

4 A SZOFTVERFOLYAMAT

TÉMAKÖRÖK

Ennek a fejezetnek az a feladata, hogy bemutassa a szoftverfolyamatot, azaz tevékenységek egy olyan koherens halmazát, mely a szoftver előállításához szükséges. Ha elolvassuk ezt a fejezetet, az alábbiakkal leszünk tisztában:

- megértjük a szoftverfolyamat és a szoftverfolyamat-modell koncepcióit;
- megismerünk három különböző szoftverfolyamat-modellt és azok használatát;
- nagy körvonalakban megértjük a szoftver követelményeinek tervezéséhez, a szoftverfejlesztéshez, a teszteléshez és az evolúcióhoz szükséges folyamatmodelleket;
- megértjük, hogyan integrálódik a Rational Unified Process a szoftverfolyamattal, ezáltal létrehozva egy modern, általános folyamatmodellt;
- bevezetést nyerünk a szoftverfolyamatot támogató CASE-technológiák világába.

TARTALOM

- 4.1. A szoftverfolyamat modelljei
- 4.2. Folyamatiteráció
- 4.3. Folyamattevékenységek
- 4.4. A Rational Unified Process
- 4.5. Számítógéppel támogatott szoftverfejlesztés

A szoftverfolyamat tevékenységek és kapcsolódó eredmények sora, amelyek egy szoftvertermék előállításához vezetnek. Ezek a szoftverfejlesztés kezdeteitől, például egy egyszerű Java vagy C nyelven megírt programtól indulhatnak, de napjainkban gyakoribb eset, hogy az új szoftver fejlesztése meglévő rendszerek kiegészítése és módosítása, illetve kulcsrakész rendszerekkel történő integrálása és konfigurálása.

A szoftverfolyamatok – mint minden szellemi folyamat – összetettek és nagyban függnék emberi tényezőktől és döntésektől. Éppen azért, mert emberi beavatkozásokat és kreativitást igényelnek, a folyamatok automatizálására történő erőfeszítések csak korlátozottan arattak sikert. A CASE-eszközök (4.5. alfejezet) képesek arra, hogy a folyamat bizonyos tevékenységeit támogassák, de nincs reális lehetőség nagyobb mértékű automatizációra az elkövetkezendő néhány éven belül azokon a területeken, ahol a szoftverfolyamatban részt vevő szoftvertervezők kreativitására van szükség.

A CASE-eszközök korlátozott lehetőségeinek egyik oka a szoftverfolyamatok mérhetetlen sokszínűségéből adódik. Nincs ideális, mindenki számára megfelelő folyamat, és a különböző szervezetek a szoftverfejlesztést homlokegyenest különböző nézőpontokból közelítik meg. A folyamatokat úgy alakították ki, hogy kiaknázzák a szervezeten belül az emberek különféle képességeit és a fejlesztendő rendszer jellegzetességeit. Néhány rendszer esetében, például a kritikus rendszereknél, mélyen strukturált fejlesztési folyamatra van szükség. Az üzleti rendszerek esetében gyorsan változnak a követelmények, ezért egy flexibilis, agilis folyamat hatékonyabb lehet.

Habár számos különböző szoftverfolyamat létezik, vannak olyan alapvető tevékenységek, amelyek minden szoftverfolyamatban közősek:

1. *Szoftverspecifikáció.* A szoftver funkcióit, illetve annak megszorításait definiálni kell.
2. *Szoftvertervezés és implementáció.* A specifikációnak megfelelő szoftvert elő kell állítani.
3. *Szoftvervalidáció.* A szoftvert validálni kell, hogy biztosítsuk, azt fejlesztettük ki, amit az ügyfél kívánt.
4. *Szoftverevolúció.* A szoftvert úgy kell alakítani, hogy megfeleljen a megrendelő kívánása szerint történő változtatásoknak.

Ebben a fejezetben áttekintem ezeket a tulajdonságokat, a könyv későbbi részeiben pedig még bővebben foglalkozom ezzel a témával.

Habár nincs „ideális” szoftverfolyamat, számos olyan terület van, ahol a szervezeteken belüli szoftverfolyamatokon javíthatunk, ugyanis azok gyakran tartalmaznak elavult technikákat, vagy nem a legjobban használják ki a folyamatok lehetőségeit az ipari szoftvertervezésben. Továbbá, egy sor szervezet nem él a szoftvertervezési módszerek adta lehetőségekkel a fejlesztéseknél.

A szoftverfolyamatok számos különböző módon fejleszthetők tovább. Akár a folyamat szabványosításával is kezdhetjük, azaz a szervezeten belüli szoftverfolyamatok sokféleségének csökkentésével. Ezzel javul a kommunikáció, csökken a tréningek ideje, és gazdaságosabbá válik az automatikus folyamatámogatás. A szabványosítás – mint az új szoftvertervezési módszerek és technikák, valamint jó tervezői praktikák bevezetésének első lépése – elengedhetetlen. A szoftverfolyamat továbbfejlesztésének témájához a 28. fejezetben még visszatérek.

4.1. A SZOFTVERFOLYAMAT MODELLJEI

Ahogy azt az 1. fejezetben írtam, a szoftverfolyamat modellje a szoftverfolyamat absztrakt reprezentációja. Minden egyes folyamatmodell különböző speciális perspektívából reprezentál egy folyamatot, de ily módon csak részleges információkkal szolgálhat magáról a folyamatról. Ebben az alfejezetben számos, nagyon általános folyamatmodellt mutatunk be (néhányiket akár *folyamatparadigmának* is nevezhetnénk), architekturális nézőpontból szemlélve őket. Azaz látni fogjuk a folyamatok szerkezetét, de nem részletezzük azok speciális tevékenységeit.

Ezek az általános modellek nem a szoftverfolyamat pontos, végleges leírásai. Sokkalta inkább hasznos absztrakciók, amelyeket a szoftverfejlesztés különböző megközelítési módjainak megértéséhez használhatunk. Tekinthetjük úgy is őket, mint folyamat-keretrendszereket, melyek kiegészíthetők és adaptálhatók, ezáltal létrehozva egy konkrétabb szoftvertervezési folyamatot.

Ebben a fejezetben az alábbi szoftverfolyamatokat mutatom be:

1. *A vízesésmodell.* Ez a folyamat alapvető tevékenységeit a folyamat különálló fázisaiként tekinti, ezek a fázisok a követelményspecifikáció, a szoftvertervezés, az implementáció, a tesztelés stb.
2. *Evolúciós fejlesztés.* Ez a megközelítési mód összefésüli a specifikáció, a fejlesztés és a validáció tevékenységeit. Az absztrakt specifikációból gyorsan ki-fejleszthető egy kezdeti rendszer. A későbbiekben ezt a rendszert kell a megrendelő inputjából továbbfinomítani úgy, hogy az kielégítse az ügyfél kívánságait.
3. *Komponensalapú fejlesztés.* Ez a megközelítési mód a nagy mennyiségű újrafelhasználható komponensek létezésén alapszik. A rendszerfejlesztési folyamat ezeknek a komponenseknek rendszerré történő integrációjára összpontosít ahelyett, hogy az alapoktól fejlesszék ki azokat.

Jelen pillanatban a szoftvertervezői gyakorlatban ez a három általános modell terjedt el széles körben. Nem kizárólagos a használatuk, és gyakran keverednek is egymással, főként a nagy rendszerek fejlesztésekor. A 4.4. alfejezetben tárgyalt Rational Unified Process kombinálja is ezeknek a modelleknek az elemeit. Egy nagy rendszer alrendszeit különböző megközelítési módok alapján is kifejleszthetik. Habár ezeket a modelleket külön-külön érdemes tárgyalni, ne felejtjük el, hogy a gyakorlatban gyakran összefonódnak.

A fenti általános modellek különféle változatait használhatják a különféle szervezetek. A legfontosabb változatuk azonban talán a formális szoftverfejlesztés, ahol formális matematikai eszközökkel modellezzük a létrehozandó rendszert. Ezt a modellt majd matematikai transzformációkkal futtatható kóddá alakítjuk, megőrizve annak konzisztenciáját.

A formális rendszerfejlesztés legismertebb példája a Cleanroom folyamat, amelyet eredetileg az IBM-nél fejlesztettek ki (Mills és mások, 1987; Selby és mások, 1987; Linger, 1994; Prowell és mások, 1999). A Cleanroom folyamat esetében minden egyes szoftverinkremenst formálisan specifikálnak, és azt majd egy transz-

formáció segítségével implementálják. A szoftver helyességét a formális megközelítéssel bizonyíthatjuk. Nincs szükség a folyamat tesztelésére, és a rendszert is a megbízhatóságra összpontosít.

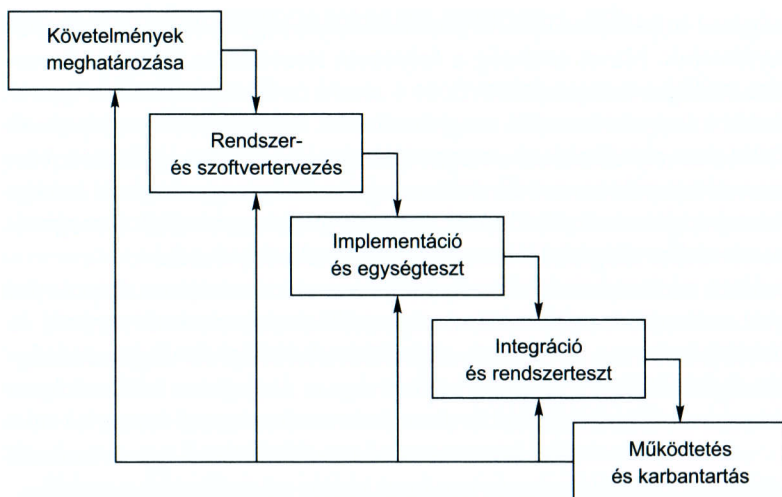
A Cleanroom és az egyéb formális megközelítések a B módszeren alapulnak (Wordsworth, 1996), amely alkalmas a szigorú értelemben vett megbízható, biztonságos rendszerek fejlesztésére. A formális megközelítés leegyszerűsíti a megbízhatósági és biztonsági esetek előállítását is, amelyek demonstrálják a megrendelő felé, hogy a rendszer megfelel a biztonsági követelményeknek.

Ezekon a speciális területeken kívül a formális transzformációkon alapuló folyamatok nem túl széles körben használtak. Speciális szakértelmet kívánnak, és a rendszerek többségében nem nyújtanak szignifikáns költségbeli vagy minőségbeli előnyt más megközelítési módokhoz képest.

4.1.1. A vízesésmodell

A szoftverfejlesztés folyamatának első publikált modellje más tervezői modellekből származik (Royce, 1970). Ezt illusztrálja a 4.1. ábra. Mivel az egyes fázisok lépcsősen kapcsolódnak egymáshoz, a modell vízesésmodellként vált ismertté, de nevezik szoftveréletről ciklusnak is. A modell alapvető szakaszai alapvető fejlesztési tevékenységekre képezhetők le:

1. *Követelmények elemzése és meghatározása.* A rendszer szolgáltatásai, megkorlátásai 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 a szoftverkövetelmények. Itt kell kialakítani a rendszer átfogó architektúráját. A szoftver tervezése az alapvető szoftverrendszer-absztrakciók, illetve a közöttük levő kapcsolatok azonosítását és leírását is magában foglalja.
3. *Implementáció és egységteszt.* Ebben a szakaszban a szoftverterv programok, illetve programegységek halmazaként realizálódnak. 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 követelményeknek. A tesztelés után a szoftverrendszer átadható az ügyfélnek.
5. *Működtetés és karbantartás.* Általában (bár nem szükségszerűen) ez a szoftver életciklusának leghosszabb fázisa. Megtörténik a rendszertelepítés és megtörténik 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.



4.1. ábra. A szoftver életciklusa

A fázisok eredménye tulajdonképpen egy vagy több dokumentum, amelyek jóváhagyása megtörtént („aláírták”). A következő fázis addig nem indulhat el, amíg az előző fázis be nem fejeződött. A gyakorlatban persze ezek a szakaszok átfedhetik egymást, és nemcsak a következő fázishoz szolgáltathatnak információt, hanem az összes többi fázishoz is. A tervezés folyamán azonosíthatunk a követelmények definiálásakor elkövetett hibákat, az implementáció során tervezési problémákat találhatunk stb. A szoftverfolyamat nem egyszerű lineáris modell, hanem a fejlesztési tevékenységek iterációjának sorozata.

A dokumentumok jóváhagyásának és előállításának költségéből adódóan az iterációk költségesek, és jelentős átdolgozásokat kívánnak. Ezért megszokott, hogy már kisszámú iteráció után is befagyasztják az adott fejlesztési fázist (például a specifikációt), és a fejlesztést későbbi fázisokkal folytatják. A problémák feloldását későbbre halasztják, kihagyják vagy kikerülik azokat. A követelményelemzés ilyen idő előtti befagyasztása oda vezethet, hogy a rendszer nem azt fogja tenni, amit a felhasználó akart. Vagy akár oda is vezethet, hogy a szoftver rosszul strukturált lesz, azaz implementációs fogásokkal próbáljuk kikerülni a tervezésembeli problémákat.

Az életciklus utolsó szakaszában (működtetés és karbantartás) a felhasználók használatba veszik a szoftvert. Ilyenkor derülnek ki az eredeti rendszerkövetelmények hibái és hiányosságai. Program- és tervezési hibák kerülnek elő, továbbá felmerülhet új funkciók szükségessége is. Ezekből adódóan a rendszert át kell alakítani oly módon, hogy az használható maradjon. Az ilyen változtatások (szoftver karbantartása) néhány, vagy akár az összes korábbi folyamatszakasz megismétlésével is járhatnak.

A vízesésmodell problémáját a projekt szakaszainak különálló részekké történő, nemflexibilis particionálása okozza. A folyamat korai szakaszaiban kell állást foglalnunk és elköteleznünk magunkat, ezzel nehezebbé válik az ügyfél köve-

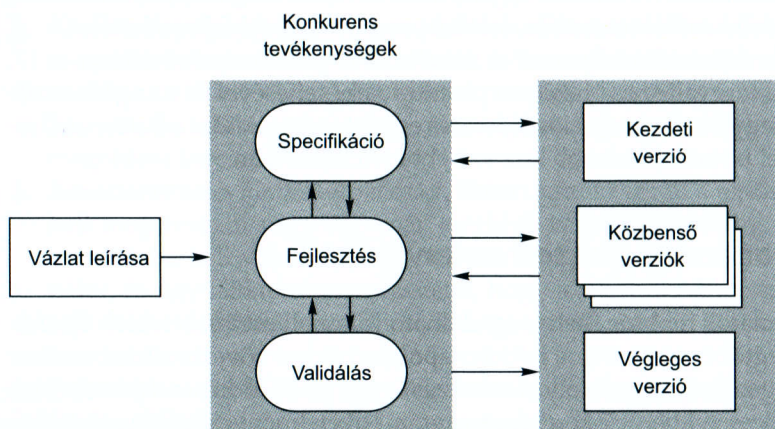
telményeinek megváltozásához történő alkalmazkodás. Mindent összevetve a vízesésmodell csak akkor használható jól, ha már előre jól ismerjük a követelményeket. Akárhogy is, a tervezők gyakorlatából visszatükröződik a vízesésmodell, vagyis arra enged következtetni, hogy a szoftverfejlesztésekben még mindig az ilyen megközelítési módokon alapuló szoftverfolyamatokat alkalmazzák, főleg ha azok egy nagyobb rendszertervezési projekt részei.

4.1.2. Evolúciós fejlesztés

Az evolúciós fejlesztés alapötlete az, hogy ki kell fejleszteni egy kezdeti implementációt, azt a felhasználókkal véleményeztetni, majd sok-sok verzión keresztül addig finomítani, míg a megfelelő rendszert el nem értük (4.2. ábra). A szétválasztott specifikációs, fejlesztési és validációs tevékenységekhez képest ez a megközelítési mód sokkal jobban érvényesíti a tevékenységek közötti párhuzamosságot és a gyors visszacsatolásokat.

Az evolúciós fejlesztéseknek két különböző típusa ismert:

1. *Feltáró fejlesztés.* Ebben az esetben a folyamat célja, hogy a megrendelővel együtt feltárjuk a követelményeket, és kialakítsuk a végleges rendszert. A fejlesztés a rendszer már ismert részeivel kezdődik. A végleges rendszer úgy alakul ki, hogy egyre több, az ügyfél által kért új tulajdonságot társítunk a már meglévőkhöz.
2. *Eldobható prototípus készítése.* Ekkor az evolúciós fejlesztés célja az, hogy a lehető legjobban megértsük az ügyfél követelményeit, és azokra alapozva pontosabban definiáljuk azokat. A prototípusnak pedig próbálgatásos módszerrel az ügyfél által meghatározott követelmények azon részeire kell koncentrálnia, amelyek kevésbé érthetők.



4.2. ábra. Evolúciós fejlesztés

A szoftverfejlesztés evolúciós megközelítése hatékonyabb, mint a vízésés-modell, ha olyan rendszert szeretnénk fejleszteni, amely közvetlenül megfelel az ügyfél kívánságainak. Az evolúciós megközelítési módon alapuló szoftverfolyamatok további előnye, hogy a rendszer-specifikáció inkrementálisan fejleszthető. Ahogy egyre jobban megértjük a felhasználók problémáit, annál jobban tükröződhetnek azok a szoftverrendszerben. Mindazonáltal a vezetőség és a tervezők szemszögéből három probléma merülhet fel:

1. *A folyamat nem látható.* A menedzsereknek rendszeresen leszállítható részeredményekre van szükségük, hogy mérhessék a fejlődést. Ha a rendszert gyorsan kell kifejleszteni, nem költséghatékony olyan dokumentációt előállítani, amely a rendszer összes verzióját tükrözi.
2. *A rendszerek gyakran szegényesen strukturáltak.* A folyamatos változtatások lerontják a szoftver struktúráját. A szoftver változtatásainak összevonása pedig egyre nehezebbé válik és nagyon költséges.

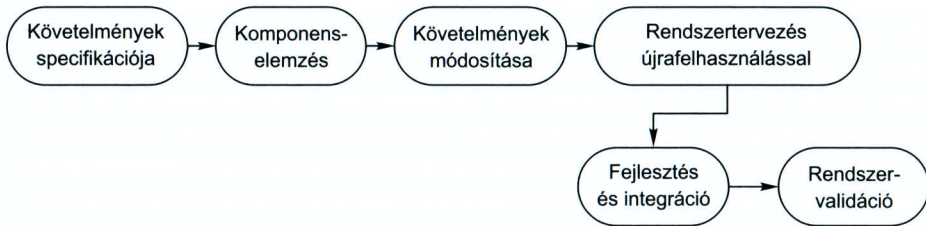
Úgy hiszem, hogy a várhatóan rövid élettartamú kis vagy a közepes rendszerek esetében is (500 000 programsorig) az evolúciós megközelítési mód a legcélravezetőbb. A nagy, hosszú élettartamú rendszerek esetében, ahol a különböző részeket különböző csoportok fejlesztik, az evolúciós fejlesztés problémái válságossá válhatnak. Ekkor ugyanis nehézkes egy stabil rendszer-architektúra létrehozása, és a csapatok közötti közreműködés megteremtése.

A nagy rendszerek számára egy „keverék” folyamatot javasolok, amely a vízésés- és az evolúciós fejlesztési modellek legjobb tulajdonságait ötvözi. Ez akár egy evolúciós megközelítési módon nyugvó eldobható prototípus készítését is magában foglalhatja, mely megoldja a specifikáció kérdéses problémáit, majd később ez a rendszer strukturáltabb megközelítési módot használva újra implementálható. A rendszer azon részei pedig, melyek előre jól ismertek, vízésésalapú folyamatban fejleszthetők. A rendszer egyéb, előre nehezen meghatározható részei, mint például a felhasználói interfész, feltáró programozással fejleszthetők.

Az evolúciós fejlesztési folyamat és annak támogatása részletesebben a 17. fejezetben kerül tárgyalásra, a rendszerprototípus készítésével és az agilis szoftverfejlesztéssel együtt. Az evolúciós fejlesztés együtt tud működni a Rational Unified Process-szel is, amiről később lesz szó ebben a fejezetben.

4.1.3. Komponensalapú szoftvertervezés

A szoftverfolyamatok többségében megtalálható valamelyest a szoftverek újr felhasználása. A legtöbb esetben ez a hétköznapiakban akkor következik be, amikor a projekten dolgozók ismernek olyan tervezést vagy akár kódot, mely hasonlít a kívánthoz. Ilyen esetekben előkeresik azokat, a követelményeknek megfelelően módosítanak rajtuk, majd beledolgozzák a rendszerbe. A 4.1.2. alfejezetben tárgyalt evolúciós megközelítésben az újr felhasználás nagyon gyakran szükséges a gyors fejlesztéshez.



4.3. ábra. Újrafelhasználás-orientált fejlesztés

Ez a hétköznapi újrafelhasználás függetlenül az azt felhasználó általános folyamatoktól is megtörténhet. Az elmúlt néhány évben jelent meg és kezdett el széles körben terjedni egy olyan szoftverfejlesztési megközelítési mód (a komponensalapú szoftvertervezés, CBSE), mely az újrafelhasználáson alapszik. Erről bővebben a 19. fejezetben is olvashatunk.

Az újrafelhasználás-orientált megközelítési mód nagymértékben az elérhető újrafelhasználható szoftverkomponensekre, illetve azok egységes szerkezetbe történő integrációjára támaszkodik. Néha ezek a komponensek saját létjogosultsággal rendelkező rendszerek, melyeket arra használhatunk, hogy biztosítsuk a rendszer specifikus funkcióit (mint például szövegformázás, numerikus számítások stb.). Az újrafelhasználás-orientált fejlesztés általános folyamatmodellje a 4.3. ábrán látható.

Amíg e kezdeti követelményspecifikációs és a validációs szakasz összehasonlítható más folyamatokkal, addig a közöttük található szakaszok az újrafelhasználás-orientált fejlesztésekben különböznek. Ezek a szakaszok a következők:

1. *Komponenselemzés.* Adott a követelményspecifikáció, és keresést kell végrehajtani, hogy mely komponensek implementálták azokat. A legtöbb esetben nincs egzakt illeszkedés, és a felhasznált komponens a funkcióknak csak egy részét nyújtja.
2. *Követelménymódosítás.* Ebben a fázisban elemezni kell a követelményeket, a megtalált komponensek információit felhasználva. Ezek után módosítani kell azokat az elérhető komponenseknek megfelelően. Ahol a módosítás lehetetlen, ott újra a komponenselemzési tevékenységet kell elővenni, és alternatív megoldást keresni.
3. *Rendszertervezés újrafelhasználással.* Ebben a szakaszban a rendszer szerkezetét kell megtervezni vagy egy már meglévő vázat felhasználni. A tervezőknek számításba kell venniük, hogy milyen komponenseket akarnak újrafelhasználni, és úgy alakítani a szerkezetet, hogy azok működhessenek. Ha nincsenek elérhető újrafelhasználható komponensek, akkor új szoftverek is kifejleszthetők.
4. *Fejlesztés és integráció.* A nem megvásárolható komponenseket ki kell fejleszteni, és a COTS-rendszerekkel egy rendszerbe integrálni. A rendszer-integráció ebben a modellben sokkal inkább tekinthető a fejlesztési folyamat részének, mint különálló tevékenységnek.

A komponensalapú modell nyilvánvaló előnye, hogy csökkenti a kifejlesztendő szoftverek számát, így csökkentve a költségeket, illetve a kockázati tényezőket. Így a legtöbb esetben a rendszer gyorsabban leszállítható. Viszont a követelményeknél elkerülhetetlenek a kompromisszumok, amelyek oda vezethetnek, hogy a rendszer nem felel meg a felhasználó valódi kívánságainak.

A CBSE egyre inkább kezd elterjedni a webszolgáltatásokat nyújtó rendszerek fejlesztése esetén. A szolgáltatás-központú fejlesztési megközelítési módokról a 12. fejezetben bővebben olvashatunk.

4.2. FOLYAMATITERÁCIÓ

A nagy rendszerek esetében a változtatások elkerülhetetlenek. Változhatnak a követelmények, az üzletmenet, vagy akár előfordulhat az is, hogy a rendszernek külső behatásokhoz kell alkalmazkodnia. Ahogy egyre több új technológia válik elérhetővé, változik a tervezés és az implementáció is. Ez azt jelenti, hogy a szoftverfolyamat nem egy egyszerű folyamat, hanem sokkal inkább a folyamattevékenységek rendszeresen ismétlődő folyamata, mely hatására a rendszert mindig átdolgozzuk az igényelt változtatások szerint.

Az iteratív fejlesztés alapjairól bővebben a 17. fejezetben olvashatunk. Ebben a fejezetben csak két különböző hibrid modellt mutatunk be, amelyek különféle megközelítési módokon alapulnak, és közvetlenül a folyamatiteráció támogatására készültek. Ezek:

1. *Inkrementális fejlesztés.* Ekkor a szoftverspecifikáció, a tervezés és az implementálás kis inkrementációs lépésekre (ún. inkremensekre) vannak felosztva, amelyeket több fordulóban végzünk el.
2. *Spirális fejlesztés.* Ebben az esetben a rendszer fejlesztése egy belülről kifelé tartó spirálvonalat követ az első vázlatától a végleges, kifejlesztett rendszerig.

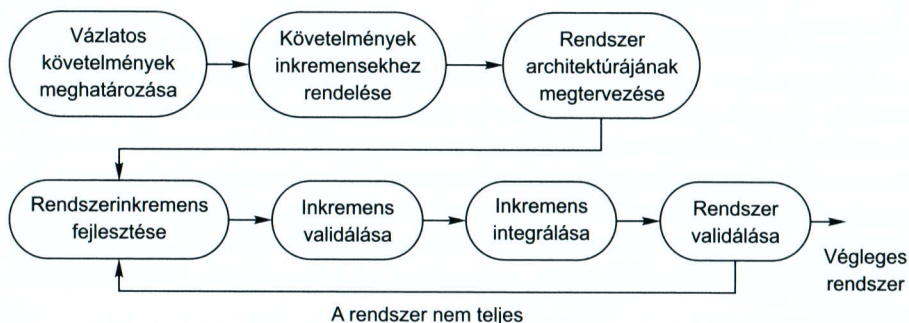
Az iteratív folyamat lényege, hogy a specifikációt a szoftverrel összekapcsolva kell fejleszteni. Habár ez így összeütközik a legtöbb szervezet beszerzési modelljeivel, ahol a komplett rendszer-specifikáció a rendszerfejlesztési szerződés része. Az inkrementális megközelítési módban nincs teljes rendszer-specifikáció addig, amíg a végső inkremens nem specifikált. Így újfajta megállapodási formákra van szükség, amelyeket a nagy megrendelők, mint például a közigazgatás, nehezen alkalmazhatónak fognak találni.

4.2.1. Inkrementális fejlesztés

A vízesésmodell azt követeli meg az ügyféltől, hogy véglegesítse a követelmények halmazát, mielőtt a tervezési folyamat elindulna, a tervezőtől pedig azt, hogy az implementáció előtt válasszon ki bizonyos tervezési stratégiákat. A fej-

lesztés ideje alatt a követelményekben bekövetkezett változások szükségessé teszik a követelményelemzés, a tervezés és az implementáció átdolgozását. Habár a vízesésmodell előnye, hogy egyszerűen menedzselhető, 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 elhasszunk a követelményekkel és a tervezésekkel kapcsolatos döntéseket, de ezek gyengén strukturált és nehezen megérthető és nehezen karbantartható rendszerekhez vezethetnek.

Az inkrementális megközelítési mód (4.4. ábra) egy köztes megközelítési módnak tekinthető, amely kombinálja a többi modell előnyeit. Egy inkrementális fejlesztési folyamatban a megrendelő nagy körvonalakban meghatározza a rendszer által nyújtandó szolgáltatásokat. Azt is meghatározza, hogy mely szolgáltatások a fontosabbak, melyek a kevésbé fontosak. Számos inkrementációs lépés csak később határozódik meg azáltal, hogy minden egyes inkrementáció funkciók egy-egy újabb halmazát adja a már meglévőkhöz. A szolgáltatások inkrementekben történő elhelyezése függ a szolgáltatás prioritásától is. A magas prioritású szolgáltatásokat hamarabb kell biztosítani a megrendelő felé.



4.4. ábra. Inkrementális fejlesztés

Amennyiben a rendszer inkrementációs lépéseit meghatároztuk, az első inkrementációs lépés által előállítandó szolgáltatások követelményeit részletesen definiálni kell, majd következhet az első inkremens fejlesztése a legmegfelelőbb fejlesztési folyamat felhasználásával. A fejlesztés ideje alatt a későbbi inkrementációs lépések további követelményeinek elemzésére is sor kerülhet, de az aktuális inkrementációs lépés követelményei nem változtathatók.

Amennyiben egy inkremens elkészült és átadták, úgy azt a megrendelő akár működtetheti is. Ez azt jelenti, hogy a rendszer bizonyos funkcióit korábban is beindíthatják. Tapasztalatot szerezhhetnek a rendszerrel kapcsolatban, mely aztán később segítségükre lehet a későbbi inkrementációs lépésekkel kapcsolatos követelmények tisztázásában vagy az aktuális inkrementációs lépés későbbi verzióinak követelménymeghatározásában. Amennyiben egy új inkremens elkészül, úgy azt integrálni kell a már meglévő inkrementekkel, így a rendszerfunkciók

köre minden egyes inkremenssel tovább bővül, javul. A közös szolgáltatások a folyamat korai szakaszában implementálhatók, vagy akár inkrementálva is megvalósíthatók, ahogy az egyes inkremensek funkciói megkívánják.

Az inkrementációs fejlesztési folyamatnak számos előnye van:

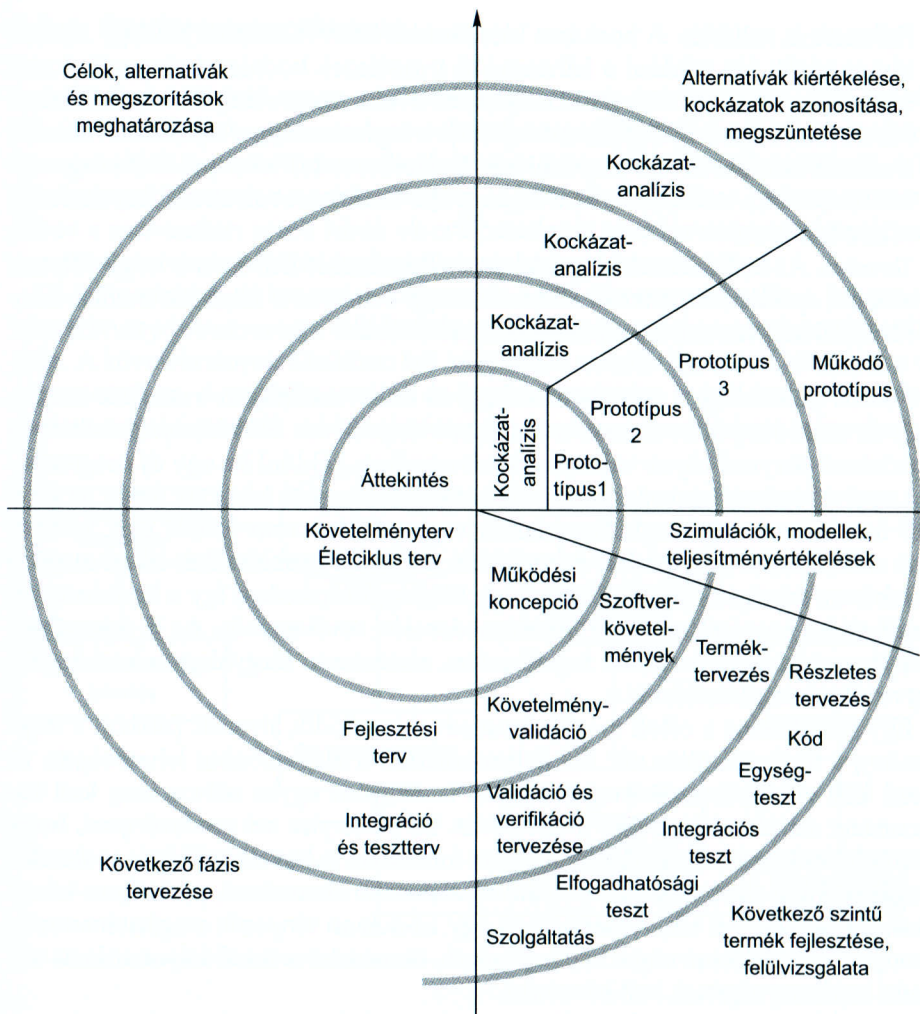
1. A megrendelőnek nem kell megvárnia, amíg a teljes rendszer elkészül, hogy hasznat húzhasanak belőle. Az első inkremens kielégítheti a legkritikusabb követelményeket, így a szoftver már menet közben használhatóvá válik.
2. A megrendelők használhatják a korábbi inkremenseket mint prototípusokat, és ezáltal tapasztalatokat, információkat szerezhetnek a későbbi rendszerinkremensek követelményeivel kapcsolatban.
3. Kisebb a kockázata annak, hogy a teljes projekt kudarcba fulladjon. Annak ellenére, hogy az inkremensekben hibák léphetnek fel, nagy a valószínűsége annak, hogy azok többségét leszállítják a megrendelőnek.
4. Ha a magas prioritású szolgáltatásokat szállítjuk le hamarabb, s a későbbi inkremenseket abba integráljuk, elkerülhetetlen, hogy a legfontosabb rendszer-szolgáltatásokat teszteljük a leggyakrabban. Ez azt jelenti, hogy sokkal kisebb annak az esélye, hogy a rendszer legfontosabb részeiben a megrendelőnél meghibásodás forduljon elő.

Mindezek ellenére az inkrementális fejlesztésnek is megvannak a saját problémái. Az inkremensek legyenek relatíve kisméretűek (nem több, mint 20 000 programsor), és minden egyes inkrementációs lépésnek szolgáltatnia kell valamilyen rendszerfunkciót. Ezáltal nehézkessé válhat a megrendelő követelményeit megfelelő méretű inkrementációs lépésekre felosztani. Sőt a legtöbb rendszer egy sor alapvető tulajdonságot követel meg, melyeket a rendszer különböző részei használnak. Azok követelményei addig részletesen nem definiálhatók, míg az inkremens implementálandó, azaz nehéz azonosítani az inkremensek közös tulajdonságait.

Az inkrementális megközelítés újabb keletű változata az *extrém programozás* (Beck, 2000). Ez a nagyon kis funkcionalitással rendelkező inkremensek fejlesztésén és leszállításán alapul, továbbá a megrendelő részt vesz a folyamatban, állandóan fejlesztik a kódot, a programozás pedig csoportos (17. fejezet).

4.2.2. Spirális fejlesztés

A szoftverfolyamat spirális modellje (4.5. ábra), melyet eredetileg Boehm (Boehm, 1988) ajánlott, manapság elég széles körben ismert. A szoftverfolyamatot nem tevékenységek és a közöttük található esetleges visszalépések sorozataként tekinti, hanem inkább spirálként reprezentálja. A spirál minden egyes körben a szoftverfolyamat egy-egy fázisát reprezentálja. Így a legbelső kör a megvalósíthatósággal foglalkozik, a következő a rendszer követelményeinek meghatározásával, a következő kör a rendszer tervezésével stb.



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

A spirál minden egyes ciklusát négy szektorra oszthatjuk fel:

1. *Célok kijelölése.* Az adott projektfázis által kitűzött célok meghatározása. Azonosítani kell a folyamat megszorításait, a terméket, fel kell vázolni a kapcsolódó menedzselési tervet. Fel kell ismerni a projekt kockázati tényezőit, és azoktól függően alternatív stratégiákat kell tervezni, amennyiben lehetséges.
2. *Kockázat becslése és csökkentése.* Minden egyes felismert projektkockázati tényező esetén részletes elemzésre kerüljön sor. Lépéseket kell tenni a kockázat csökkentése érdekében. Ha például fennáll annak a kockázata, hogy a követelmények nem megfelelők, akkor egy prototípusrendszert lehet fejleszteni.

3. *Fejlesztés és validálás.* A kockázat kiértékelése után választani kell egy fejlesztési modellt. Ha például a felhasználói interfészek kockázati tényezői dominánsak, a megfelelő fejlesztési modell az evolúciós prototípus-készítés lehet. Ha a biztonságosság kockázati tényezője a legfontosabb, a fejlesztést formális transzformációkra lehet alapozni stb. A vízesésmodell lehet például a legmegfelelőbb abban az esetben, ha a legnagyobb kockázatot az alrendszeres integrálása hordozza.
4. *Tervezés.* Át kell tekinteni a projektet és döntéseket kell hozni, hogy folytassuk-e a spirált a következő ciklussal, vagy sem. Ha a folytatás mellett döntünk, fel kell vázolni a projekt következő fázisát.

Fontos különbség a spirális és az egyéb szoftverfolyamat-modellek között, hogy az előbbi explicit számol a kockázati tényezővel. Hétköznapi értelemben a kockázati tényező olyan valami, ami elromolhat. Például ha egy új programozási nyelvet szándékozunk használni, kockázati tényező lehet az, hogy az elérhető fordítók esetleg megbízhatatlanok vagy például nem tudják elég hatékonyan előállítani a tárgykódot. A kockázati tényezők problémákat okozhatnak a projektben, mint például határidő- vagy költségátúllépéseket, így a kockázat minimalizálása nagyon fontos projektmenedzselési tevékenység. Az 5. fejezetben, amelyben a projektvezetéssel foglalkozom, részletesen tárgyalom a kockázatot, illetve a kockázatkezelést is.

Egy spirálciklus a célok meghatározásával kezdődik, ilyenek például a teljesítmény, a funkcionalitás stb. Fel kell sorolnunk a megvalósítás lehetőségeit, és hozzá kell vennünk azok megszorításait is. Minden egyes célhoz meg kell határozunk minden lehetséges alternatívát. Ez többnyire azt eredményezi, hogy azonosításra kerülnek a projekt kockázati forrásai is. A következő lépés ezeknek a kockázatoknak a kiértékelése: például részletesebb elemzéssel, prototípus-készítéssel, szimulációval stb. Amennyiben egy kockázati tényezőt meghatároztunk, bizonyos fejlesztéseket végre kell hajtánunk, és azt a következő folyamatfázis tervezési tevékenységének kell követnie.

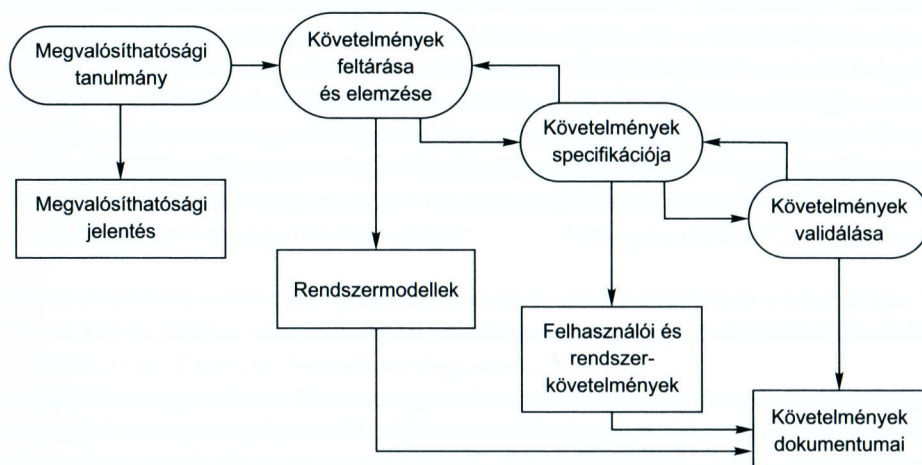
4.3. FOLYAMATTEVÉKENYSÉGEK

Négy alapvető folyamattevékenységet szoktunk elkülöníteni: specifikáció, fejlesztés, validáció és evolúció. Ezeket a különféle fejlesztési folyamatok különféleképpen szokták szervezni. A vízesésmodell esetében ezek egy szekvenciába szerveződnek, míg az evolúciós fejlesztésnél összefésülődnek. Az, hogy hogyan kezeljük ezeket a tevékenységeket, nagyban függ a szoftver típusától, az embevertől, illetve a szervezeti struktúrától is. Nincs jó vagy rossz módja ezek szervezésének, így ebben a fejezetben én is csak a tevékenységek szervezésének különböző lehetőségeit mutatom be.

4.3.1. Szoftverspecifikáció

A szoftverspecifikáció avagy a követelménytervezés az a folyamat, ahol megértjük és definiáljuk, hogy a rendszernek milyen szolgáltatásokat kell biztosítania, és azonosítjuk a rendszer üzemeltetésének és fejlesztésének megszorításait. A követelmények tervezése a szoftverfolyamat különösen kritikus szakasza, azaz az ebben a szakaszban vétett hibák elkerülhetetlenül problémákhoz vezetnek majd a rendszertervezés későbbi szakaszában és az implementációban.

A követelmények tervezési folyamatát a 4.6. ábra mutatja. Ez a folyamat a követelménydokumentum előállítását eredményezi, amely a rendszer specifikációja. A követelmények általában két különböző szinten kerülnek kifejtésre a dokumentumban. A végfelhasználók és ügyfelek számára le kell írni a követelményeket, míg a fejlesztők sokkal részletesebb rendszer-specifikációt igényelnek.



4.6. ábra. A követelménytervezés folyamata

A követelmények tervezésének négy nagy fázisát különböztethetjük meg:

1. *Megvalósíthatósági tanulmány.* Meg kell becsülni, hogy a felhasználók kívánságai kielégíthetők-e az adott szoftver- és hardvertechnológia mellett. A vizsgálatoknak el kell dönteniük, hogy a betervezett rendszer üzleti szempontból költséghatékony-e, és hogy a meglévő költségvetési megszorításokkal kivitelezhető-e. A megvalósíthatóság elemzésének relatíve olcsónak és gyorsnak kell lennie. Az eredménynek információkkal kell szolgálnia arról, folytassuk-e a munkát egy még részletesebb elemzéssel vagy sem.
2. *Követelmények feltárása és elemzése.* Ez a folyamat a rendszerkövetelmények meglévő rendszereken történő megfigyelésén, a potenciális felhasználókkal és beszerzőkkel folytatott megbeszéléseken, tevékenységelemzéseken stb. alapszik. Akár egy vagy több különböző rendszermodell, illetve prototípus elkészítését is magában foglalhatja. Ezek segítik ugyanis az elemzőket, hogy jobban megértsék a rendszert ahhoz, hogy azt pontosíthassák.

3. *Követelményspecifikáció.* A követelményspecifikáció az elemzési tevékenységek során összegyűjtött információk egységes dokumentummá alakítása, amely követelmények egy halmazát határozza meg. A dokumentumnak a követelmények két típusát kell tartalmaznia. A felhasználói követelmények a rendszerkövetelmények absztrakt leírása, amelyek a végfelhasználóknak, illetve a megrendelőknek szólnak. A konkrét rendszerkövetelmények pedig jobban részletezik az elkészítendő rendszer által nyújtandó funkciókat.
4. *Követelményvalidáció.* Ez a tevékenység ellenőrzi, hogy mennyire valószerűek, konzisztensek és teljesek a követelmények. A folyamat során fel kell tárni a követelmények dokumentumában található hibákat. A feltárt hibákat a dokumentum módosításával ki kell javítani.

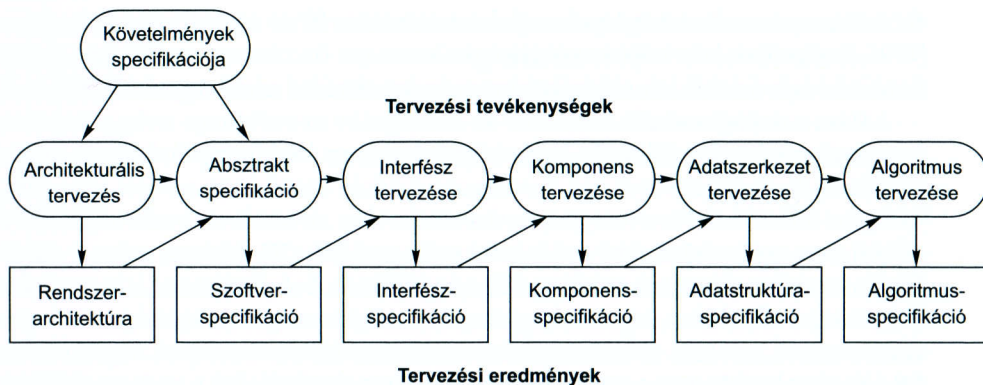
Természetesen a követelményfolyamat tevékenységeit nem kell szigorú sorrendben végrehajtani. A követelmények elemzése folytatható a meghatározásuk és specifikálásuk alatt, és a folyamat során bármikor új követelmények is napvilágra kerülhetnek. Ily módon az elemzés, a meghatározás és a specifikáció tevékenységei összefésülhetnek, egymást átfedhetnek a folyamatban. Az agilis módszerek esetében, mint például az extrém programozásban, a követelmények inkrementálisan vannak kifejlesztve a felhasználói prioritásoknak megfelelően. Az aktuálisan felderített követelmények azoktól a felhasználóktól származnak, akik éppen tagjai a fejlesztő csapatnak.

4.3.2. Szoftvertervezés és implementáció

A szoftverfejlesztés implementációs szakasza nem más, mint a rendszer-specifikáció futtatható rendszerre történő konvertálása. Ez mindig magában foglalja a szoftver tervezését és a programozást, de például evolúciós megközelítési mód esetén a specifikáció finomítását is tartalmazhatja.

A szoftver tervezése az implementálandó szoftver struktúrájának és az adatoknak mint a rendszer részeinek a meghatározása, valamint a rendszerkomponensek közötti interfészek és néha a használt algoritmusok leírása. A tervezők nem egyenes úton haladnak, hanem iteratív módon számos különféle verzió kifejlesztésén keresztül. A tervezési folyamat járulékos formalitást és kifejtést is magában foglal a terv fejlesztése közben, valamint folytonos visszalépéseket a korábbi tervek javítására.

A tervezési folyamat számos különféle absztrakciós szinten levő rendszermodell kifejlesztését is tartalmazhatja. Amint a tervet részekre osztjuk, napvilágra kerülnek a korábbi hibák és hiányosságok. Ezek a visszacsatolások biztosítják, hogy továbbfejleszthessük a korábbi tervezési modelleket. A 4.7. ábra egy ilyen folyamat modellje és azt mutatja meg, hogyan adhatók meg a tervleírások a tervezés különböző szakaszaiban. Ez a diagram azt sugallja, hogy a tervezési folyamat szakaszai szekvenciálisak. Igazság szerint a tervezési folyamat tevékenységei átfedik egymást. A szakaszok közötti visszacsatolás és a következetes átdolgozás minden tervezési folyamatban elkerülhetetlen.



4.7. ábra. A tervezési folyamat általános modellje

A következő szakasz specifikációja jelenti a tervezési tevékenység kimenetét. Ez a specifikáció lehet absztrakt, formális specifikáció, amelyet azért készítünk, hogy tisztázzuk a követelményeket, de akár annak a specifikációja is lehet, hogyan kell a rendszer részeit realizálni. Ahogy folytatódik a tervezési folyamat, ezek a specifikációk egyre részletesebbé válnak. A folyamat végeredménye pedig az implementálandó algoritmusok és adatszerkezetek precíz specifikációja.

A tervezési folyamat tevékenységei:

1. *Architektúrális tervezés.* A rendszert felépítő alrendszereket és a köztük található kapcsolatokat azonosítani és dokumentálni kell. Ezt a nagyon fontos témakört a 11., 12. és 13. fejezetben tárgyalom.
2. *Absztrakt specifikáció.* Minden egyes alrendszer esetében meg kell adni szolgáltatásaik absztrakt specifikációját és azokat a megkorlátozásokat, amelyek mellett azok működnek.
3. *Interfész tervezése.* Minden egyes alrendszer számára meg kell tervezni és dokumentálni kell annak egyéb alrendszerek felé mutatott interfészeit. Ennek az interfész-specifikációnak egyértelműnek kell lennie, azaz lehetővé kell tennie, hogy anélkül használhassunk egy alrendszert, hogy tudnánk, hogyan is működik. A 10. fejezetben tárgyalt formális specifikációs módszer is használható ebben a fázisban.
4. *Komponens tervezése.* A szolgáltatásokat el kell helyezni a különböző komponensekben, és meg kell tervezni a komponensek interfészeit.
5. *Adatszerkezet tervezése.* Meg kell határozni és részletesen meg kell tervezni a rendszer implementációjában használt adatszerkezeteket.
6. *Algoritmus tervezése.* Meg kell tervezni és pontosan meg kell határozni a szolgáltatások biztosításához szükséges algoritmusokat.

Ez a tervezési folyamat általános modellje és a valódi, gyakorlati folyamatok különböző módokon alkalmazhatók. Lehetséges adaptációk:

1. A tervezés utolsó két lépése – az adatstruktúrák és algoritmusok tervezése – elhalasztható az implementálásig.
2. A tervezés feltáró közelítésében a rendszerinterfész az adatstruktúrák specifikálása után tervezhető.
3. Az absztrakt specifikációs lépés kihagyható, ez általában a kritikus rendszereknél lényeges.

Ahol az agilis fejlesztési módszereket használják (17. fejezet), ott a tervezési folyamatok kimenetei nem különálló specifikációs dokumentumok lesznek, hanem inkább a kódban fognak megjelenni. Azaz a rendszer-architektúra megtervezése után a tervezés további szakaszai inkrementálisan kerülnek végrehajtásra. Minden egyes inkremens egy programkódot reprezentál.

A szoftvertervezés módszeresebb megközelítési módja a *strukturált módszer*, amely a grafikus modelleken alapul (8. fejezet), és számos esetben biztosít automatikus kódgenerálást. A strukturált módszereket az 1970-es években vezették be a funkcióorientált tervezés támogatására (Constantine és Yourdon, 1979; Gane és Sarson, 1979). Egy sor objektumorientált tervezési megközelítés is napvilágot látott (Robinson, 1992; Booch, 1994), amelyeket az 1990-es években egyesítettek, és létrejött a Unified Modelling Language (UML) és a hozzá kapcsolódó egységes tervezési folyamat (Rumbaugh és mások, 1991; Booch és mások, 1999; Rumbaugh és mások, 1999a, 1999b). A könyv írásának idején már folyamatban van az UML nagyobb revíziója (UML 2.0) is.

A strukturált módszer tervezési folyamatmodellt, a terv reprezentációjához jelölésrendszert, jelentésformátumokat, szabályokat és tervezési útmutatókat tartalmaz. Habár számos módszer létezik, nagyon sok bennük a közös. A strukturált módszerek az alábbi rendszermodellek közül támogathatnak néhányat vagy akár az összeset:

1. Egy objektummodellt, ez adja meg a rendszer objektumosztályait és azok függőségeit.
2. Egy szekvenciamodellt, ez a rendszer objektumainak kapcsolatát mutatja a futás alatt.
3. Egy állapotátmenet-modellt, ami a rendszer állapotainak és az átmenetek triggereinek modellje.
4. Egy strukturált modellt, ez a rendszer komponenseit és a köztük lévő kölcsönhatásokat dokumentálja.
5. Egy adatfolyammodellt, mellyel a rendszert a megfelelő helyen végrehajtott adattranszformációk segítségével modellezzük. Ez alaphelyzetben nem objektumorientált módszer, de gyakran alkalmazzák a valós idejű és az üzleti rendszerek tervezésében.

A gyakorlatban a módszerek nem túl formális tanácsokat adnak, így a különböző tervezők különféle tervek fognak előállítani. Ezek a „módszerek” valójában szabványos jelölések és gyakorlati jó tanácsok halmazai. A tervezők kreativitására természetesen még mindig szükség van a rendszer részekre bontásában

és abban, hogy a tervek kellő mértékben feleljenek meg a rendszer-specifikációnak. A tervezőkre vonatkozó tapasztalati vizsgálatok (Bansler és Bødker, 1993) azt mutatják, hogy ritkán követik vakon a módszereket, inkább a helyi körülményektől függően aprólékosan válogatnak az útmutató előírásaiból.

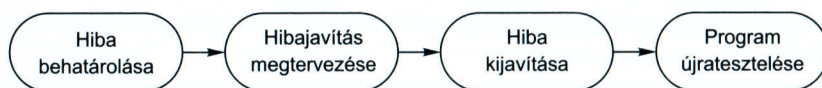
Egy program fejlesztése a rendszer implementációja érdekében természetesen a tervezési folyamatot követi. Noha a programok bizonyos fajtái, mint például a biztonságkritikus rendszerek, már részleteiben megtervezettek, mielőtt még bármilyen implementáció elkezdődne, elfogadott a tervezés és programozás későbbi szakaszait összefésülni. A CASE-eszközök gyakran használhatók arra, hogy a tervből segítségükkel generáljuk a program szerkezetét, vázát. Ez legtöbbször az interfészek implementációjának definiálására szolgáló kódokat tartalmaz, és a legtöbb esetben a fejlesztőnek csak a programkomponensek műveleteinek kódjait kell kifejtenie.

A programozás személyekre szabott tevékenység, nincs általános szabály, amit követni lehet. Bizonyos programozók a fejlesztést azon komponensekkel kezdik, melyeket a legjobban megértettek, majd csak azután fejlesztik a kevésbé érthető komponenseket. Mások pont fordítva teszik, a legvégére hagyják majd az ismerős komponenseket, ugyanis tudják, hogyan kell azokat fejleszteni. Néhány fejlesztő szereti a folyamat elején definiálni az adatokat, és aztán a program fejlesztésének vezérlésére használni azokat, mások addig hagyják az adatokat meghatározatlanul, ameddig csak lehet.

Rendes körülmények között a programozók végrehajtanak bizonyos teszteléseket az általuk kifejlesztett kódokon. Ezek a legtöbb esetben felfedik a program hibáit, amelyeket aztán ki kell javítani. Ez a folyamat a *nyomkövetés, belövés*. A hibák felderítése és a belövés külön folyamatok. A tesztelés meghatározza, hogy vannak-e hibák. A belövés pedig a hibák helyének meghatározásával és azok kijavításával foglalkozik.

A 4.8. ábra a lehetséges belövési folyamatot illusztrálja. A kódban található hibákat lokalizálni és a programot a követelményeknek megfelelően módosítani kell. Ezután meg kell ismételni a tesztelést, hogy megbizonyosodjunk arról, hogy a változtatást kifogástalanul vittük véghez. Így a belövési folyamat mind a szoftverfejlesztési, mind a szoftvertesztelési folyamatnak a része.

A belövőnek hipotéziseket kell generálnia a program megfigyelhető viselkedéseire, aztán tesztelnie kell ezeket a hipotéziseket annak a reményében, hogy megtalálja a kimeneti anomáliákat okozó hibákat. A hipotézisek tesztelése magában foglalhatja a programszöveg manuális nyomon követését is. Új teszteszközökre lehet szükség, hogy lokalizálhassuk a problémát. Az interaktív belövőeszközök nagy segítségünkre lehetnek a belövési folyamatban, ugyanis képesek arra, hogy megmutassák a program változóinak futás közbeni értékét és nyomon kövessék a végrehajtott utasításokat.

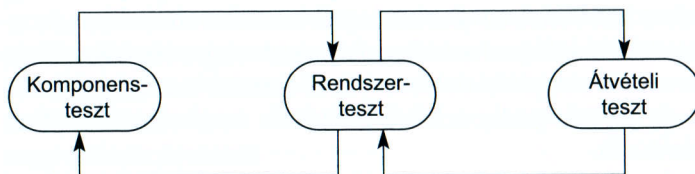


4.8. ábra. A belövési folyamat

4.3.3. Szoftvervalidáció

A szoftvervalidáció, vagy még általánosabban a verifikáció és a validáció (V & V) azzal hivatott foglalkozni, hogy megmutassa, a rendszer konform a saját specifikációjával, és hogy a rendszer megfelel a rendszert megvásárló ügyfél elvárásainak. Ez olyan ellenőrzési folyamatokat foglal magában, mint például szemléket, felülvizsgálatokat (22. fejezet) a szoftverfolyamat minden egyes szakaszában a felhasználói követelmények meghatározásától kezdve egészen a program kifejlesztéséig. A validációs költségek nagy része mindazonáltal az implementáció után jelentkezik, akkor, amikor a működő rendszert tesztljük (23. fejezet).

A kis programok kivételével a rendszerek nem tesztelhetők magukban mint monolitikus egységek. A 4.9. ábra háromlépéses tesztelési folyamatot mutat be, ahol teszteljük a rendszer komponenseit, majd az integrált rendszert, és végezetül a teljes rendszert a megrendelő adataival. Ideális esetben a komponensek hibáit már a folyamat korai szakaszaiban felderítjük, az integrációs szakaszban pedig az interfész problémáit deríthetjük ki. Akárhogy is, a programban felderített hibákat ki kell javítani, és ez azt vonhatja maga után, hogy a tesztelési folyamat egyéb szakaszait is meg kell ismételni. Ha a programkomponensekben található hibák az integrációs tesztelés alatt látnak napvilágot, akkor a folyamat iteratív, a későbbi szakaszokban nyert információk visszacsatolandók a folyamat korábbi szakaszaiba.



4.9. ábra. A tesztelési folyamat

A tesztelési folyamat szakaszai:

1. *Komponens (vagy egység) tesztelése.* Az egyedi komponenseket tesztelni kell, és biztosítani kell tökéletes működésüket. Minden egyes komponenst az egyéb rendszerkomponensektől függetlenül kell tesztelni.
2. *Rendszer tesztelése.* A komponensek integrált egysége alkotja a teljes rendszert. Ez a folyamat az alrendszerek és interfészeik közötti előre nem várt kölcsönhatásokból adódó hibák megtalálásával foglalkozik. Ezen túl érinti a validációt is, vagyis hogy a rendszer eleget tesz-e a rendszer funkcionális és nemfunkcionális követelményeinek és az eredendő rendszertulajdonságoknak. Nagy rendszerek esetén ez egy többfázisú folyamat is lehet, ahol a komponenseket alrendszerekbe építjük, melyeket külön-külön tesztelhetünk, mielőtt összeépítjük belőlük a teljes rendszert.
3. *Átvételi tesztelés.* Ez a tesztelési folyamat legutolsó fázisa a rendszer használatbavétele előtt. A rendszert ilyenkor inkább a megrendelő adataival kell teszt-

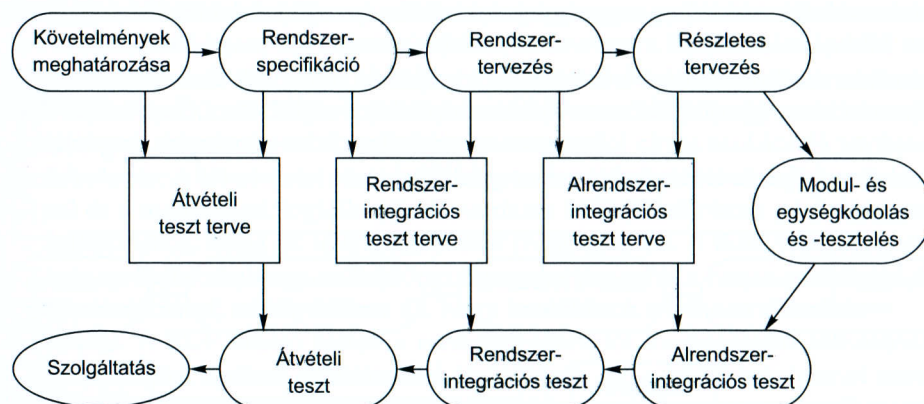
telni, mintsem szimulált tesztadatokkal. Az átvételi tesztelés olyan hibákat és hiányosságokat fedhet fel a rendszerkövetelményekben, melyekhez csak a rendszer valós adatokkal történő vizsgálata vezethet, ugyanis azok legtöbbször nagyban különböznek a tesztadatoktól. Az átvételi tesztelés olyan követelménybeli problémákat is felvethet, amelyeknél a rendszer tulajdonságai nem tesznek eleget a felhasználó kívánalmainak, vagy esetleg a rendszer teljesítménye elfogadhatatlan.

Normális esetben a komponensek fejlesztése és tesztelése összefonódik. A programozók elkészítik a saját tesztadataikat, és nekiállnak az általuk fejlesztett kód inkrementális tesztelésének. Ez gazdaságilag érzékeny módszer, ugyanis a programozó ismeri az általa fejlesztett komponenst a leginkább, így módon ő a legjobb személy arra, hogy a lehető legjobb tesztadatokat állítsa elő.

Ha inkrementális fejlesztési módot alkalmazunk, akkor minden inkeremenst annak kell tesztelnie, aki kifejlesztette. Ezek a tesztek az inkremensek követelményeire alapulnak. Az extrém programozás esetén a tesztek a követelményekkel együtt fejlesztik ki, még a teljes fejlesztési folyamat előtt. Ez abban segíti a tesztelőket, hogy jobban megértsék a követelményeket, illetve hogy ne legyen késés a tesztesetek kifejlesztésében.

A tesztelés későbbi szakaszai számos programozó előre tervezett közös munkáját jelentik. Tesztelők független csoportja dolgozhat előre formalizált teszttervek alapján, amelyek a rendszer-specifikáció és -tervezés alapján jöttek létre. A 4.10. ábra azt illusztrálja, hogyan kapcsolják össze a teszttervek a tesztelési és fejlesztési tevékenységeket.

Az átvételi tesztelést alfa-tesztelésnek is szokták nevezni. A megrendelésre készített rendszerek egy egyedülálló kliensnek készülnek. Az alfa-tesztelési folyamatot addig kell folytatni, amíg a rendszerfejlesztő és a kliens egyet nem ért abban, hogy a leszállított rendszer a rendszerkövetelmények egy elfogadható implementációja.



4.10. ábra. A szoftverfolyamat tesztelési fázisai

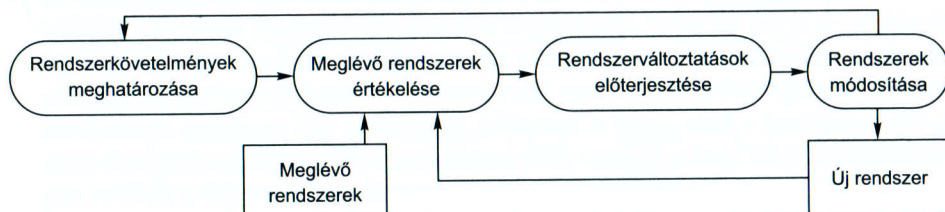
Amikor egy rendszer mint szoftvertermék piacra kerül, gyakran egy másik tesztelés is végbemegy, amelyet béta-tesztelésnek nevezünk. A béta-tesztelés magában foglalja a rendszer számos potenciális felhasználójához történő leszállítását, akikkel megegyezés történt a rendszer használatára, és ők jelentik a rendszerrel kapcsolatos problémáikat a rendszerfejlesztőknek. Ezáltal a rendszer valódi használatba kerül, és számos olyan hiba válik felfedezhetővé, amelyet a rendszer építői esetleg nem láthattak előre. Ezután a visszacsatolás után a rendszer módosítható és kiadható további béta-tesztelőknek, vagy akár általános használatra is.

4.3.4. Szoftverevolúció

A szoftverrendszerek flexibilitása az egyik legfontosabb ok, amiért egyre több és több szoftver egyesül nagy, összetett rendszerekben. Mihelyt megszületik a döntés valamilyen hardver gyártásáról, nagyon drágává válnak a hardverben történő változtatások. A szoftverek esetében azonban a rendszer fejlesztésének ideje alatt változtatások bármikor megtehetők. Ezek a változtatások akár nagyon költségesek is lehetnek, de még mindig jóval olcsóbbak, mint a hasonló mértékű hardver-változtatások.

Mindig éles demarkációs vonal húzódott a szoftverfejlesztés folyamata és a szoftverevolúció (szoftverkarbantartás) között. A szoftverfejlesztés kreatív tevékenység, ahol a kezdeti koncepcióból egy munkarendszeren keresztül a szoftverrendszer elkészül. A szoftver karbantartása viszont a rendszeren történő változtatások folyamata, a rendszer működésbe állítása után. Habár a „karbantartás” költségei akár a fejlesztés költségeinek többszörösét is elérhetik, a karbantartási folyamat sokkal kisebb kihívásnak tekinthető, mint egy eredeti szoftver kifejlesztése.

Ez az elkülönítés egyre lényegtelenebbé válik. Kevés szoftverrendszer tekinthető napjainkban teljesen újnak, így egyre több az alapja annak, hogy a fejlesztést és a karbantartást egy egységnek tekintsük, és nem két különálló folyamatnak. Sokkal valószerűbb a szoftvertervezést evolúciós folyamatként kezelni, ahol a szoftver élettartama alatt a követelményekkel és az ügyfél elvárásaival együtt folyamatosan változik. Ezt az evolúciós folyamatot a 4.11. ábra illusztrálja, ahol a szoftver életciklusa során folyamatosan változik, a követelmények megváltozásának és a felhasználói igényeknek megfelelően.



4.11. ábra. Rendszer-evolúció

4.4. A RATIONAL UNIFIED PROCESS

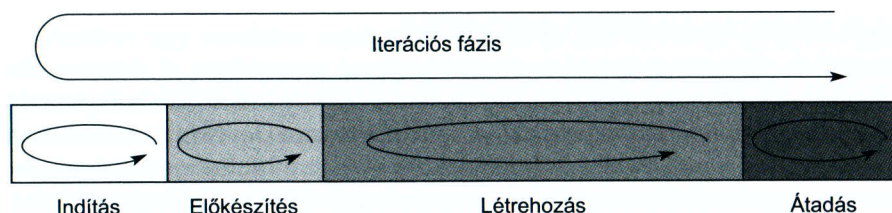
A Rational Unified Process (RUP) jó példája a modern folyamatmodelleknek, melyek az UML-ből és a hozzá kapcsolódó Unified Software Development Process-ből (Rumbaugh és mások, 1999b) származnak. Jó példája a hibrid modelleknek is, ugyanis mindegyik korábbi általános folyamatmodellből (4.1. alfejezet) tartalmaz elemeket, támogatja az iterációt (4.2. alfejezet), és jól illusztrálja a specifikáció és a tervezés tevékenységeit (4.3. alfejezet). A RUP felismeri, hogy a konvencionális folyamatmodellek a folyamatoknak csak egy egyszerű nézetét adják. Ezért a RUP alapesetben három perspektívával jellemezhető:

1. dinamikus perspektívával, amely a modell fázisait mutatja;
2. statikus perspektívával, amely a végrehajtandó folyamattevékenységeket mutatja;
3. gyakorlati perspektívával, amely jól hasznosítható gyakorlatokat javasol a folyamat alatt.

A RUP leírásai általában megpróbálják a statikus és a dinamikus perspektívákat egy egyszerű diagrammal ábrázolni (Krutchen, 2000). Érzésem szerint ez csak megnehezíti a folyamat megértését, éppen ezért minden egyes perspektívát különállóan kezelek.

A RUP is fázisokból áll, egészen pontosan négy diszkrét fázist tudunk elkülöníteni a folyamat során. A vízesésmodellől eltérően, ahol a fázisok folyamattevékenységeknek voltak megfelelően, a RUP fázisai sokkal közelebb állnak az üzleti vonatkozásokhoz, mint a technikaiakhoz. A 4.12. ábra mutatja a RUP fázisait:

1. *Indítás.* Az indítási fázis célja a rendszer üzleti vonatkozásainak tisztázása. Azonosítani kell a külső entitásokat (embereket és rendszereket), melyek kapcsolatban lesznek majd a rendszerrel, és meg kell adni ezeket a kapcsolatokat. Ezek után ez alapján el lehet dönteni, hoz-e elegendő nyereséget a rendszer. Amennyiben nem, úgy a projekt ezután a fázis után elvethető.
2. *Előkészítés.* Ennek a célja a probléma megértése, az architektúrális keretrendszer biztosítása, a projektterv kifejlesztése és a kulcs kockázati tényezők azonosítása. Ennek a fázisnak a befejezése után rendelkezni fogunk a rendszerkövetelményeink modelljével (UML használati eset alakban), egy architektúrális leírással, valamint a szoftver fejlesztési tervével.
3. *Létrehozás.* A létrehozási fázis legfőképpen a rendszertervvel, a programozással és a teszteléssel foglalkozik. A rendszer különböző részei párhuzamosan fejleszthetők, majd ez alatt a fázis alatt integrálhatók. A fázis teljesítése után már rendelkezünk egy működő szoftverrendszerrel és a hozzá csatlakozó dokumentációval, amely készen áll, hogy leszállítsuk a felhasználónak.
4. *Átadás.* A RUP utolsó fázisa a rendszer fejlesztői környezetből felhasználói környezetbe történő átültetésével foglalkozik. Ez a legtöbb folyamat esetén hiányzik, pedig ez rendkívül fontos és nem problémamentes tevékenység. A fázis eredménye egy dokumentált szoftverrendszer, mely megfelelően működik az üzemeltetési környezetben.



4.12. ábra. A Rational Unified Process fázisai

A RUP az iterációt kétféleképpen támogatja, mint ahogy az a 4.12. ábrán is látható. Minden egyes fázis iteratív módon is végrehajtható, ahol az eredmények inkrementálisan állnak majd elő. És ezenfelül még az egész fázishalmaz is inkrementálható, ezt mutatja a nyíl az ábrán, mely az Átadás után az Indítás felé mutat ciklikusan.

A RUP statikus nézete a fejlesztési folyamat tevékenységeire fókuszál. Ezeket *munkafolyamatnak* nevezi a RUP. Hat mag munkafolyamatot különböztethetünk meg a folyamatban, és három mag támogató munkafolyamatot. A RUP az UML-lel (egy objektumorientált modellezőnyelvvel) karöltve került kifejlesztésre, így a munkafolyamatok leírása UML-modellekkel történik. A mag tervezési és támogató munkafolyamatokat a 4.13. ábrán láthatjuk.

Munkafolyamat	Leírás
Üzleti modellezés	Az üzleti folyamatokat használati esetekkel modellezi.
Követelmények	A rendszerrel kapcsolatban álló aktorok azonosítása megtörténik és a rendszerkövetelményeket használati esetekkel modellezzik.
Elemzés és tervezés	Létrejön a dokumentált tervmodell az architektúráis modellek, a komponensmodellek, az objektummodellek és a szekvenciamodellek felhasználásával.
Implementálás	A rendszer komponensei implementálásra kerülnek, kialakul az alrendszerszerkezet. Automatikus kódgenerátorok segítségével a tervmodellből kód állítható elő.
Tesztelés	A tesztelés iteratív folyamat, amely szorosan kapcsolódik az implementáláshoz. Az implementálást a rendszerteszt teszi teljessé.
Bevezetés	Megszületik a termék, ezt átadják a felhasználóknak és telepítik a munkahelyeiken.
Konfiguráció- és változtatáskezelés	Ez a támogató munkafolyamat kezeli a rendszer változtatásait (29. fejezet).
Projektmenedzsment	Ez a támogató munkafolyamat kezeli a rendszer fejlesztését (5. fejezet).
Környezet	Ez a munkafolyamat a szoftverfejlesztő csapat számára elérhető szoftvereszközökre vonatkozik.

4.13. ábra. A Rational Unified Process statikus munkafolyamatai

A statikus és dinamikus nézetekkel azért is érdemes tisztában lenni, mivel a fejlesztési folyamat fázisai nincsenek külön munkafolyamatokhoz kötve. Elméletileg minden RUP-munkafolyamat bármelyik fázisában lehet aktív folyamat. Persze az üzleti modellezés és a követelmények munkafolyamatokra szokták a legtöbb időt szentelni a fázisok korai szakaszában, és a kései szakaszokban pedig inkább a tesztelés és a bevezetés kerül előtérbe.

A RUP gyakorlati perspektívája a rendszer fejlesztéséhez elengedhetetlen szoftvertervezési gyakorlatokkal foglalkozik. Hat alapvető legjobb gyakorlat ajánlott:

1. *Fejlesszük a szoftvert iteratívan.* Tervezzük meg a rendszer inkrementseit a megrendelő prioritásai alapján, és hozzuk előre a rendszer magas prioritású tulajdonságait a fejlesztési folyamat elején.
2. *Kezeljük a követelményeket.* Explicit módon dokumentáljuk a megrendelő követelményeit, és tartjuk mindig karban annak változásait. Elemezzük a változások rendszerre gyakorolt hatásait, mielőtt elfogadnánk azokat.
3. *Használunk komponensalapú architektúrákat.* A rendszert szervezzük komponensekbe, mint ahogy azt már korábban is láthattuk.
4. *Modellezzük a szoftvert vizuálisan.* Használunk grafikus UML-modelleket a szoftver statikus és dinamikus nézeteihez.
5. *Ellenőrizzük a szoftverminőséget.* Bizonyosodjunk meg róla, hogy a szoftver megfelel a szervezeti minőségi előírásoknak.
6. *Ellenőrizzük a szoftverváltoztatásokat.* Menedzseljük a szoftverváltoztatásokat változtatáskezelési eszközökkel és konfigurációkezelési eljárásokkal (29. fejezet).

A RUP nem alkalmas minden különböző típusú szoftver kifejlesztésére, de az általános folyamatok új generációjának tekinthető. A legfontosabb innováció a fázisok és munkafolyamatok szeparálása, valamint az, hogy a felhasználói környezetben történő üzembe állítás a folyamat része. A fázisok dinamikusak, és konkrét célokkal rendelkeznek. A munkafolyamatok statikusak és olyan technikai tevékenységeket tartalmaznak, melyek nem rendelhetők hozzá egyes fázisokhoz, hanem a teljes fejlesztésen keresztül használhatók fel a szakaszok céljainak elérésére.

4.5. SZÁMÍTÓGÉPPEL TÁMOGATOTT SZOFTVERTERVEZÉS

Aszámítógéppel támogatott szoftvertervezés (Computer-Aided Software Engineering, CASE) névvel – jelentésén túl – azokat a szoftvereket is szoktuk illetni, melyeket a szoftverfolyamat különböző tevékenységeinek támogatására használunk, például követelményelemzés, tervezés, programfejlesztés és tesztelés. A CASE-eszközök ezért tervezői szerkesztőprogramokat, adatszótárakat, fordítókat, be-lövő- és nyomkövető eszközöket, rendszerépítési eszközöket stb. tartalmaznak.

A CASE-technológia bizonyos folyamattevékenységek automatizációjával támogatja a szoftverfolyamatot, és információkat szolgáltat az éppen fejlesztés alatt álló szoftverről. Néhány példa arra, hogy milyen tevékenységeket lehet automatizálni CASE-eszközök segítségével:

1. a grafikus rendszermodellek fejlesztését a követelmények specifikációjának vagy a szoftver tervezésének részeként;
2. a terv megértését adatszótár segítségével, mely a tervben található egyedekekről és a kapcsolataikról tartalmaz információt;
3. a felhasználói interfészek generálását egy grafikus interfészleírásból, mely interaktív módon a felhasználóval együtt készült el;
4. a program belövését a futó programról adott információk biztosításával;
5. a programok automatikus fordítását a régi verziójú programnyelvekből, mint például COBOL, újabb keletű verziókba.

A CASE-technológia napjainkban a szoftverfolyamat legtöbb rutintevékenységre elérhető. Ez oda vezetett, hogy a szoftverek minőségében és a termelésben bizonyos feljavulás volt érezhető, de kisebb mértékben, mint azt a CASE-eszközök korai pártolói jósolták. Azt állították, hogy nagyságrendi javulás lesz érezhető az integrált CASE-eszközök használatával. Valójában az aktuálisan elért javulás csak 40 százalékos volt (Huff, 1992). Habár ez jelentős, nem vált valóra az a várakozás, amit a CASE-eszközök 80-as, 90-es években történt bevezetésekor jósoltak, hogy lényegesen csökkenteni fogják a szoftverfolyamat költségeit.

A CASE-eszközök használatával elérhető javulást két tényező határolja be:

1. A szoftvertervezés lényegében tervezői tevékenység, mely a kreatív gondolkodáson alapszik. A létező CASE-eszközök automatizálják a rutintevékenységeket, és megpróbálják hasznosítani a mesterséges intelligencia bizonyos technológiáit, hogy még jobban támogassák a tervezést, de azt nem túl nagy sikerrel teszik.
2. A legtöbb szervezetben a szoftvertervezés csoportos tevékenység, és a szoftvertervezők rengeteg időt töltenek azzal, hogy a csapat más tagjaival eszmét cseréljenek. A CASE-technológia ehhez nem nyújt túl nagy segítséget.

A CASE-technológia napjainkra megérett, és a CASE-eszközök és -eszköz-készletek már számos szállítótól elérhetők. Mindamellet én most nem ezekre a specifikus eszközökre összpontosítok, itt csak röviden áttekintem, és kiegészítem néhány specifikus jellemzővel a könyv kapcsolódó fejezeteiben található ismereteket. A könyv weblapjára elhelyeztem számos olyan hiperhivatkozást, melyek részletesebb anyagokra mutatnak a CASE-integrációval, illetve a CASE-eszközök gyártóival kapcsolatban.

4.5.1. CASE-osztályozás

A CASE-osztályozás abban segít, hogy megértsük a különböző típusú CASE-eszközöket és azoknak a szoftverfolyamat tevékenységeinek támogatásában betöltött szerepét. A CASE-eszközöket többféle módon osztályozhatjuk, mindegyik más-más nézőpontból szemléli ezeket az eszközöket. Ebben az alfejezetben három ilyen perspektívából tárgyalom a CASE-eszközöket, nevezetesen:

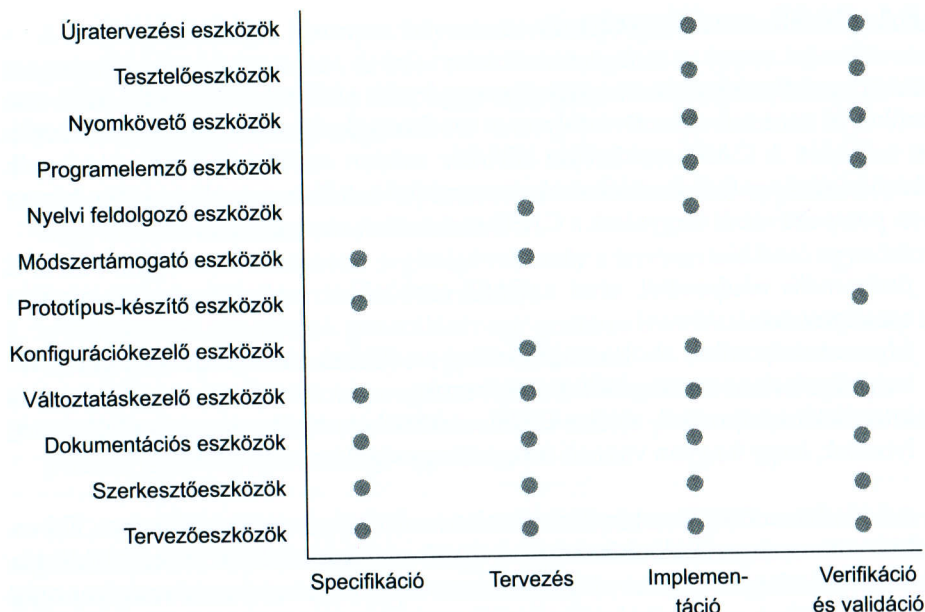
1. *funkcionális nézőpontból*, ahol a CASE-eszközöket sajátos funkcióik alapján osztályozzuk;
2. *folyamatnézőpontból*, ahol az eszközöket az általuk támogatott folyamattevékenységeinek szemszögéből osztályozzuk;
3. *integrációs nézőpontból*, ahol a CASE-eszközöket abból a szempontból osztályozzuk, hogy hogyan vannak integrált egységekbe szervezve.

A 4.14. ábra a CASE-eszközöket a funkcionalitásuk alapján osztályozza. Ebben a táblázatban számos különböző típusú CASE-eszközt sorolunk fel, és példákat is adunk minden egyes eszközre. Ez a lista nem teljes, nem tartalmazza az olyan speciális eszközöket, mint például az újrafelhasználhatóságot támogató eszközöket.

Eszköztípus	Példák
Tervezői eszközök	PERT-eszközök, becslési eszközök, táblázatkezelők
Szerkesztőeszközök	Szövegszerkesztők, diagramszerkesztők, szövegfeldolgozók
Változtatáskezelő eszközök	Követelmény követhetőségi eszközök, változtatásvezérlő rendszerek
Konfigurációkezelő eszközök	Verziókezelő rendszerek, rendszerépítő eszközök
Prototípus-készítő eszközök	Nagyon magas szintű programnyelvek, felhasználói interfész-generátorok
Módszertámogató eszközök	Tervszerkesztők, adatszótárak, kódgenerátorok
Nyelvi feldolgozó eszközök	Fordítók, értelmezők
Programelemző eszközök	Keresztreferencia-generátorok, statikus elemzők, dinamikus elemzők
Tesztelőeszközök	Tesztadat-generátorok, állomány-összehasonlítók
Nyomkövető eszközök	Interaktív nyomkövető, belsőrendszerek
Dokumentációs eszközök	Arculattervező programok, képszerkesztők
Újratervezési eszközök	Kereszthivatkozási rendszerek, program-újrastrukturáló rendszerek

4.14. ábra. A CASE-eszközök funkcionális osztályozása

A 4.15. ábra a CASE-eszközök egy másik lehetséges osztályozását mutatja be. Ez az ábra a különböző CASE-eszközök által támogatott folyamatfázisokat szemlélteti. A tervezéshez, előrejelzéshez, szövegek szerkesztéséhez, dokumentumkészítéshez és konfigurációkezeléshez használható eszközök az egész szoftverfolyamat során használhatók.



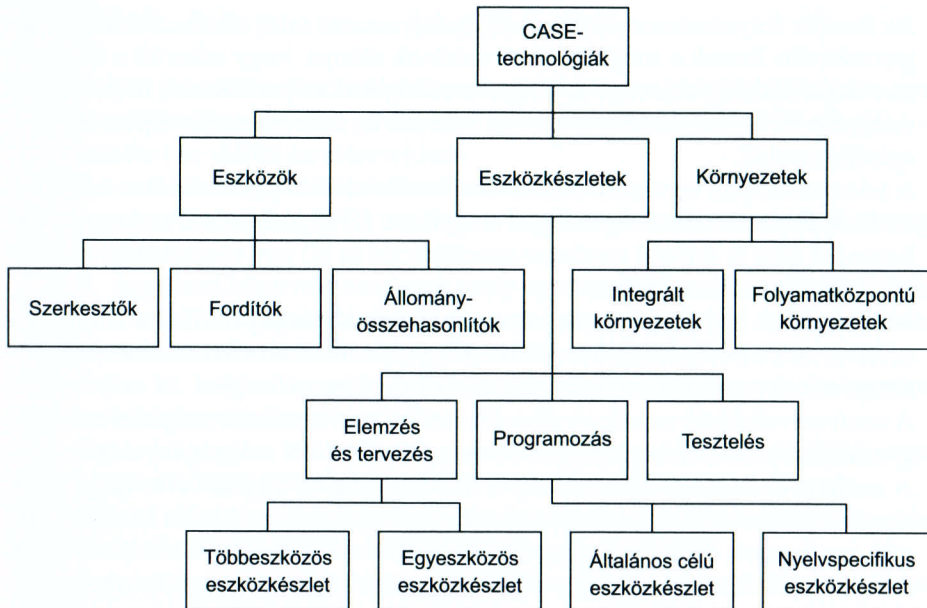
4.15. ábra. CASE-eszközök tevékenységalapú osztályozása

A CASE-technológiák által nyújtott támogatás nagysága szintén egy lehetséges osztályozási szempont. Fuggetta (Fuggetta, 1993) ajánlotta, hogy a CASE-rendszereket az alábbi három kategóriába érdemes csoportosítani:

1. Az *eszközök* az egyéni folyamatlépéseket támogatják, például a terv konzisztencia-ellenőrzését, a program fordítását, a teszteredmények összehasonlítását stb. Az eszközök lehetnek általános célúak, különálló eszközök (például szövegszerkesztők), illetve eszközkészletekbe szervezhetők.
2. Az *eszközkészletek* a munkafolyamat olyan fázisait támogatják, mint például a specifikáció, a tervezés stb. Ezek az eszközkészletek alaphelyzetben eszközök többé vagy kevésbé integrált halmazából állnak.
3. A *környezetek* a szoftverfolyamat minden, vagy az alapvetően fontos részeit támogatják. Rendes körülmények között különböző eszközkészleteket tartalmaznak, amelyeket valamilyen módon integráltak.

A 4.16. ábra ezt az osztályozást mutatja be. A CASE-eszköz példák csak illusztrációk, számos eszközkészlet és eszköz kimaradt a diagramból.

Általános célú eszközöket általában a szoftvertervező döntése alapján vesznek használatba. Az eszközkészletek azonban támogathatnak több olyan módszert is, amelyek például a szoftverfejlesztéshez alkalmazható folyamatmodelleket, szabályhalmazokat, útmutatókat is tartalmaznak. Én a magam részéről a környezeteket úgy osztályozom, hogy vannak integrált környezetek és vannak folyamatközpontú környezetek. Az integrált környezetek az adatokhoz, a vezérléshez és a beállításokhoz adnak infrastruktúrát, míg a folyamatközpontú környezetek sokkal



4.16. ábra. Eszközök, eszközkészletek, környezetek

általánosabbak. Olyan szoftverfolyamatbeli ismeretekkel és folyamatmotorral rendelkeznek, amelyek az adott folyamatmodell felhasználva tanácsokat is adnak a tervezőknek, hogy mely eszközt vagy eszközkészletet használják és mikor.

A gyakorlatban e két osztály között eléggé elmosódott a határ. Az eszközök akár külön termékként is eladhatók, de különböző folyamatok támogatása érdekében egymásba is ágyazhatók. Például a legtöbb szövegszerkesztő napjainkban tartalmaz beépített diagramszerkesztőket. A tervezéshez használatos CASE-eszközkészletek egyre nagyobb mértékben támogatják a programozást és a tesztelést, így egyre jobban hasonlítanak a környezetekre, mintsem speciális eszközkészletekre. Ezekből adódóan nem mindig könnyű egy termékről eldönteni, hogy melyik osztályba tartozik. Mindamellet ennek eldöntése első lépésként nagyon jó módja annak, hogy megismerjük az általuk nyújtott folyamatátvitel mértékét.

Kulcsfogalmak

- A szoftverfolyamatok azok a tevékenységek, melyek egy szoftverrendszer előállításában részt vesznek. A szoftverfolyamat-modell a folyamat absztrakt reprezentációja.
- Minden egyes szoftverfolyamat tartalmazza a szoftver specifikációját, a szoftver tervezését és implementációját, a szoftvervalidációt és a szoftverevolúciót.
- Az általános folyamatmodellek a szoftverfolyamat szerveződését írják le. Ilyen általános szoftverfolyamat-modellek a vízesésmodell, az evolúciós fejlesztés, a formális rendszerfejlesztés és az újrafelhasználás-orientált fejlesztés.

- Az iteratív folyamatmodellek a szoftverfolyamatot mint ciklikus tevékenységet tekintik. Ennek a megközelítési módnak előnye, hogy elkerüli a túlságosan korai állásfoglalásokat és elkötelezettségeket a specifikáció, illetve a tervezés terén. Ilyen iteratív modellek például az inkrementális fejlesztés és a spirális modell.
- A követelmények tervezése a szoftverspecifikáció kifejlesztésének a folyamata. Ez a folyamat két dolgot foglal magában: (1) ki kell fejleszteni egy, a felhasználó által is érthető rendszer-specifikációt és (2) egy részletesebb specifikációt a fejlesztők számára.
- A tervezési és implementációs folyamat a követelményspecifikáció futtatható szoftverrendszerre transzformálásával foglalkozik. A rendszer tervezési módszerei ennek a transzformációnak a részeként használhatók.
- A szoftvervalidáció annak az ellenőrzése, hogy a rendszer megfelel-e a saját specifikációjának, és hogy eleget tesz-e a felhasználók valós igényeinek.
- A szoftverevolúció a létező szoftver módosításaival foglalkozik, hogy azok megfeleljenek az új követelményeknek. Ez napjainkban a kis- és közepes méretű rendszerek szokványos szoftverfejlesztési megközelítési módjává válik.
- A Rational Unified Process egy modern általános folyamatmodell, amely fázisokba van szervezve (indítás, előkészítés, létrehozás, átadás), de a tevékenységek (követelmények, elemzés és tervezés stb.) azoktól el vannak különítve.
- A CASE-technológiák automatizált támogatást nyújtanak a szoftverfolyamathoz. A CASE-eszközök a különálló folyamattevékenységeket, a CASE-eszköz-készletek a tevékenységek egy halmazát támogatják, a CASE-környezetek célja pedig minden vagy a legtöbb folyamatbeli tevékenység támogatása.

További irodalom

Extreme Programming Explained: Embrace Change. Egy evangelista könyv, amely az extrém programozási folyamatról szól. A szerző az extrém programozás bevezetője, így a rajongása nagyon áthatja a könyvet. (Beck, K., 2000, Addison-Wesley).

The Rational Unified Process – An Introduction. Ez a legolvasmányosabb könyv a RUP-ról a jelen könyv írása idején. Krutchen jól írja le a folyamatot, de szerintem egy kicsit keveset foglalkozik a folyamat gyakorlati használatának nehézségeivel. (Krutchen, P., 2000, Addison-Wesley).

Managing Software Quality and Business Risk. Ez elsődlegesen a szoftvermenedzselésről szóló könyv, de van egy kiváló fejezet (4. fejezet) a folyamatmodellekről. (Ould, M., 1999, John Wiley & Sons.)

„A classification of CASE technology”. Ez a cikk egy osztályozási sémát mutat be, bár Fuggetta részletesebben illusztrálja, hogyan lehet számos piaci terméket ebbe a sémába elhelyezni, mint azt mi tettük. (Fuggetta, A., *IEEE Computer*, 26 (12), December 1993.)

Feladatok

- 4.1. Adjon javaslatot rövid indoklással együtt a legmegfelelőbb általános rendszerfolyamat-modellre, amelyet a fejlesztés menedzselésének alapjául választhat az alábbi rendszerekhez:
 - autók feltörésvédelmi ellenőrző rendszere;
 - virtuális valóság (VR) rendszer a rendszerek karbantartásának támogatására;
 - egyetemi könyvelő rendszer, mely egy meglévő rendszert fog felváltani;
 - vasúti utasok interaktív rendszere a vonatok indulási és érkezési idejének lekérdezésére, mely az állomásokon lesz telepítve.
- 4.2. Fejtse ki, hogy az evolúciós fejlesztéssel fejlesztett programokat gyakran miért nehezebb karbantartani.
- 4.3. Fejtse ki, hogyan használható a szoftverfolyamat vízésésmodellje és a prototípus-készítési modell egyszerre a spirális folyamatmodellben.
- 4.4. Mik az előnyei annak, ha a RUP-hoz hasonlóan statikus és dinamikus nézetet is adunk egy szoftverfolyamathoz?
- 4.5. Indokolja meg, hogy miért fontos a felhasználó számára fejlesztett követelmények és a fejlesztők számára fejlesztett rendszerkövetelmények között különbséget tenni a követelmények tervezésének folyamatában.
- 4.6. Adja meg a szoftver tervezési folyamatának fontosabb tevékenységeit és azok kimeneteit. Egyed-kapcsolat (ER) diagramot használva mutassa meg a lehetséges kapcsolatokat a tevékenységek és kimeneteik között.
- 4.7. Mi egy tervezési módszer öt komponense? Vegye bármelyik ön által ismert módszert, és adja meg komponenseit. Becsülje fel az ön által választott módszer teljességét.
- 4.8. Tervezzon egy folyamatmodellt rendszertesztek futtatásához és azok eredményeinek feljegyzéséhez.
- 4.9. Fejtse ki, hogy a valós világ körülményei között használt szoftverrendszert miért kell megváltoztatni, vagy ellenkező esetben fokozatosan használhatatlanná válhat.
- 4.10. Tegyen javaslatot arra, hogyan lehetnek a vezetőknek a CASE-eszközök beszerzésére irányuló felelősségükben segítségükre a CASE-technológiák osztályozási sémái.
- 4.11. Tekintse át a helyi fejlesztési környezetében található eszközök lehetőségeit, és osztályozza ezen eszközöket a javasolt paraméterek alapján (funkció, tevékenység, támogatás mértéke).
- 4.12. Történetileg a technológiák bevezetése mély változásokat okozott a munkaerőpiacon, és ideiglenesen elmozdított embereket a munkájukból. Elemezze, vajon a CASE-technológia bevezetése is hasonló eredményekkel jár-e a szoftvertervezők számára. Amennyiben úgy gondolja, hogy nem, fejtse ki, miért nem. Ha úgy gondolja, igen, és csökkenteni fogja a munkalehetőségeket, etikus-e a tervezőknek passzívan vagy aktívan ellenállni a technológia bevezetésének?