gyors és kiszámíthatatlan változása miatt, ha szigorúan betartanánk a fenti lépéseket, akkor két dolog történhetne: vagy minden egyes változást követnénk, akkor újra kellene kezdeni a teljes életciklust, így sosem jutnánk el a kész szoftverig, vagy figyelmen kívül hagynánk az új igényeket, ezzel viszont egy olyan szoftvert kapnánk, amely olyan funkcionalitást nyújt, melyre már nincs is szükség. Inkább az iteratív módszerekhez közelítenek ezek a fejlesztési folyamatok: a szoftvert szétszedik eléggé kis részekre, amelyeken belül a követelmények nagyjából állandóak, és ezt a kis részt fejlesztik ki mintha a teljes rendszer lenne. Itt helyet kap mindegyik fázis, majd ha elkészült a programrész, újabbat vesznek és kezdik elölről. Ezzel gyakorlatilag a teljes fejlesztést tekintve az egyes fázisok átfedik egymást, mintha a mindegyik érvényben lenne a rendszer életciklusának egészében.

Természetesen, ha az idő ellen kell dolgozni, akkor annak általában a munka alaposság lesz az áldozata: az egyes lépésekben elkészítendő specifikációk és leírások nem kell hogy teljesek legyenek, csak a legszükségesebb dokumentumokat készítik el, és amennyire csak lehet kihasználják azt, hogy a tervek alapján rengeteg kód legenerálható a megfelelő eszközök segítségével. Az egyes inkrementumokat lehetőleg olyan kicsire tervezik, hogy egy 10-20 fős fejlesztőcsapat képes legyen elvégezni a megvalósításhoz szükséges munkát, így nincs szükség túlságosan sok szervezésre, a csapaton belüli hatékony kommunikációt kell megoldani csupán. A felhasználói felületek létrehozását is valamilyen olyan fejlesztői környezet segítségével készítik el, ahol csak össze kell rakosgatni egy grafikus felületen keresztül a képernyő elemeit, és a szükséges kódot a rendszer generálja. Ez vonatkozik a webes alkalmazások felü-

leteire is, már léteznek olyan programok, amelyek segítségével ugyanilyen elven tudunk weblapokat készíteni.

Tehát a hagyományos módszerek segítségével nem lehetett a fejlődés ütemét tartani, ezért új módszereket dolgoztak ki, ekkor hozták létre az ún. agilis módszereket, melyek közös jellemzői, hogy a szoftverfejlesztési folyamatba bevonja a felhasználókat is, inkrementális módon fejleszt és adja át a készülő alkalmazást, nem alkalmazza mereven az egyes fejlesztési folyamatokat, megengedi, hogy a programozók a saját tapasztalataik alapján kidolgozott lépéseken keresztül valósítsák meg a projekt célját, fontos tényezője a fejlesztésnek a változtathatóság és a változékonyság, és az egyszerűségre koncentrál a megvalósításban és a folyamatok irányításában is.

Az agilis módszerek közül a legismertebb az extrém programozás. Ebben minden követelményt forgatókönyvként állítanak össze, amely közvetlenül feladatok egy soraként kerül implementálásra. A programozók párokban dolgoznak és minden feladatra teszteket készítenek, még mielőtt a kódot megírnák. Minden tesztnek sikeresen le kell futnia, mielőtt az új kódot integrálnák a teljes rendszerbe.

Említést érdemelnek még a gyors alkalmazásfejlesztési technikák, melyek az agilis módszerek előtt is léteztek már. A gyors alkalmazásfejlesztés (Rapid Application Development, RAD) a negyedik generációs nyelvekből fejlődtek ki, és adatintenzív alkalmazások fejlesztésére használják. Így olyan eszközkészlettel rendelkeznek, melyek adatok készítésére, kiírására, keresésére és jelentések generálására alkalmasak. Eszközeik között megtaláljuk az SQL-t, mint adatbázis-programozási nyelvet, mely segítségével közvetlenül is adhatunk parancsokat az adatbázisnak, vagy felhasználói űrlapok alapján is könnyek generálhatóak SQL utasítások. Tartalmaznak interfész-generátorokat, melyek segítségével könnyen és gyorsan készíthetünk felhasználói felületeket; jelentésgenerátorokat, melyek az adatbázisban lévő információk alapján hoz létre egy sablon segítségével megadott dokumentumokat, végül pedig szerves részét képezik olyan eszközök, melyek megkönnyítik különböző irodai alkalmazásokkal való kapcsolatok kiépítését, így például a szükséges numerikus feladatokat elvégezhetjük egy táblázatkezelő segítségével, vagy a jelentéseket sablonjait elkészíthetjük egy szövegszerkesztő alkalmazással.

Néhány esetben az inkrementális módszerek nem alkalmazhatóak a feladat jellegéből vagy a magától a szerződés kikötéseiből adódóan. Ekkor viszont a követelmények teljeskörű leírására van szükség. Ezek pontos megismeréséhez alkalmazható az inkrementális fejlesztés,

ha az eldobható prototípus-készítést választjuk a követelmények feltárásához. Azért hívják eldobhatónak, mert a prototípusok nem azért készülnek, hogy a megrendelő éles helyzetben is használja őket, csak arra, hogy a szoftvert elkészítő cég programozói jobban megértsék a problémát. Tehát a különböző tervezési lehetőségek kipróbálására készítik őket. Ezek nem mások, mint a rendszer egy kezdeti verziói, mely a funkcionalitás azon részét szolgáltatják, amelyeket a fejlesztők a legkevésbé értenek. Emellett a tervezési folyamatban felhasználhatóak az egyes szoftvermegoldások hatékonyságának vizsgálatára, a felhasználói felületek tervezésében szükséges tapasztalatok megszerzéséhez, végül pedig a tesztelési folyamatban is felhasználhatóak tesztek futtatására. A felhasználók is hamar egy olyan használható programhoz juthatnak, amit saját maguk kipróbálhatnak, kifejthetik róla véleményüket és eldönthetik, hogy az elméleti síkon felállított követelmények ténylegesen fogják-e segíteni a munkájukat illetve további követelmények megfogalmazására adhatnak nekik ötletet. Megfigyelhető, hogy a felhasználói felületek dinamikus természete miatt a szöveges leírások és statikus képernyőtervek nem megfelelőek a felhasználói felületekkel kapcsolatos követelmények pontos meghatározására, erre érdemes gyors prototípus-készítést alkalmazni, majd a felhasználók bevonásával értékelni a kapott eredményt.

6 Az eredmény

Az előzőek alapján látható, hogy ha komolyan gondoljuk a szoftverfejlesztést, akkor nem is annyira egyértelműek a dolgok, azonban minden lépésnek megvan a maga haszna, és ha ezeket jól fel tudjuk használni, akkor kevesebb meglepetés érhet minket, mintha csak egyszerűen nekiülne az ember és elkezdene kódolni.

Mindennek eredményeként pedig a kitűzött rendszert meg is lehet valósítani, ahogy én is tettem, így ebben a részben kicsit részletesebben szeretném bemutatni az egyes részeit, hogyan működik az elkészült alkalmazás, milyen funkcionalitást valósítottam meg, és melyek azok, amelyeket idő hiányából és a tapasztalatlanságomból kifolyólag nem sikerült elkészíteni. Természetesen az eddigiekben feltüntetett példák, mint azt bizonyára az olvasó is észrevette, az elkészített rendszerrel kapcsolatban készültek. Megpróbáltam minden dokumentumra példákat készíteni, és ha nem is mindet, de néhány darabot szemléltető jelleggel elhelyezni ebben a dolgozatban.

Mivel tanulmányaim során a Java technológiák megismerésére helyeztem a fő hangsúlyt, ezért mindenképp ilyen alapokon szerettem volna elkészíteni a célként kitűzött alkal-

mazást. Erre minden lehetőségem megvolt, hiszen a Sun megfelelően gazdag eszközökkel rendelkező keretrendszert biztosít webes alkalmazások fejlesztéséhez, amelyet Java Enterprise Edition-nek hívnak (J2EE röviden). Adatbázisként az Apache Derby nevű adatbázis alkalmazását használtam, mely az 1.6-os Java verzióban is helyet foglal immár. Az adatbázisréteg megvalósításához mindezek mellett Hibernate technológiát vettem igénybe, mely segítségével megfelelően el lehet fedni az adatbázis-kezelő specifikus hívásokat, ezzel is rugalmasabbá téve az alkalmazást. A megjelenítési rétegben természetesen Java Server Pages-t illetve tag library-ket használtam, továbbá Struts technológiát alkalmaztam, mely segítségével az MVC modellnek megfelelően meg tudtam valósítani a megjelenítés és az üzleti logika által reprezentált működtetés szétválasztását. Mindezeket egy Spring alapú üzleti logikai réteg fogja össze, az aspektusorientált megközelítésnek megfelelően és az egyes rétegek határain elhelyezkedő interfészeknek köszönhetően az egyes rétegek nincsenek szoros összeköttetésben, bármikor tetszőlegesen kicserélhetnénk őket egy másik implementációval.

A rendszer egyik alapkövét a legalsó réteg alkotja. Alatta a már említett Derby adatbázis fut, de mivel az adatbázis-kapcsolat egy konfigurációs fájl segítségével állítható be, ezért bármilyen más adatbázist is alkalmazhatunk a megfelelő beállítások megadásának segítségével. A 12. példában látható osztályok mind ún. adatszállító objektumok osztályai (data transfer object – DTO), melyek az adatbázis és a legalsó réteg között haladnak. Ezeknek megfelelően adatbázis táblák jöttek létre, melyek lefedik az egyes osztályok attribútumait. Azért nem tűntettem fel sehol relációs adatbázissémákat, mert az objektum-relációs leképezés a Hibernate-nek köszönhetően automatikus. Nekem csak a leképezéshez szükséges xml fájlokat kellett elkészítenem, amelyben megadtam, hogy milyen attribútumai vannak az egyes osztályoknak, és azok milyen típussal rendelkeznek (opcionálisan megadható, hogy milyen nevű oszlopokra és táblákra képezzen le). Ezek után a HibernateTools nevű eszközzel ezekből az xml-ekből legeneráltattam a megfelelő Java osztályok forráskódjait és az adatbázis táblák létrehozásához szükséges sémát. Ebben a sémában a használt adatbázis-kezelő számára megfelelő SQL DDL utasítások vannak, amelyek létrehozzák a szükséges táblákat. A köztük lévő kapcsolatokat is a Hibernate hozza létre, ezért van az, hogy objektum-referenciákat használhatunk és nem csak külsőkulcs hivatkozásokat.

Az adatbázis kapcsolat és a Hibernate használatának elfedésére implementáltam egy Data Access Object tervezési mintát, ezek az osztályok a dao csomagba kerültek. Ebben a csomagban található egy GenericDAO interfész, amely az összes DAO interfész közös őse, és

a közös metódusok specifikációját ide emeltem ki, mint például egy adott azonosítóval rendelkező objektum betöltése az adatbázisból, vagy az összes adott típusba tartozó objektum betöltése. Ezek minden osztály esetén ugyanúgy történnek, ezért így csak egyszer kell ezt implementálni, ezzel nő a karbantarthatóság. Megfigyelhető, hogy gyakorlatilag minden DTO-hoz tartozik egy DAO interfész, ezzel biztosítva, hogy minden egyes objektumot különkülön önmagában is be tudunk tölteni. Minden DAO interfész azokat a szükséges metódusokat definiálja, amelyek az adott osztály objektumai szempontjából fontosak, például a BoatDAO definiál egy adott típushoz tartozó összes szabad csónak listáját szolgáltató objektumot (ez a megjelenítéshez kell, a felhasználó csak olyan típusok közül választhat, amelyekhez tartozik szabad csónak), vagy a UserDAO-ben van olyan metódus, amely egy adott felhasználónévhez tartozó User objektumot adja vissza. Az impl csomagban természetesen ezen interfészeket implementáló osztályok találhatóak, illetve egy belső util csomagban egy HibernateUtil osztály, mely a Hibernate-tel való kapcsolattartást és a session-kezelést végzi.

Az üzleti logikai réteget a bs csomag valósítja meg. Ez is az előző réteghez hasonlóan jól definiált interfészhalmazzal rendelkezik. Ezek ún. ManagementService interfészek, amelyek egy adott feladatkörben kiszolgálják a felsőbb réteget az alatta lévő réteg segítségével. A szükséges funkcionalitásokat három területre osztottam fel. A csónakok és típusaik kezelésével foglalkozó BoatManagementService interfész, a foglalásokkal és a kölcsönzésekkel foglalkozó CharterManagementService és végül a felhasználói funkcionalitást kielégítő UserManagementService interfészek találhatóak ebben a csomagban. Ezzel gyakorlatilag lefedtem a szükséges szolgáltatásokat, amelyeket a megjelenítési réteg fog felhasználni. Ezeknek is megvan mind a megfelelő implementációjuk, illetve az exception csomagban egy BusinessServiceException osztály reprezentálja a lehetséges kivételek körét. Az egyszerűség kedvéért, ez egy azonosítóval látja el a különböző kivételeket, amelyek felmerülhetnek, és ezeket a kódokat csomagolja ebbe a kivételosztályba, melyet a felsőbb rétegben kiértékelhetünk. Ez egy egyszerű és igény szerint bővíthető módja a kivételkezelésnek.

A legfelső réteg a megjelenítési réteg jellegéből adódóan egy külön (webes) projektben foglal helyet. Mivel Struts-ot alkalmaztam ebben a rétegben, ezért ennek a technológiának a terminológiája és iránymutatása szerint alakítottam ki a projekt struktúráját. Ez egy Action osztállyal dolgozik, amely a különböző oldalakon lévő formokkal való interakciókat vezérlik. Minden egyes gombhoz tartozik egy form, és ezen form mögött egy megfelelő Action osztály leszármazottjának példánya áll. Ezek ActionBean-ekkel operálnak, amely úgy működik, hogy

például a bejelentkezésnél meg kell adni egy felhasználónevet és egy jelszót. Ezért van egy LoginFormBean, amely rendelkezik két példányváltozóval, melyek a beírt felhasználónevet és jelszót fogják tárolni. A bejelentkezés gombra való kattintáskor a háttérben egy ilyen LoginFormBean példány jön létre, amely fel van töltve a beírt adatokkal, és ez átadódik a LoginFormAction példánynak, ami kiértékeli a Bean-ben lévő adatokat, természetesen az üzleti réteg segítségével. Majd reagál valamilyen módon, ha a megadott authentikációs adatok megfelelőek, akkor a session-ben elhelyezi a felhasználót azonosító User objektumot, és visszairányítja arra az oldalra, ahol bejelentkezett a felhasználó, ha pedig nem helyesek a megadott adatok, akkor a hibaüzenetnek megfelelő azonosítót helyezi el a sessionben, és visszaküldi az aktuális oldalnak. A mögöttes weboldalak egyszerű jsp oldalak, amelyek Struts-os tag könyvtárakat használnak, így képesek megjeleníteni az Action objektumok műveleteinek eredményét.

Van még egy harmadik projekt is, amely egy Enterprise Archive (EAR) projekt, ez csak adminisztrációs célokat szolgál, mert a másik két projektet fogja egybe. Mindezt úgy, hogy elkészít egy ear állományt, amelyben megtalálhatóak a lefordított osztályok állományai, illetve a szükséges egyéb állományok (xml konfigurációs állományok, képek, css fájlok, jsp és html oldalak stb.) Ennek segítségével csak egyetlen állományt kell elhelyezni a webszerver megfelelő (deploy) könyvtárában, és már be is lehet tölteni az alkalmazást és használni.

Említést szeretnék még tenni a rétegek együttműködésének hátteréről. Ahhoz, hogy a rétegek ne legyenek szorosan összefogva, tehát egyszerűen ki lehessen venni bármelyik réteget és betenni egy másik implementációt a helyére, ahhoz készültek a megfelelő réteginterfész specifikációk illetve emellett a Spring keretrendszer nyújtotta lehetőségeket használtam fel. A Spring segítségével megvalósítható az irányítás „kifordítása” (inversion of control – IoC). Ezzel a technikával és az aspektusorientált szemléletmóddal elérhető, hogy egy adott objektum létrehozásakor vagy működésekor úgymond „beinjektáljuk” a szükséges egyéb objektumokat a műveletbe.

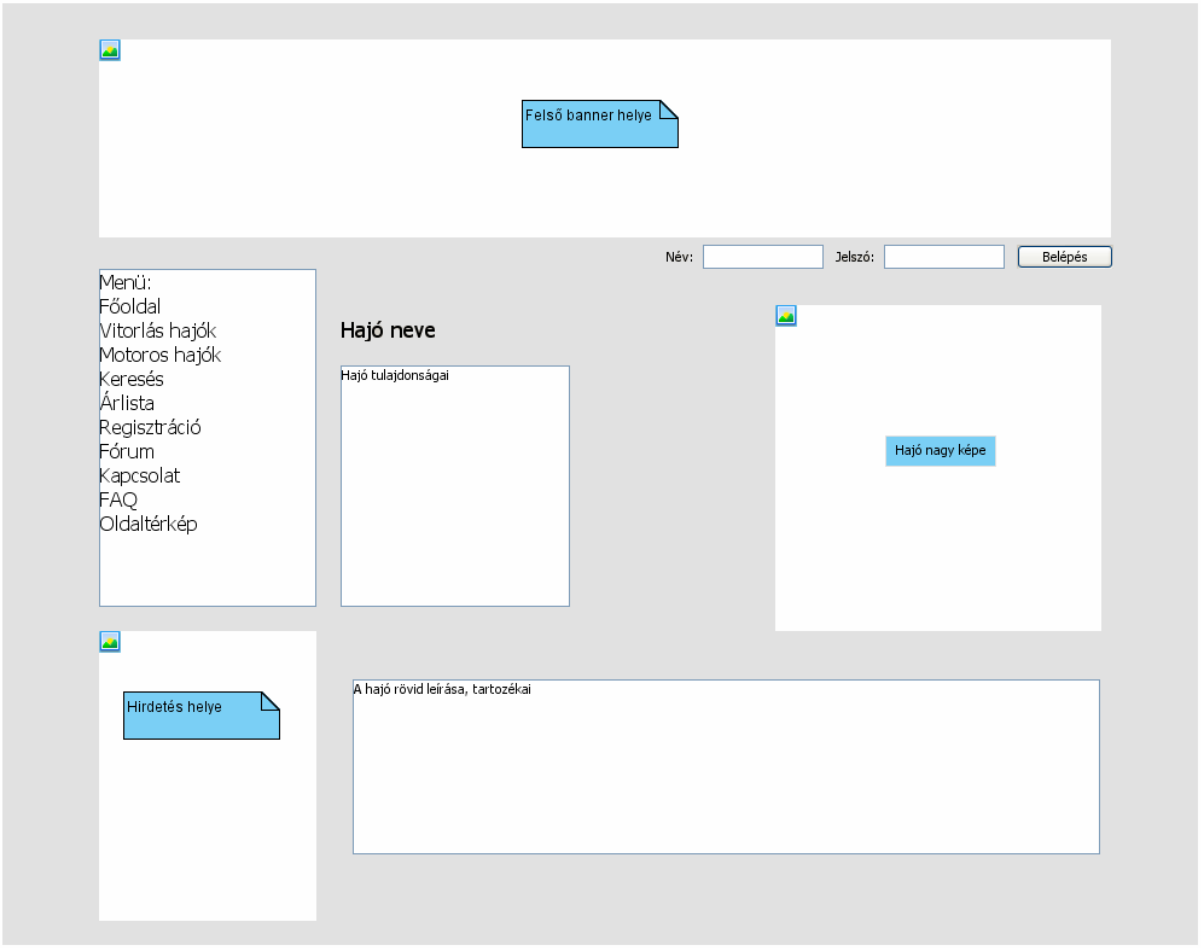
Tehát például ha a UserManagementService objektum számára szükség van egy UserDAO példányra, akkor azt nem az objektum keresi meg, hiszen ezzel szorosan összelánccolnánk a két réteget. Ehelyett a Spring-re bízzuk a probléma megoldását: a keretrendszertől kérjük el a működéshez szükséges UserManagementService objektumot, amely úgy példányosítja, hogy megad számára egy UserDAO objektumot is. Így használatra kész, de nem az objektumok döntik el, hogy melyik implementációt kapják meg, ezt a Spring végezte el, egy

megfelelő konfigurációs fájl megadásának segítségével. Így ha van egy új perzisztencia-réteg implementációnk, egyszerűen csak átírjuk a konfigurációs állomány megfelelő részeit és a működés mehet tovább, nem kell a kódhoz nyúlnunk. Ennek a lehetőségnek nagy jelentősége van a rendszer tesztelési időszakában is, hiszen ekkor az általunk meghatározott tesztkörnyezetben végezhetjük egy réteg vagy csak néhány objektum ellenőrzését, hiszen csak egy állományban kell átírni pár sort. Ha a tesztek sikeresen lefutottak, akkor az éles környezethez csak a megfelelő már elkészült rétegobjektumokat kell „hozzádrótozni" a saját implementációnkhoz.

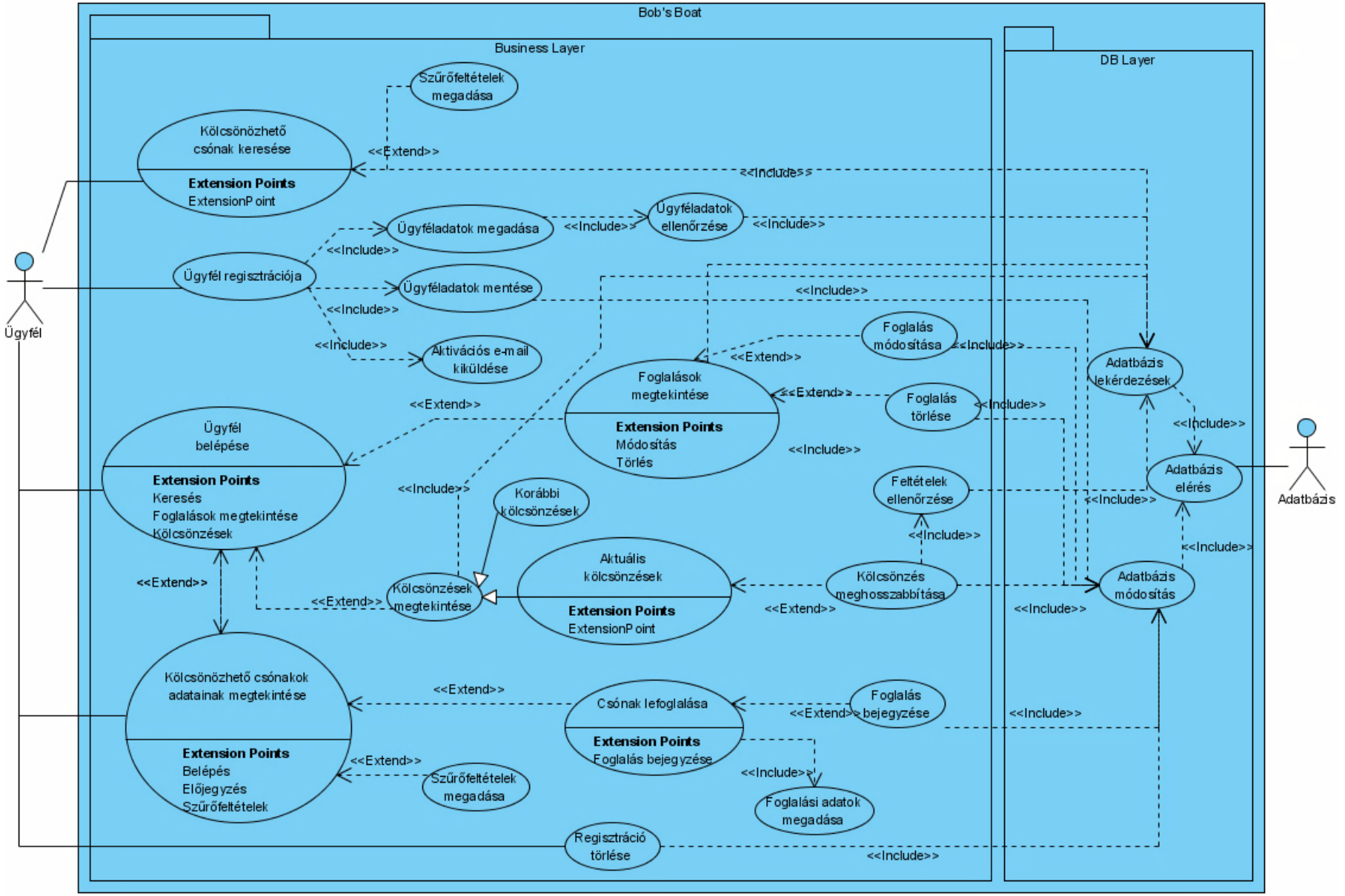
A fentiek segítségével a kitűzött feladatok nagy részét sikerült megvalósítanom. Az egyes csónakok böngészhetőek a főoldalon keresztül, keresni lehet közöttük a csónaktípus jellemzők alapján, a felhasználók tudnak regisztrálni, erről email értesítést is kapnak. A regisztrált felhasználó be tud jelentkezni, tud foglalásokat rögzíteni, illetve megtekintheti a jelenlegi és korábbi kölcsönzéseit, ugyanúgy tud keresni a csónakok között, mint azt már korábban ismertettem. Az oldalak fel vannak készítve arra, hogy stíluslapok segítségével formázást lehessen hozzájuk megadni, azonban ténylegesen designt csak a főoldalhoz készítettem. Amennyire csak lehetett, az XHTML szabványnak megfelelően készítettem el a kódokat. A dokumentációk nagy része is elkészült, bár a feladat egyszemélyes jellege és kis méretéből adódóan, sok dokumentumból és diagramból csak szemléltető darabok készültek, hogy a jelen dolgozatban tudjak róluk értekezni.

Amit nem sikerült megvalósítani: az alkalmazás főként az ügyfeleket kiszolgáló funkcionalitásokat valósítja meg, így nem készült el az alkalmazottak munkájához szükséges kliensprogram, bár ezen feladat igazából már kimutat a webes fejlesztés területéről. Továbbá csak ötlet szinten maradt egy webszolgáltatás megvalósítása, mely segítségével a motorcsónak adatbázis lekérdezhető lett volna, illetve a megadott jellemzők alapján keresni is lehetett volna benne. Illetve az oldal tartalmaz AJAX technológiával készült dinamikus elemeket, pedig a mai web 2.0-s oldalak egyik nagy vívmánya az aszinkron kommunikációban rejlő lehetőségek alkalmazását jelenti.

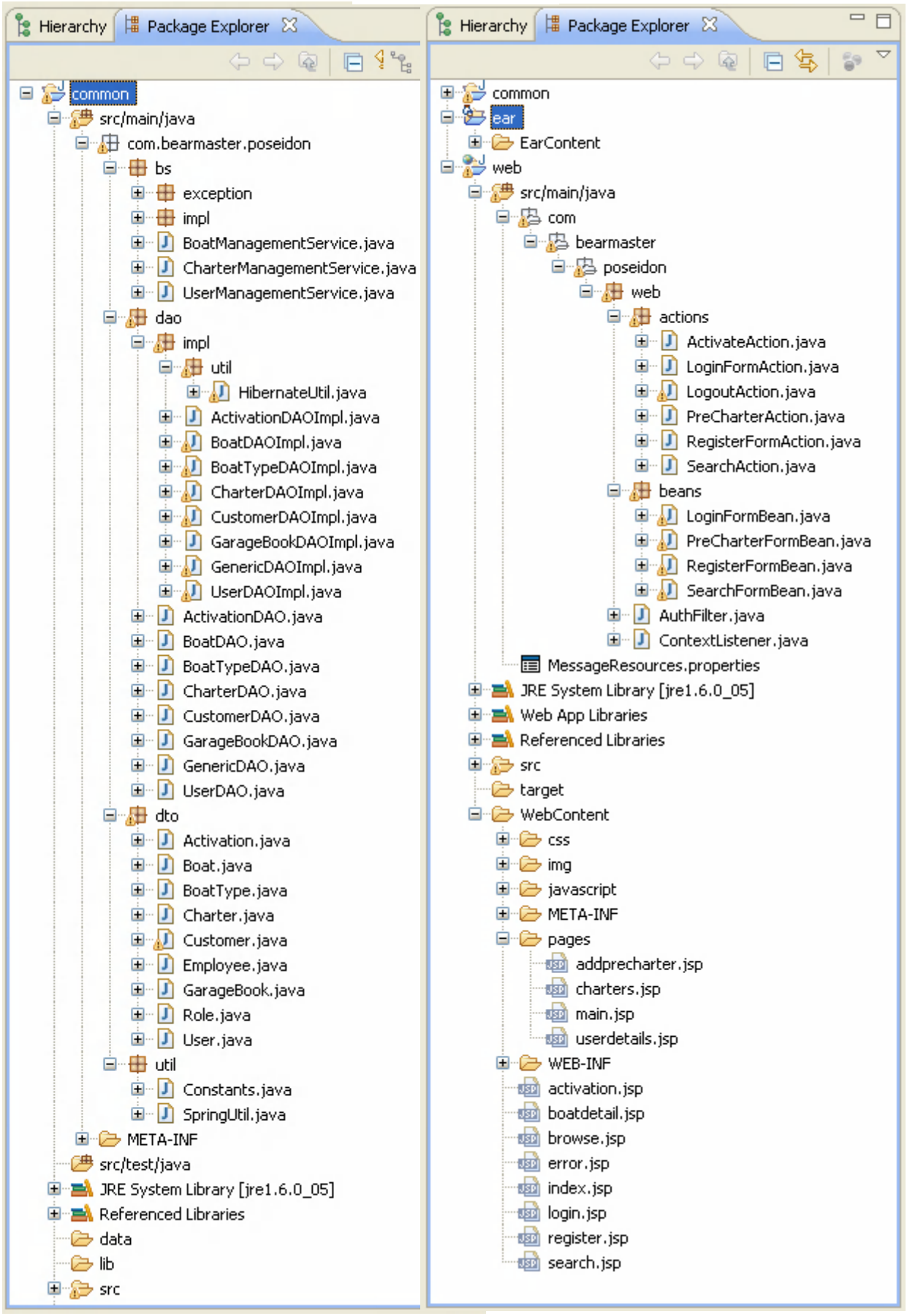
Függelék



1. Csónak részletei (felületvázlat)


Bob's Boat
Business Layer
DB Layer
Ügyfél
Adatbázis
Szűrőfeltételek megadása
Kölcsönözhető csónak keresése
Extension Points
ExtensionPoint
Ügyféladatok megadása
Ügyféladatok ellenőrzése
Ügyfél regisztrációja
Ügyféladatok mentése
Aktivációs e-mail kiküldése
Foglalás módosítása
Foglalások megtekintése
Extension Points
Módosítás
Törlés
Foglalás törlése
Ügyfél belépése
Extension Points
Keresés
Foglalások megtekintése
Kölcsönzések
Korábbi kölcsönzések
Feltételek ellenőrzése
Adatbázis lekérdezések
Adatbázis elérés
Adatbázis módosítás
Aktuális kölcsönzések
Extension Points
ExtensionPoint
Kölcsönzések megtekintése
Kölcsönzés meghosszabbítása
Kölcsönözhető csónakok adatainak megtekintése
Extension Points
Belépés
Előjegyzés
Szűrőfeltételek
Csónak lefoglalása
Extension Points
Foglalás bejegyzése
Foglalás bejegyzése
Szűrőfeltételek megadása
Foglalási adatok megadása
Regisztráció törlése
<<Extend>>
<<Include>>

Csomagszerkezet



Képernyőképek

Irodalomjegyzék

Ian Sommerville: Szoftver-rendszerek fejlesztése, Panem Könyvkiadó Kft., Budapest, 2007

Vég Csaba: Alkalmazásfejlesztés a Unified Modeling Language szabványos jelöléseivel, Logos 2000, 1999

Robert A. Maksimchuk, Eric J. Naiburg: UML földi halandóknak, Kiskapu Kft., 2006

Kondorosi Károly, Szirmay-Kalos László, László Zoltán: Objektum orientált szoftverfejlesztés, ComputerBooks, 2007

Craig Larman: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Addison Wesley Professional, 2004

Rational Unified Process Best Practices for Software Development Teams (A Rational Software Corporation White Paper)

http://en.wikipedia.org/wiki/Internet_protocol_suite

http://hu.wikipedia.org/wiki/ENIAC

http://wish.hu/cikkek/web20.html

http://en.wikipedia.org/wiki/Internet

http://en.wikipedia.org/wiki/IBM_PC

http://en.wikipedia.org/wiki/Use_case

http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/activity.htm

http://www.hik.hu/tankonyvtar/site/books/b10110/ch03s03s02.html

http://teszteles.blog.hu

Köszönetnyilvánítás

Köszönetet szeretnék mondani dr. Kuki Attilának, aki vállalta a dolgozatom témavezetését és javaslataival segítette a munkámat.

Továbbá köszönettel tartozom:

- *Nagy Andrásnak* és *Uzonyi Bélának*, hogy az ötlet alapjául szolgáló RFT gyakorlati anyagaikat a rendelkezésemre bocsátották,
- *Szakács Lászlónak* az RFT gyakorlaton közösen alkotott csoportunk eredményességéért, aminek nyomán a jelen dolgozatomnak ezt a témát választottam,
- *Pistár Zoltánnak*, hogy szobatársamként kitartott az utóbbi 3 évben, és szüntelen érdeklődésével késztetett engem is az új technológiák megismerésére,
- és végül de nem utolsó sorban *Deák Mariannak*, aki mindvégig bíztatott és segített átlendülni a problémás helyzeteken, és a megvalósítás során nyújtott lelkes segítségéért és megjegyzéseiért.