

FEJLESZTÉS

Amikor a szoftvertervezés mint tudományág először megjelent, a fejlesztési folyamat a legtöbb rendszer esetében nem csupán egy folyamat volt, amely a terv specifikációján alapuló program megírását jelentette. Ehhez imperatív programozási nyelveket használtak, mint például a C, a Fortran vagy az Ada. A szoftvertervezési kézikönyvekben a szoftverfejlesztéssel foglalkozó részek pedig leginkább a jó programozási gyakorlatokra összpontosítottak.

Mára azonban a szoftverfejlesztésnek számos különböző útja létezik. Megtalálhatók közöttük a hagyományos programozási nyelvek, mint például a C++ vagy a Java, a szkriptelés, az adatbázis-programozás, a CASE-eszközök által generált programok és az újrafelhasználáson alapuló szoftvertervezés. Továbbá azt a tényt is felismerték, hogy nincs valódi különbség a fejlesztés és a karbantartás között; és ettől kezdve a fejlesztésre kezdetünk úgy gondolni, mint a programevolúció folyamatának első lépésére. Ezekre a fejlesztésekre reflektálva építettem be ezt az új részt a könyvbe, a fejlesztési technikákra összpontosítva. Öt fejezet található ebben a részben:

1. A 17. fejezet a gyors szoftverfejlesztési technikákat bemutató új fejezet. A mai üzleti kihívások a vállalatok számára azt jelentik, hogy a szoftverüket minél gyorsabban kell leszállítani, amivel az új kihívásokra és lehetőségekre reagálni tudnak. A fejezetben bemutatom az agilis fejlesztési módszereket, kiemelt figyelemet fordítva az extrém programozásra. Ezenfelül még bemutatok néhány környezetet a gyors alkalmazásfejlesztésre, valamint a rendszer-prototípezálás helyes használatát.
2. A 18. és 19. fejezetek témája az újrafelhasználás-alapú szoftvertervezés. Az elmúlt néhány évben a szoftver-újrafelhasználás egyre általánosabbá vált, így mára az újrafelhasználás-alapú fejlesztés a szoftvertervezés egyik meghatározó irányzata lett. A 18. fejezet egy áttekintést ad az újrafelhasználással történő fejlesztésről és szoftver-újrafelhasználásról. A 19. fejezet a komponensalapú szoftvertervezésre összpontosít, beleértve a komponenskompozíciót és a CBSE-folyamatot is.
3. A 20. fejezet a kritikus rendszereket tárgyalja, amelyek végigkísérik a könyvet. Számos fejlesztési irányzatot bemutatok, amelyek a rendszer üzembiztonságát támogatják, beleértve a hibaelkerülést és hibatűrést. Ezenfelül még bemutatom, hogy miként alkalmazhatjuk a programozási szerkezeteket és technikákat az üzembiztonság elérésére. A fejezet záró részében visszatérek a szoftverarchitektúrák témaköréhez, és bemutatok a hibatűrő architektúráis megközelítéseket.
4. A 21. fejezet témája a szoftverevolúció. A szoftverrendszerek számára a változás elkerülhetetlen, és ahelyett, hogy a változtatás folyamatát külön tevékenységnek tekintenénk, úgy gondolom, értelmesebb úgy tekinteni, mint a megkezdett szoftverfejlesztési folyamat folytatását. Ebben a fejezetben az evolúció elkerülhetetlenségét, a szoftverkarbantartást, az evolúciós folyamatot és az ősrendszerek evolúciójával kapcsolatos döntések meghozatalát tárgyalom.

17 GYORS SZOFTVERFEJLESZTÉS

TÉMAKÖRÖK

Ezen fejezet célkitűzése, hogy bemutassa a szoftverfejlesztés néhány olyan megközelítését, amely a gyors szoftverátadást segíti elő. Miután a fejezet végére értünk:

- megértjük, hogyan vezethet egy iteratív, inkrementális szoftverfejlesztési megközelítés a használhatóbb szoftverek gyorsabb átadásához;
- megértjük az agilis fejlesztési módszerek és a dokumentált specifikációkon és terveken alapuló szoftverfejlesztési módszerek közötti különbségeket;
- megismerjük az extrém programozás alapelveit, gyakorlati alkalmazásait és néhány korlátját;
- megértjük, hogyan segíthet a prototipizálás megoldani az elvárásokkal és a tervekkel kapcsolatos bizonytalanságok problémáját, amikor is a fejlesztésnek egy specifikációalapú megközelítéssel kell használni.

TARTALOM

- 17.1. Agilis módszerek
- 17.2. Extrém programozás
- 17.3. Gyors alkalmazásfejlesztés
- 17.4. Szoftverprototípus készítése

A vállalatok ma globális, gyorsan változó környezetben működnek. Reagálnak az új lehetőségekre és piacokra, a gazdasági környezet változásaira, és a versengő termékek és szolgáltatások megjelenésére. A szoftver része minden üzleti műveletnek, így lényeges, hogy egy új szoftvert gyorsan kifejlesszenek, kihasználva ezzel az új lehetőségek adta előnyöket, és válaszolva a versenykihívásokra. A szoftverrendszereknél így ma a legkritikusabb követelmény a gyors fejlesztés és üzembe helyezés. Tény, hogy sok vállalat hajlandó engedni a minőségből és kompromisszumokat kötni a szoftver gyors üzembe helyezése érdekében.

Mivel ezek a vállalatok folyamatosan változó környezetben tevékenykednek, sokszor gyakorlatilag lehetetlen, hogy a szoftverrel szemben stabil elvárásokat fogalmazzanak meg. Az elvárások elkerülhetetlenül változnak, mert az ügyfelek

nem tudják előre megjósolni, mennyire függ majd a rendszer a gyakorlati alkalmazásától, illetve hogy miként fog a többi rendszerrel együttműködni és milyen felhasználói műveleteket lehet majd automatizálni. A valós elvárások csak akkor tisztázódnak, amikor a rendszert már átadták és a felhasználók tapasztalatot szereznek a használata során.

Nem tartoznak a gyors szoftverfejlesztéshez azok a szoftverfejlesztési folyamatok, amelyek az elvárások teljes specifikációján, majd a rendszer tervezésén, elkészítésén és tesztelésén alapulnak. Az elvárások változásával vagy követelménybeli problémák felmerülésével szükségessé válik a rendszerterv, illetve az implementáció átdolgozása és újratesttelése. Ennek következménye, hogy a szokványos egymásra épülő fejlesztési elmélet, illetve a specifikációalapú műveletek gyakran hosszabb időt igényelnek, és a végleges szoftver is csak az eredetileg tervezettnél sokkal később kerül az ügyfélhez.

Ez gyorsan változó üzleti környezetben komoly gondokat okozhat. Idővel a szoftvert elkészül, de addigra a körülmények megváltozása miatt gyakorlatilag használhatatlanná válik. Ennek következtében, különösen a vállalati rendszerek esetében, lényegesek az olyan fejlesztési folyamatok, amelyek a gyors szoftverfejlesztésre és átadásra összpontosítanak.

A gyors szoftverfejlesztési folyamatokat arra tervezték, hogy segítségükkel gyorsan készíthessünk használható szoftvereket. Ez általában iteratív folyamat, ahol a specifikáció, a tervezés, a fejlesztés és a tesztelés átfedi egymást. Ilyenkor a szoftvert nem egy teljes egységként fejlesztik, hanem lépésenként, és minden lépés magában foglalja a rendszer egy további funkcionalitását. Habár a gyors szoftverfejlesztésnek sok megközelítése létezik, néhány közös, alapvető jellemző mindegyiknél fellelhető:

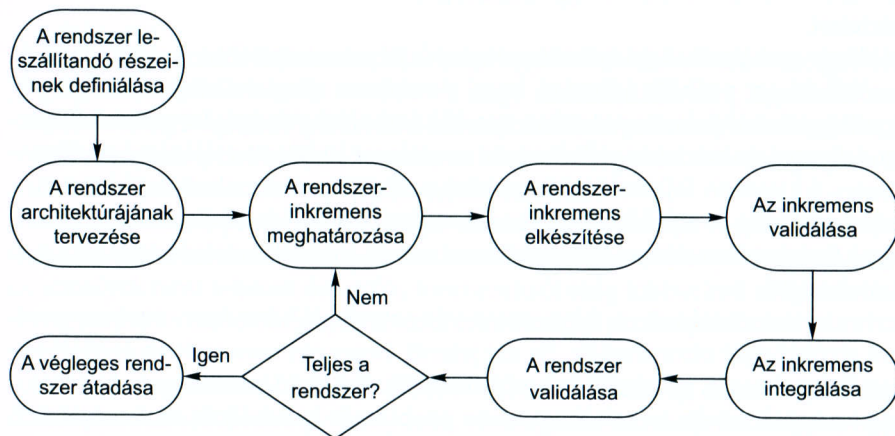
1. A specifikáció, a tervezés és az implementálás folyamata konkurens módon zajlik. Nincs részletes rendszer-specifikáció és a tervezési dokumentáció minimális vagy a rendszert implementáló programozási környezet által automatikusan generált. A felhasználói elvárások leírása csak a rendszer legfontosabb jellemzőit határozza meg.
2. A rendszert lépésről lépésre fejlesztik. A végfelhasználók és a rendszer többi érintettje részt vesznek minden lépés specifikációjában és az eredmény kiértékelésében. Indítványozhatnak változtatásokat a szoftverben, valamint új követelményeket fogalmazhatnak meg, amelyeket a rendszer egy későbbi fejlesztési lépésében kellene megvalósítani.
3. A rendszer felhasználói felülete gyakran egy beépített fejlesztői környezet használatával készül el, amely lehetőséget biztosít az interfész gyors elkészítésére rajzolással és ikonok helyezésével. A rendszer ezután generálni tud egy webalapú felületet a böngészők számára, vagy egy platformspecifikus felületet, mint például a Microsoft Windows ablakai.

Az inkrementális fejlesztés, amelyet a 4. fejezetben mutattam be, maga után vonja, hogy az elkészítés és az átadás lépésenként történjen, nem pedig egyetlen nagy csomagban. Minden egyes lépés a szoftver egy új szintjét eredményezi. Az

inkrementális megközelítés használatának két legfőbb előnye a szoftverfejlesztésben a következő:

1. *Az ügyfélszolgáltatások gyorsított átadása.* A rendszer korai inkremensei tartalmazhatják a leglényegesebb funkcionalitásokat, így az ügyfél a rendszerből már a fejlesztés korai szakaszában is hasznot tud húzni. Az ügyfelek láthatják az elvárásaik gyakorlati megvalósítását, és megfogalmazhatnak változtatásokat is, amelyeket majd a rendszer későbbi kiadásában beépítenek.
2. *Felhasználói elvárások a rendszerrel szemben.* A rendszer felhasználóit be kell vonni az inkrementális fejlesztés folyamatába, hogy visszajelzéseket adjanak a fejlesztőcsapatnak az előző lépés után átadott rendszerről. Az ő részvételük nemcsak azt jelenti, hogy a rendszer sokkal inkább meg fog felelni a követelményeknek, hanem azt is, hogy a rendszer végfelhasználói elkötelezték magukat a szoftver mellett, és szeretnék működő állapotba hozni azt.

Az inkrementális fejlesztés általános folyamatmodellje a 17.1. ábrán látható. Megfigyelhető, hogy a folyamat korai szakaszai az architektúrális tervezésre összpontosítanak. Ha nem fontoljuk meg jól már a folyamat elején a rendszer architektúráját, a rendszer átfogó struktúrája az új szintek átadásával gyakran instabillá válhat.



17.1. ábra. Egy iteratív fejlesztési folyamat

Az inkrementális szoftverfejlesztés, véleményem szerint, a legjobb megközelítés a legtöbb vállalat és személy számára, mert számunkra ez a problémamegoldás legtermészetesebb módja. Ritkán dolgozunk ki teljes megoldásokat a problémáinkra, de a megoldásainkat lépésenként továbbfejlesztjük, illetve viszszalépünk, ha észrevesszük, hogy hibáztunk.

Mindamellet ez a megközelítés sok nehézséget rejt magában, kiváltképpen azoknál a nagyvállalatoknál, amelyek meglehetősen merev eljárásokat alkalmaznak, és olyan szervezeteknél, ahol a szoftverfejlesztést általában külső vállalko-

zásra bízzák. Az iteratív fejlesztés és az inkrementális átadás legfőbb nehézségei a következők:

1. *Kezelési problémák.* A nagy rendszerek számára kialakított szoftverkezelési struktúrák azzal a céllal készülnek, hogy egy olyan szoftverfolyamat-modellel dolgozzunk, amely szabványos részeredményeket generál, hogy felmérhessük munkánk állapotát. Az inkrementálisan fejlesztett rendszerek viszont olyan gyorsan változnak, hogy nagyon drágák lennének, ha túl sok rendszerdokumentációt készítünk. Továbbá az inkrementális fejlesztés néha szokatlan technológiák használatát igényli ahhoz, hogy biztosíthassuk a szoftver lehető leggyorsabb átadását. A vezetők számára nehézséget okozhat a már meglévő munkatársakkal dolgoztatni inkrementális fejlesztési folyamatokon, mert hiányozhat a szükséges szakértelmük.
2. *Szerződésbeli problémák.* A hagyományos szerződés, amelyet az ügyfél és a szoftverfejlesztő köt, a rendszer-specifikáción alapszik. Amikor nincs specifikáció, nehéz megtervezni a rendszerfejlesztési szerződést. Az ügyfelek elégedetlenek lehetnek az olyan szerződéssel, amely egyszerűen csak a projekttel eltöltött időért fizeti a fejlesztőket, mert ez csúszást jelenthet a feladatokban, vagy a költségvetés túllépéséhez vezethet. Ezzel szemben viszont a fejlesztők nem szívesen fogadnak el olyan szerződést, amelyben fix összeg van megnevezve, mert nem tudják felügyelni a végfelhasználók által kért változtatások mértékét.
3. *Validációs problémák.* Egy specifikációalapú folyamat esetében a verifikáció és a validáció azt próbálja igazolni, hogy a rendszer megfelel a specifikációnak. Egy független V & V csapat már a specifikáció elérhetőségét követően elkezdhet dolgozni és tesztek előkészíteni a rendszer implementálásával párhuzamosan. Az iteratív fejlesztési folyamatok próbálják minimálisra csökkenteni a dokumentációt és átfedésben dolgozni a specifikáción és a fejlesztésen. Ennélfogva az inkrementálisan fejlesztett rendszerek független validációja nehezen kivitelezhető.
4. *Karbantartási problémák.* A folyamatos változtatások bármilyen szoftverrendszer struktúráját elronthatják. Ez azt jelenti, hogy aki nem volt az eredeti fejlesztőcsapat tagja, nehezen érthetőnek találhatja majd a szoftvert. A refactoring az egyik módja annak, hogy ezt a problémát redukáljuk, ami azt jelenti, hogy a szoftverstruktúrákat a fejlesztési folyamat alatt allandóan tökéletesítjük. Ezt a 17.2. alfejezetben tárgyalom, ahol is az extrém programozásról lesz szó. Továbbá ha különleges technológiát, mint például RAD-környezetet (17.3. alfejezet) használunk a gyors fejlesztés megalapozására, a RAD-technológia elavulttá válhat, ezért nehéz lehet olyan embert találni, aki rendelkezik a rendszer kezeléséhez szükséges tudással.

Természetesen léteznek olyan rendszertípusok, amelyek esetében az inkrementális fejlesztés és átadás nem a legjobb megközelítés. Ezek a nagyon nagy rendszerek, amelyek esetében a fejlesztésen több, más-más helyen lévő csapat dolgozik, vagy néhány beágyazott rendszer, amelyeknél a szoftver erősen függ a

hardverfejlesztéstől, és néhány kritikus rendszer, amelyek esetében minden követelményt elemezni kell, hogy átvizsgáljuk a rendszert olyan kölcsönhatások után kutatva, amelyek veszélyeztethetik a rendszer biztonságát.

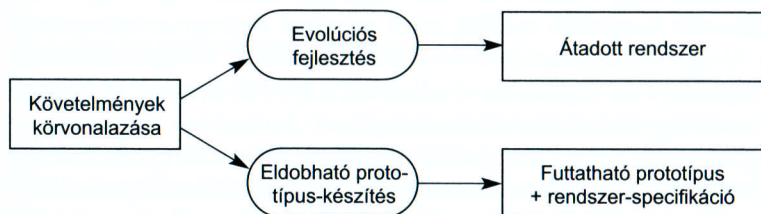
Ezek a rendszerek persze ugyanúgy szenvednek a kétéssé és folyton változó követelmények okozta problémáktól. Ezért a problémák megkeresésére, illetve azért, hogy valami hasznunk is származzon az inkrementális fejlesztés adta lehetőségekből, egy hibrid folyamatot használhatunk, ahol iteratívan fejlesztjük a rendszerprototípust, és ezt platformként használhatjuk a rendszerkövetelményekkel és tervekkel való kísérletezéshez. A prototípusból nyert tapasztalatok megerősíthetnek bennünket arról, hogy a követelmények a rendszer felhasználói valós elvárásainak megfelelnek.

Itt a *prototípus-készítés* fogalma alatt olyan kísérleti rendszerek kifejlesztésének iteratív folyamatát értem, amelyet *nem* az ügyfélhez történő kihelyezésre szánunk. Egy rendszerprototípus azért kerül kifejlesztésre, hogy a szoftverfejlesztőnek és az ügyfélnek segítsen megérteni, valójában mit is kell implementálni. Mindamellelt az *evolúciós prototípus-készítés* fogalmát néha az inkrementális szoftverfejlesztés szinonimájaként szokták használni. A prototípus-készítéssel nem hagyunk fel, hanem folyamatosan fejlesztjük a prototípust, hogy az ügyfél követelményeinek megfeleljen.

A 17.2. ábra azt mutatja be, hogy az inkrementális fejlesztésnek és a prototípus készítésének különböző célkitűzései vannak:

1. Az inkrementális fejlesztés célja, hogy egy működő rendszert szolgáltatson a végfelhasználóknak. Ez azt jelenti, hogy normál esetben a leginkább megértett és a legmagasabb prioritású felhasználói követelményekkel indulunk. Az alacsonyabb prioritású és homályos követelményeket akkor implementáljuk, amikor a felhasználóknak szükségük lesz rájuk.
2. Az eldobható prototípus-készítés célja, hogy validálja vagy származtassa a rendszerkövetelményeket. Ekkor azokkal a követelményekkel kell kezdeni, amelyek nem teljesen érthetők, mert ezekről még többet kell megtudnunk. Az egyszerűbb követelményeket nem is szükséges prototípusolni.

A két megközelítés közötti további fontos különbség a rendszer minőségpolitikája. Az eldobható prototípusok rövid élettartamúak. Szükséges, hogy lehetőség legyen a fejlesztés folyamán ezek gyors változtatására, de hosszú távú kezelhetőségre nincs szükség. Akár a gyenge teljesítmény és megbízhatóság is



17.2. ábra. Inkrementális fejlesztés és prototípus-készítés

hardverfejlesztéstől, és néhány kritikus rendszer, amelyek esetében minden követelményt elemezni kell, hogy átvizsgáljuk a rendszert olyan kölcsönhatások után kutatva, amelyek veszélyeztethetik a rendszer biztonságát.

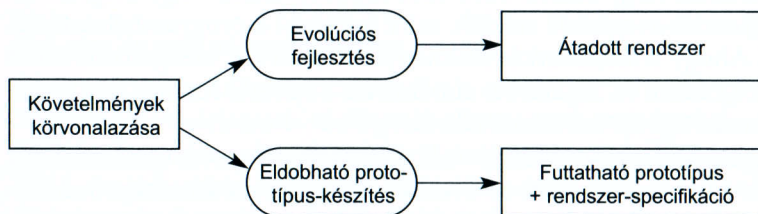
Ezek a rendszerek persze ugyanúgy szenvednek a kétes és folyton változó követelmények okozta problémáktól. Ezért a problémák megkeresésére, illetve azért, hogy valami hasznunk is származzon az inkrementális fejlesztés adta lehetőségekből, egy hibrid folyamatot használhatunk, ahol iteratívan fejlesztjük a rendszerprototípust, és ezt platformként használhatjuk a rendszerkövetelményekkel és tervekkel való kísérletezéshez. A prototípusból nyert tapasztalatok megerősíthetnek bennünket arról, hogy a követelmények a rendszer felhasználói valós elvárásainak megfelelnek.

Itt a *prototípus-készítés* fogalma alatt olyan kísérleti rendszerek kifejlesztésének iteratív folyamatát értem, amelyet *nem* az ügyfélhez történő kihelyezésre szánunk. Egy rendszerprototípus azért kerül kifejlesztésre, hogy a szoftverfejlesztőnek és az ügyfélnek segítsen megérteni, valójában mit is kell implementálni. Mindamellet az *evolúciós prototípus-készítés* fogalmát néha az inkrementális szoftverfejlesztés szinonimájaként szokták használni. A prototípus-készítéssel nem hagyunk fel, hanem folyamatosan fejlesztjük a prototípust, hogy az ügyfél követelményeinek megfelelően.

A 17.2. ábra azt mutatja be, hogy az inkrementális fejlesztésnek és a prototípus készítésének különböző célkitűzései vannak:

1. Az inkrementális fejlesztés célja, hogy egy működő rendszert szolgáltatson a végfelhasználóknak. Ez azt jelenti, hogy normál esetben a leginkább megértett és a legmagasabb prioritású felhasználói követelményekkel indulunk. Az alacsonyabb prioritású és homályos követelményeket akkor implementáljuk, amikor a felhasználóknak szükségük lesz rájuk.
2. Az eldobható prototípus-készítés célja, hogy validálja vagy származtassa a rendszerkövetelményeket. Ekkor azokkal a követelményekkel kell kezdeni, amelyek nem teljesen érthetők, mert ezekről még többet kell megtudnunk. Az egyszerűbb követelményeket nem is szükséges prototipizálni.

A két megközelítés közötti további fontos különbség a rendszer minőségpolitikája. Az eldobható prototípusok rövid élettartamúak. Szükséges, hogy lehetőség legyen a fejlesztés folyamán ezek gyors változtatására, de hosszú távú kezelhetőségre nincs szükség. Akár a gyenge teljesítmény és megbízhatóság is



17.2. ábra. Inkrementális fejlesztés és prototípus-készítés

elfogadható egy eldobható prototípus esetében mindaddig, amíg elősegíti a követelmények megértését.

Ezzel szemben az inkrementális fejlesztésű rendszereket, ahol a kezdeti verziókból fejlődik ki a végső rendszer, ugyanolyan szervezeti minőségben kell fejleszteni, mint minden más szoftvert. Ezeknek szilárd struktúrával kell rendelkezniük annak érdekében, hogy éveken át karbantarthatók legyenek. Megbízhatónak és hatékonynak kell lenniük, illetve meg kell felelniük a megfelelő szervezeti szabványoknak is.

17.1. AGILIS MÓDSZEREK

Az 1980-as években és az 1990-es évek elején volt egy nagyon széles körben elterjedt nézet, ami szerint a jobb szoftver előállításának legjobb módja, hogy gondosan megtervezzük a projektet, formalizáljuk a minőséggel szembeni követelményeinket, használjuk a CASE-eszközök által biztosított elemzési és tervezési módszereket, és irányított, precíz szoftverfejlesztési folyamatok használatával jutunk el a végeredményhez. Ezt a szemléletet azok a szoftvertervezői közösségek képviselik, akik nagy, nagyszámú különálló programból felépülő hosszú élettartamú szoftverrendszerek fejlesztésével foglalkoztak.

Ezeknek a programoknak a nagy többsége általában valamilyen kritikus rendszer volt (3. fejezet). Ezeket a szoftvereket nagy létszámú csapatok fejlesztették, akik néha különböző cégeknél dolgoztak. A csapatok gyakran még földrajzilag is szétszórtan helyezkedtek el, és hosszú ideig dolgoztak a szoftveren. Példa ilyen típusú szoftverre a modern repülőik irányítórendszere, amelynél akár 10 évig is eltarthat, míg az eredeti specifikációtól eljut az átadásig. Az ilyen megközelítések (néhányról ebben a könyvben is szó van) jelentős mértékű többletmunkát vonnak maguk után a rendszer átgondolásának, tervezésének és dokumentációjának fázisában. Ez a többletteleher viszont kiegyenlítődik, amikor több fejlesztőcsapat munkáját kell koordinálni, vagy amikor egy kritikus rendszerről van szó, illetve amikor tudjuk, hogy a szoftver élettartama alatt számos különböző ember lesz megbízva a rendszer karbantartásával.

Mindamellet, amikor ezt a komoly, terveken alapuló fejlesztési megközelítést kis- és középvállalkozások rendszereinek fejlesztésekor alkalmazták, a megjelenő többletmunka olyan mértékű volt, hogy gyakran ez határozta meg a szoftverfejlesztés folyamatának nagy részét. Több időt töltöttek azzal, hogy megtervezzék a rendszerfejlesztés megfelelő módját, mint magával a programfejlesztéssel és a teszteléssel. Ahogy a rendszerkövetelmények változnak, elengedhetetlenül szükség lesz átdolgozásra és, legalábbis elméletben, a specifikációnak és a tervezésnek a programmal együtt kellene változnia.

Ezzel az időigényes megközelítéssel való elégedetlenség oda vezetett, hogy szoftverfejlesztők egy része az 1990-es években új, agilis módszereket indítványozott. Ezek megengedték a fejlesztőcsapatnak, hogy magára a szoftverre koncentráljanak, nem pedig annak tervezésére és dokumentálására. Ezek az agilis

módszerek egységesen a szoftverspecifikáció, a fejlesztés és az átadás iteratív megközelítésére alapoztak, és elsősorban arra szánták, hogy olyan vállalati alkalmazásfejlesztést támogassanak, amelyek esetében a rendszerkövetelmények gyakori változtatásokon esnek át a fejlesztési folyamat alatt. Ezeknek a módszereknek az a célja, hogy a működő szoftvert minél hamarabb átadhassák az ügyfélnek, aki ezt követően új és változtatott követelményeket javasolhat, amelyek egy későbbi iteráció folyamán bekerülhetnek a rendszerbe.

Talán a legismertebb agilis módszer az extrém programozás (Beck, 1999; 2000), amelyet a fejezet későbbi részében ismertetek. Mindamellett más agilis megközelítések is léteznek: Scrum (Schwaber és Beedle, 2001); Crystal (Cockburn, 2001); Adaptive Software Development (Highsmith, 2000); DSDM (Stapleton, 1997); Feature Driven Development (Palmer és Felsing, 2002). Ezeknek a módszereknek a sikere oda vezetett, hogy több hagyományos, rendszermodellezésen alapuló fejlesztési módszert is magukba integráltak, eljutva az agilis modellezés fogalmához (Ambler és Jeffries, 2002) és a Rational Unified Process (Larman, 2002) agilis változatához.

Annak ellenére, hogy ezek az agilis módszerek mind az inkrementális fejlesztés és átadás fogalmán alapulnak, különböző folyamatokat ajánlanak ennek eléréséhez. Mindamellett van néhány közös alapelvük, és ennek következtében sok hasonlóság is van közöttük. Ezek az alapelvek a 17.3. ábrán láthatók.

Alapelv	Leírás
Az ügyfél bevonása	Az ügyfeleket minél jobban be kell vonni a fejlesztési folyamatba. A szerepük az, hogy a rendszerkövetelményekről gondoskodjanak, azokat prioritizálják, és kiértékeljék a rendszer minden egyes iterációját.
Inkrementális átadás	A szoftver lépésenként fejlődik, az ügyfél adja meg a követelményeket, amelyeket majd az egyes lépésekben implementálni kell.
Az emberek nem folyamatok	A fejlesztőcsapat képességeit ismerni kell és ki is kell használni. Hagyjuk, hogy a csapattagok a feladatokat a saját módszerükkel végezzék el, nem pedig előírt folyamatok szerint.
Változtatási lehetőség	Számítani kell a rendszerkövetelmények változására, és úgy kell tervezni a rendszert, hogy azzal összeegyeztethetők legyenek ezek a változások.
Kezelési egyszerűség	Az egyszerűségekre kell koncentrálni mind a fejlesztendő szoftver, mind pedig a fejlesztési folyamat tekintetében. Amikor csak lehet, aktívan dolgozni kell a rendszer komplexitásának csökkentésén.

17.3. ábra. Az agilis módszerek alapelvei

Az agilis módszerek támogatói elhivatottan terjesztették módszereik használatát, és gyakran figyelmen kívül hagyták azok hátrányait, ami szélsőséges, bár a megközelítés problémáihoz mérten talán eltúlzott visszhangot váltott ki (Stephens és Rosenberg, 2003). Sokkal elfogadhatóbb az a kritika, amit például DeMarco és Boehm (DeMarco és Boehm, 2002) fogalmaz meg, hangsúlyozva az

agilis módszerek előnyeit és hátrányait. Szerintük a legjobb választás a hibrid megközelítés lenne, ahol az agilis módszerek magukban foglalnának néhány tervezeten alapuló fejlesztési technikát is.

A gyakorlatban azonban az agilis módszerek alapelveit néha nagyon nehéz megvalósítani:

1. Amíg az ügyfél fejlesztési folyamatba való bevonásának ötlete vonzó, annak sikere attól függ, hogy az ügyfél tud is és akar is időt tölteni a fejlesztőcsapattal, illetve aki képviselni tudja a rendszer minden érintettjét. Gyakran az ügyfél képviselőjének más elfoglaltsága van, és ezért nem tud teljes mértékben részt venni a szoftverfejlesztésben.
2. Előfordulhat, hogy az egyes csapattagok nem rendelkeznek megfelelő személyi adottságokkal az intenzív részvételhez, ami viszont jellemző az agilis módszerek esetében. Ennek következtében nem tudnak megfelelően együttműködni a csapat többi tagjával.
3. A változtatások fontossági sorrend szerint való rendezése meglehetősen nehéz, különösen olyan rendszereknél, amelyeknél sok az érintett. Az érintettek tipikusan különböző prioritásokat adnak a különböző változtatásoknak.
4. Az egyszerűség fenntartása külön munkát igényel. Az ütemterv és az átadás sürgőssége keltette nyomás alatt a csapattagoknak esetleg nincs elég idejük arra, hogy a rendszert egyszerűsítsék.

Az inkrementális fejlesztéssel és átadással van egy másik nem technikai, de általános probléma, ami akkor jön elő, ha a rendszer megrendelője egy külső szervezetet bíz meg a rendszerfejlesztéssel. A 6. fejezetben megbeszéltek alapján a szoftverkövetelmények dokumentuma általában része a szerződésnek, amit az ügyfél és a szolgáltató köt. Mivel az agilis módszerek velejárója az inkrementális specifikáció, a szerződések megfogalmazása ilyen fejlesztések esetén nehézkes lehet.

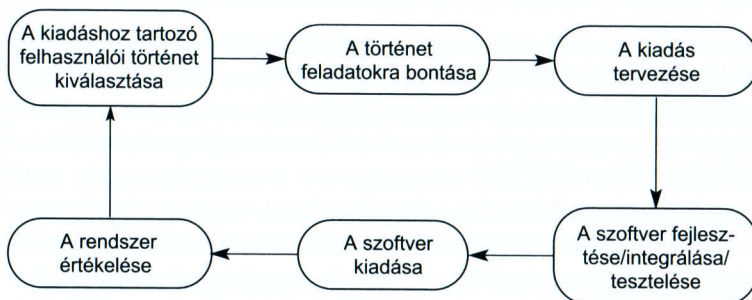
Következésképpen az agilis módszereket olyan szerződésekre kell alapozni, ahol az ügyfél a rendszerfejlesztéshez szükséges időért fizet egy bizonyos követelmény kifejlesztése helyett. Ha azonban minden jól megy, ez hasznot hoz mind az ügyfélnek, mind a fejlesztőnek. Mindamellet ha problémák merülnek fel, komoly viták adódhatnak abból, hogy ki a hibás és kinek kell fizetnie a többletmunkát és -erőforrást, ami a probléma megoldásához szükséges.

Minden módszernek megvannak tehát a korlátai, és az agilis módszerek csak bizonyos típusú rendszerfejlesztésre felelnek meg. Szerintem ezek a módszerek a kis- és középvállalkozások rendszereinek, illetve a személyi számítógépes termékek fejlesztésére a legalkalmasabbak. Nem igazán alkalmasak nagyobb kaliberű rendszerek fejlesztésére, ahol a fejlesztőcsapatok különböző helyen dolgoznak, és ahol komplex kölcsönhatások léphetnek fel más hardver- és szoftverrendszerekkel. Az agilis módszerek kritikus rendszerek fejlesztésére sem ajánlottak, ahol a rendszerkövetelmények részletes elemzése szükséges annak érdekében, hogy megértsük azok biztonságosságának vagy védettségének a jelentőségét.

17.2. EXTRÉM PROGRAMOZÁS

Az extrém programozás (XP) talán a legismertebb és legszélesebb körben használt agilis módszer. A nevet Beck (Beck, 2000) adta, mert a megközelítés úgy fejlődött ki, hogy elismert gyakorlati alkalmazásokat erőltetett, mint például az iteratív fejlesztés és az ügyfél „extrém” részvételének használata.

Az extrém programozásban minden követelményt forgatókönyvként állítanak össze (ezt hívják felhasználói történetnek), amely közvetlenül feladatok egy soraként kerül implementálásra. A programozók párokban dolgoznak és minden feladatra teszteket készítenek, még mielőtt a kódot megírnák. Minden testnek sikeresen le kell futnia, mielőtt az új kódot elhelyeznék a rendszerben. A rendszer kiadásai között csak kis idő telik el. A 17.4. ábra az XP folyamatát mutatja be, amint a fejlesztés alatt álló rendszer egy új inkremensét állítja elő.



17.4. ábra. Az extrém programozás kiadási ciklusa

Az extrém programozás magában foglal néhány gyakorlati alkalmazást (17.5. ábra), amelyek megfelelnek az agilis módszerek alapelveinek:

1. Az inkrementális fejlesztés a rendszer kisméretű, gyakori kiadásán keresztül valósul meg, valamint a követelmények leírásának egy olyan megközelítésével, ami az ügyfél-történetek vagy forgatókönyvek alapján a folyamattervezés alapjául szolgálhat.
2. Az ügyfél részvételének biztosítása elérhető a fejlesztőcsapatba történő teljes munkaidős bevonásával. Az ügyfél képviselője részt vesz a fejlesztésben és a rendszer elfogadási tesztjeinek meghatározásáért felelős.
3. A párban való programozás, a rendszerkód fölötti együttes tulajdonjog, illetve egy olyan fenntartható fejlesztési folyamat, ami nem von maga után felesleges munkaórákat, az embereket támogatja, nem a folyamatokat.
4. A változtatásokat a rendszeres rendszerverziók, az előrehozott tesztelést alkalmazó fejlesztés és a folyamatos integráció támogatja.
5. Az egyszerűség fenntartásához szükséges a kód minőségének folyamatos tökéletesítéssel történő növelése, valamint az egyszerű tervek használata a rendszer jövőbeli változtatásainak elkerülésére.

Fogalom	Definíció
Inkrementális tervezés	A követelményeket történetkártyákra jegyzik, a történeteknek egy kiadásba történő bekerülése a rendelkezésre álló időtől és a relatív prioritásuktól függ. A fejlesztők ezeket a történeteket fejlesztési feladatokra osztják (17.6. és 17.7. ábra).
Kisméretű kiadások	Azokat a funkcionálisokat fejlesztik először, amelyek minimálisak, hasznosak és üzleti hasznot hozhatnak. A rendszer kiadásai gyakoriak, és lépésenként adnak új funkcionálisokat a legelső kiadáshoz.
Egyszerű tervezés	Az egyszerű tervezés megvalósítja a követelményekben leírtakat, de nem többet.
Fejlesztés előrehozott teszteléssel	Automatikus, egységes tesztrendszert használnak, hogy teszteket írjanak egy új funkcionális ellenőrzésére, még mielőtt az implementálásra kerülne.
Refactoring	Minden fejlesztővel szemben elvárás, hogy folyamatosan tökéletesítse a kódot, amint arra lehetőséget lát. Ez egyszerűen kezelhetővé teszi a kódot.
Páros programozás	A fejlesztők párokban dolgoznak, egymás munkáját ellenőrzik, és biztosítják a támogatást, hogy mindig jó munkát végezzenek.
Közös tulajdonjog	A fejlesztőpárok a rendszer minden területén dolgoznak, ezért nincsenek fejlesztési szakterületek, és a fejlesztők közösen birtokolják az egész kódot. Bárki bármit megváltoztathat.
Folyamatos integrálás	Amint egy feladat elkészül, azt beintegrálják a teljes rendszerbe. Minden ilyen integrálás után a rendszernek az összes egységeszten meg kell felelnie.
Fenntartható iram	A sok túlóra nem megfelelő, mert ez gyakran ronthatja a kód minőségét és a középtávú termelékenységet.
Helyszíni ügyfél	A rendszer végfelhasználója egy képviselőjének (az ügyfél) teljes munkaidőben az XP-csapat rendelkezésére kell állnia. Ezért az extrém programozási folyamatban az ügyfél a fejlesztőcsapat tagja, és felelős azért, hogy a csapatot ellássa az implementációhoz szükséges rendszerkövetelményekkel.

17.5. ábra. Extrém programozási gyakorlatok

Egy XP-folyamatban az ügyfél bensőséges szerepet játszik a rendszerkövetelmények specifikációjában és fontossági sorrendjének meghatározásában. A követelményeket nem a szükséges rendszerfunkciók listájaként adják meg, hanem a rendszer megrendelője a fejlesztőcsapat tagjaként megbeszéli a forgatókönyvet a csapat többi tagjával. Közös kifejlesztésként egy „történetkártyát”, ami magában foglalja az ügyfél kéréseit. A fejlesztőcsapat ezután megpróbálja implementálni a forgatókönyvet a szoftver egy jövőbeli kiadásában. Példa ilyen történetkártyára a 6. fejezetben található LIBSYS-rendszer, ami egy forgatókönyv alapján készült és a 17.6. ábrán látható.

Amint a történetkártyák elkészültek, a fejlesztőcsapat ezt feladatokra bontja, és megbecsüli az implementáláshoz szükséges időt és erőforrásokat. Az ügyfél ezután rangsorolja az implementálandó történeteket, kiválasztva azokat, amelyek azonnal felhasználhatók üzleti célok támogatására. Természetesen a követelmények változásával, a nem implementált történetek változhatnak vagy elhagyha-

Egy cikk letöltése és kinyomtatása

Először válasszuk ki egy megjelenített listából a kívánt cikket. Aztán adjuk meg a rendszernek a fizetési módot – ez történhet előfizetéssel, vállalati számláról vagy hitelkártyával.

Ezután kapunk egy kitöltendő szerzői jogi űrlapot. Ha ezt elküldtük, a cikk letöltődik a számítógépünkre.

Ezután választunk egy nyomtatót, és kinyomtatjuk a cikk egy másolatát. Közzöljük a rendszerrel a sikeres nyomtatás tényét.

Ha a cikk csak nyomtatható, akkor nem őrizhetjük meg a PDF-verziót, így az automatikusan törlődik a számítógépünkről.

17.6. ábra. Dokumentumletöltés történetkártyája

tók. Ha változtatások szükségesek egy olyan rendszerben, amelyet már átadtak, új történetkártyákat kell készíteni, és megint csak az ügyfél dönti el, hogy ezeknek a változtatásoknak elsőbbségük van-e az új funkcionalitással szemben vagy sem.

Az extrém programozás az iteratív fejlesztés „extrém” megközelítését használja. Naponta többször is megjelenhet a szoftver új verziója, de az inkremensek nagyjából kéthetente kerülnek csak az ügyfélhez. Amikor a programozó létrehozza a rendszer következő verzióját, minden létező automatizált teszten végig kell futtatnia, beleértve az új funkcionalitásokhoz készült teszteket is. Az új verziót pedig csak akkor lehet elfogadni, ha minden teszt sikeresen lefutott.

A hagyományos szoftvertervezés egyik alapelve, hogy a változtatásokra kell tervezni. Ez azt jelenti, hogy előre kell látni a szoftver jövőbeli változtatásait, és már eleve úgy kell megtervezni, hogy azokat a változtatásokat egyszerűen lehessen implementálni. Az extrém programozás azonban mégis figyelmen kívül hagyta azokat az alapelveket, amelyeket arra alapoznak, hogy a változtatásokra történő tervezés időpocsékolás. Gyakran nem a várt változtatások, hanem teljesen más változtatási kérések valósulnak majd meg.

Az előre nem várt változtatások implementálásával az a probléma, hogy rontják a szoftver struktúráját, ami miatt a megvalósíthatóságuk egyre nehezebbé és nehezebbé válik. Az extrém programozás úgy oldja meg ezt a problémát, hogy a szoftver folyamatos kódátszervezését javasolja. Ez azt jelenti, hogy a programozócsapat tökéletesítési lehetőségeket keres a szoftverben, és azokat azonnal implementálja. Ennek köszönhetően a szoftver mindig könnyen érthető és könnyen változtatható marad az új történetek implementálását követően is.

17.2.1. Tesztelés az XP-ben

Ahogy az a fejezet bevezetésében már említettem, az egyik legfontosabb különbség az iteratív fejlesztés és a tervalapú fejlesztés között a rendszer tesztelésének módja. Iteratív fejlesztés esetén nincs rendszer-specifikáció, amit egy külső

tesztelő használhatja a rendszer tesztjeinek fejlesztéséhez. Következésképpen az iteratív fejlesztés néhány megközelítésénél igencsak informális a tesztelés folyamata.

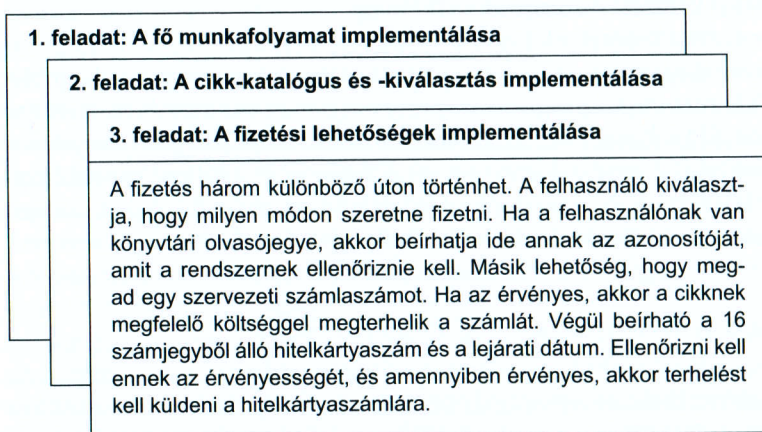
A teszteléssel és a rendszer validálásával kapcsolatos problémák elkerülésére az XP nagyobb hangsúlyt fektet a tesztelés folyamatára, mint a többi agilis módszer. A rendszertesztelés központi szerepet játszik az XP-ben, ahol is egy olyan megközelítést dolgoztak ki, ami csökkenti annak a valószínűségét, hogy az új rendszerinkremensekkel hibák kerüljenek a már létező szoftverbe.

Az XP-ben történő tesztelés kulcsjellemzői a következők:

1. előrehozott teszteléssel történő fejlesztés;
2. inkrementális tesztfejlesztés a forgatókönyvek alapján;
3. felhasználók bevonása a tesztek fejlesztésébe és a validálásba;
4. automatizált tesztelőeszközök használata.

Az előrehozott teszteléssel történő fejlesztés az XP egyik legfontosabb újítása. Az előre elkészített tesztek mind az interfészt, mind a viselkedési specifikációt meghatározza a fejlesztendő funkcionalitás számára. Ezzel csökkennek a követelményproblémák és az interfészek félreértései. Ez a megközelítés minden olyan folyamat esetében alkalmazható, ahol tisztán látható a rendszerkövetelmény és az azt implementáló kód közötti kapcsolat. Az extrém programozásban mindig látható ez a kapcsolat, mert a követelményeket reprezentáló történetkártyák feladatokra vannak bontva, és a feladatok mind egy-egy implementációs egységet jelentenek.

Ahogy már említettem, az XP-ben a felhasználói követelmények forgatókönyvekként vagy történetekként jelennek meg, és a felhasználó rangsorolja ezeket a fejlesztéshez. A fejlesztőcsapat felmér minden forgatókönyvet, és feladatokra bontja azt. Minden egyes feladat a rendszer egy-egy különálló jellemzőjét reprezentálja, és egy egységtesztet lehet tervezni a feladathoz. A példa kedvéért a dokumentumletöltéshez készült történetkártya néhány feladatkártyája a 17.7. ábrán látható.



17.7. ábra. A dokumentumletöltés feladatkártyái

4. teszt: Hitelkártya érvényességének ellenőrzése**Bemenet:**

A hitelkártyaszámot egy karaktersorozat, míg a kártya lejáratí hó-napját és évét két egész szám reprezentálja.

Tesztek:

Ellenőrizni kell, hogy a karaktersorozat minden bájtja számjegy-e.

Ellenőrizni kell, hogy a hónap 1 és 12 között van-e és az év nagyobb vagy egyenlő-e az aktuális évvel.

A hitelkártya számának első 4 számjegye alapján ellenőrizni kell a kibocsátó érvényességét a kártyakibocsátók táblázata alapján.

A hitelkártya érvényességét ellenőrizzük úgy, hogy elküldjük a kártyaszámot és a lejáratí dátuminformációt a kártya kibocsátójához.

Kimenet:

OK vagy hibaüzenet, ami a kártya érvénytelenségét jelzi.

17.8. ábra. Tesztesetleírások a hitelkártya érvényességének ellenőrzésére

Minden feladat egy vagy több egységtesztet eredményez, ami a feladatban leírt implementációt ellenőrizi. A 17.8. ábrán egy olyan tesztet rövidített leírása látható, amely azt ellenőrizi, hogy érvényes hitelkártyaszám lett-e implementálva.

Az ügyfél szerepe a tesztelési folyamatban az, hogy a történetekhez segítsen elfogadási teszteket fejleszteni, amelyeket a rendszer következő kiadásában implementálni kell. Ahogy arról a 23. fejezetben szó lesz, az *elfogadási tesztelés* az a folyamat, amikor a rendszert az ügyfél adataival ellenőrizzük, hogy megfelel-e az ügyfél valódi elvárásainak. Az XP-ben az elfogadási tesztelés ugyanúgy inkrementális, mint a fejlesztés. Ennél az előbbi történetnél az elfogadási tesztben szerepelni fog számos dokumentum kiválasztása, a díjaik különböző módon történő kifizetése, valamint a különböző nyomtatókon történő kinyomtatásuk. A gyakorlatban viszont elfogadási tesztek egész sorát szokták készíteni.

Az előrehozott teszteléssel történő fejlesztés és az automatikus teszteszközök használata az XP-irányzat fontos erősségei. Ahelyett, hogy megírnánk a programkódot, majd aztán a kódhoz tartozó teszteket, az előrehozott teszteléssel történő fejlesztés azt jelenti, hogy a tesztet hamarabb megírjuk, mint a kódot. Jobban megvizsgálva, a tesztet mint futtatható komponenst előbb írjuk meg, mint ahogy a feladatot implementálnánk. Amint a szoftver implementációja kész, rögtön futtatható rajta a teszt. Ennek a tesztelőkomponensnek függetlennek kell lennie, szimulálnia kell a tesztelendő bemenetek megadását, és ellenőriznie kell, hogy a végeredmény megfelel-e a kimenet specifikációjának. Az automatikus teszteszköz egy olyan rendszer, ami elküldi ezeket az automatikus teszteket végrehajtásra.

Az előrehozott teszteléssel történő fejlesztés esetében mindig vannak olyan tesztek, amelyek gyorsan és könnyedén futtathatók. Ez azt jelenti, hogy ahányszor egy új funkcionalitás kerül a rendszerbe, a tesztek már le is futtathatók rajta, és rögtön előjönnek azok a problémák, amelyeket az új kód okozott.

Az előrehozott teszteléssel történő fejlesztés esetében a feladat implementálónak átfogóan érteniük kell a specifikációt, hogy tesztek készíthessenek a rendszerhez. Ez azt jelenti, hogy a specifikációban lévő félreérthetőségeket és hiányosságokat tisztázni kell még az implementálás megkezdése előtt. Továbbá a módszer szintén elkerüli a „lemaradt teszt” problémáját, ami akkor léphet fel, ha a rendszer fejlesztője nagyobb lépésekben dolgozik, mint a tesztelő, így az implementáció egyre jobban és jobban megelőzi a tesztelést, ami miatt néha kihagynak bizonyos tesztek, hogy tartani tudják az ütemtervet.

Mindamellett az előrehozott teszteléssel történő fejlesztés nem mindig működik úgy, ahogy várnánk. A programozók jobban szeretnek programozni, mint tesztelni, és ezért néha hiányos tesztek írók, amelyek a szélsőséges szituációkat nem ellenőrzik. Ráadásul néhány tesztet nagyon nehéz is megírni. Például egy komplex felhasználói felület esetében gyakran nehezebb egységteszteket írni a kódhoz, amely implementálja a megjelenítési logikát és a képernyők közötti munkafolyamatot. Végül nagyon nehéz megítélni a tesztek befejezettségét. Annak ellenére, hogy sok rendszertesztünk van, lehet, hogy a tesztjeink nem biztosítanak teljes körű ellenőrzést. Lehet, hogy a rendszer kritikus részei nem kerültek futtatásra, így azok teszteletlenül maradnak.

Az ügyfélre alapozni abban, hogy támogassa az elfogadási tesztek fejlesztését, néha az XP tesztelési folyamatának legnagyobb nehézségét jelenti. Az ügyfél szerepét betöltő személynek korlátozott a rendelkezésre álló ideje, és lehet, hogy nem tud teljes munkaidőben együtt dolgozni a fejlesztőcsapattal. Az ügyfél úgy érezheti, hogy a követelmények megadása elégséges, és tiltakozhat a tesztelési folyamatokba történő bevonása ellen.

17.2.2. Páros programozás

Egy másik bevezetésre került innovatív gyakorlati alkalmazás az, hogy a programozók párban dolgoznak a szoftverfejlesztés alatt. Gyakorlatilag együtt ülnek le ugyanahhoz a munkaállomáshoz a szoftver fejlesztésére. A fejlesztés nem mindig vonja magával, hogy ugyanazok a párok dolgozzanak együtt. Az alapötlet a párok dinamikus választása, így a fejlesztési folyamat során a csapat minden tagja dolgozik párban a többi taggal.

A párban való programozásnak a következő előnyei vannak:

1. Támogatja a rendszerre vonatkozó közös tulajdon és felelősségtudat érzését. Ez Weinberg (Weinberg, 1971) önzetlen programozás eszméjét tükrözi, ami alapján a szoftver a csapat egészének tulajdona és az egyes személyek nem tartoznak felelősséggel a kód problémáiért. Helyette a csapatnak kollektív felelőssége van a problémák megoldása tekintetében.
2. Úgy működik, mint valamilyen informális átvizsgálási folyamat, mert minden egyes kódsort legalább két ember átnéz. A kód áttekintése és vizsgálata (22. fejezet) nagyon eredményes lehet a szoftverhibák nagy százalékának a felderítésére.

sében. Mindamellett időigényes lehet a megszervezése, és tipikusan késéseket is okozhat a fejlesztési folyamatban. Amíg a párban való programozás egy kevésbé formális folyamat, ami valószínűleg nem fed fel annyira sok hibát, de egy sokkal olcsóbb átvizsgálási folyamat, mint a formális programátvizsgálás.

3. Segít a kódátszervezésben, ami a szoftver tökéletesítésének egy folyamata. Az XP egyik alapelve, hogy a szoftver folyamatos kódátszervezést igényel. Ez azt jelenti, hogy a kód egyes részeit újra kell írni, hogy tökéletesítsük annak átláthatóságát vagy struktúráját. Hagyományos fejlesztési környezetben ennek a gyakorlati alkalmazása nehézségeket okozhat, mert ez a munka a hosszú távú előnyök érdekében szükséges, és a kódátszervezéssel foglalkozó egyén így kevésbé hatékony munkaerőnek ítéltető, mint az, aki egyszerűen csak fejleszti a kódot. Ahol viszont a párban való programozást és a kollektív tulajdonjogot alkalmazzák, már azonnal előnyhöz jutnak a kódátszervezés által, és ezért valószínűleg támogatni fogják a folyamatot.

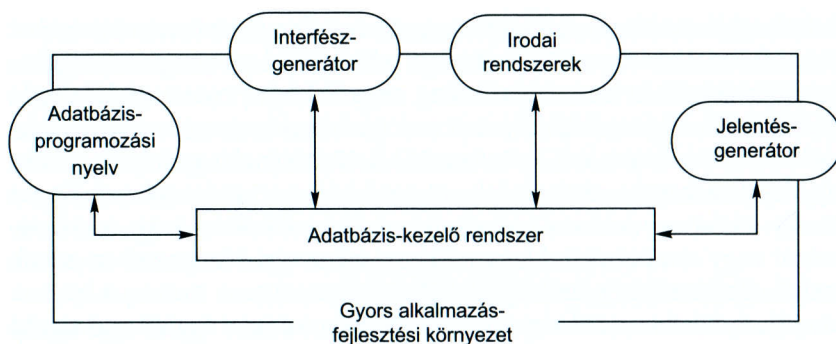
Azt gondolhatnánk, hogy a párban való programozás kevésbé hatékony, mint az egyéni programozás, és a fejlesztőpárok csak feleannyi kódot produkálnak majd, mintha a két egyén külön-külön dolgozna. Az XP-fejlesztésekkel foglalkozó tanulmányok azonban nem ezt igazolják. A fejlesztői eredményesség a párban való programozás esetében úgy tűnik, vetekedik a két külön dolgozó egyén termelékenységével (Williams és mások, 2000). Ennek az oka, hogy a párok megbeszélnek a szoftverrel kapcsolatos kérdéseket még mielőtt nekilátnának a fejlesztésnek, és ezért valószínűleg kevesebb hibás startot vesznek, és így kevesebb kódot kell majd újraírniuk. Az informális átvizsgálásoknak köszönhetően elkerült hibák eredményeként kevesebb időt igényel majd a tesztelés folyamán fedezett hibák kijavítási folyamata.

17.3. GYORS ALKALMAZÁSFEJLESZTÉS

Bár az agilis módszerek (mint az iteratív fejlesztés egyfajta megközelítései) nagy figyelmet kaptak az elmúlt néhány évben, a vállalati rendszereket már sok éve fejlesztik iteratívan, gyors alkalmazásfejlesztési technikákat használva. A gyors alkalmazásfejlesztési (Rapid Application Development, RAD) technikák az 1980-as évek úgynevezett negyedik generációs nyelveiből fejlődtek ki, és az adatin-tenzív alkalmazások fejlesztésére használták. Következésképpen általában olyan eszközkészletekbe vannak szervezve, amelyek adatok készítésére, kiíratására, keresésére és jelentések készítésére alkalmasak. A 17.9. ábra egy RAD-rendszer tipikus szervezését mutatja be.

A RAD-környezetben található eszközök:

1. *Adatbázis-programozási nyelv*, amely eszközöket biztosít adatbázis-struktúrák használatára és manipulálására. Az SQL (Groff és mások, 2002) az általános



17.9. ábra. Egy gyors alkalmazásfejlesztési környezet

- adatbázis-programozási nyelv. SQL-parancsokat közvetlenül is írhatunk, vagy a végfelhasználók számára űrlapok alapján automatikusan is generálhatunk.
2. *Interfész-generátor*, amely adatbeviteli űrlapok, felületek készítésére és megjelenítésére alkalmas.
 3. *Kapcsolat az irodai szoftverekkel*, mint például egy táblázatkezelővel numerikus adatok elemzésére és kezelésére, vagy egy szövegszerkesztővel a jelentéssablonok készítésére.
 4. *Jelentésgenerátor*, amely arra szolgál, hogy az adatbázisban lévő információkból jelentéseket lehessen előállítani.

A RAD-rendszerek azért sikeresek, mert a vállalati alkalmazások között nagyon sok hasonlóság van, de ezt már a 13. fejezetben kifejtettem. Lényegében ezek az alkalmazások arra szolgálnak, hogy frissítsék az adatbázist, illetve jelentéseket készítsenek az adatbázisban lévő információk alapján. A bemenetre és kimenetre szabványos formátumokat használnak. A RAD-rendszerek ötletét továbbvitték olyan interaktív alkalmazások készítésének irányába, amelyek egy vállalati adatbázis információinak absztrakcióján alapulnak, az adatokat megmutatják a végfelhasználóknak egy terminálon vagy munkaállomáson keresztül, és változtatják az adatbázist a felhasználók kéréseinek megfelelően.

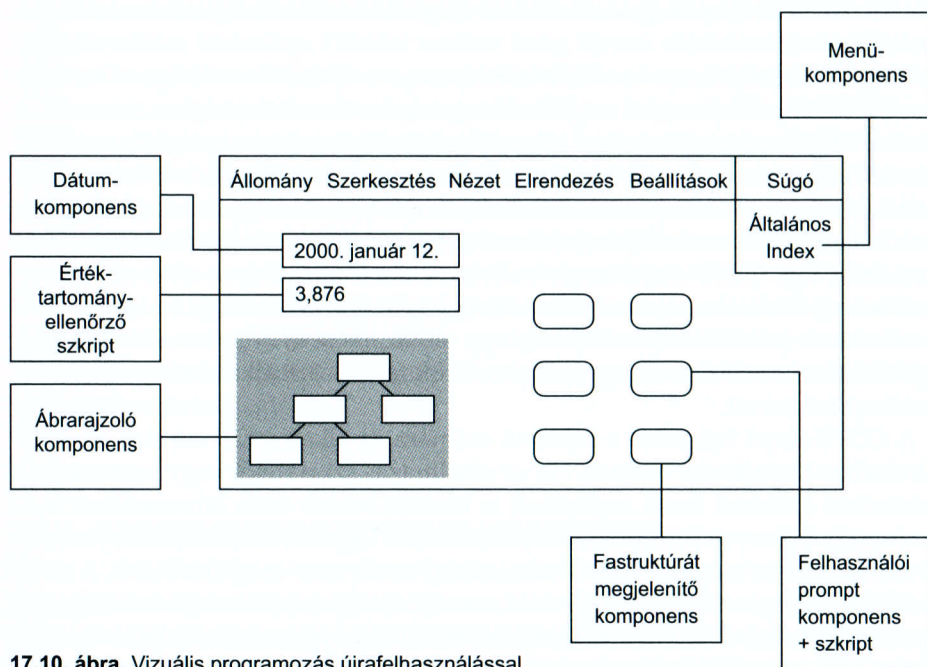
Számos vállalati alkalmazás strukturált bemeneti és kimeneti űrlapokon alapul, ezért a RAD-környezetek komoly lehetőségeket biztosítanak a képernyők definiálására és a jelentések elkészítésére. A képernyők gyakran összekapcsolt űrlapok sorozataként vannak meghatározva (egy alkalommal tanulmányozhatunk egy alkalmazást, amely 137 űrlapdefiníciót tartalmazott), ezért a képernyőgeneráló rendszernek biztosítania kell a következő eszközöket:

1. *Interaktív űrlap-összeállítás*, ahol a fejlesztő definiálja a megjelenítendő mezőket és azt, hogy ezek hogyan helyezkednek el.
2. *Űrlapok összekapcsolása*, ahol a fejlesztő megadhatja, hogy egyes bemenetek hogyan befolyásolják további űrlapok megjelenítését.
3. *Mezőellenőrzés*, ahol a fejlesztő meghatározhatja az űrlap mezőinek értéktartományát.

Most már minden RAD-környezet támogatja a webböngészőkön alapuló adatbázis-felületek fejlesztését. Ezek lehetővé teszik az adatbázis elérését bármilyen internetkapcsolattal rendelkező helyről, csökkentve a képzési és a szoftverköltéseket, és lehetővé téve a külső felhasználóknak az adatbázis elérését. Mindamellet a webböngészőkkel és az internetprotokollokkal járó korlátozásokból az következik, hogy ez a megközelítés nem biztosan felel meg a nagyon gyors, interaktív válaszokat igénylő rendszerek számára.

A legtöbb RAD-rendszer már vizuális programozási eszközöket is tartalmaz, amelyek a rendszer interaktív fejlesztéséhez nyújtanak segítséget. Az alkalmazásfejlesztők interaktívan készítik a rendszert úgy, hogy az interfészt képernyők, mezők, gombok és menük megadásával definiálják. Ezek névvel rendelkeznek és feldolgozószkriptek társulnak az interfész egyes részeihez (például egy gomb neve az, hogy Szimulál). Ezek a szkriptek lehetnek újrafelhasználható komponensek, speciális célú kódok vagy ezek keverékei.

Ezt a megközelítést a 17.10. ábrán illusztráltam, ami az alkalmazás egy képernyőjét mutatja be, ami menüket, bemeneti mezőket (a képernyő bal oldalán lévő fehér mezőket), kimeneti mezőket (a képernyő bal oldalán lévő szürke mezőket) és gombokat (a kerekített sarkú téglalapokat a képernyő jobb oldalán) tartalmaz. Amikor ezeket az entitásokat a vizuális programozórendszerrel elhelyezzük a képernyőn, a fejlesztő definiálja, hogy melyik újrafelhasználható komponenssel kell azt társítani, vagy pedig ír egy programrészletet, ami a szükséges műveleteket végrehajtja. A 17.10. ábra azokat a komponenseket is mutatja, amelyek az egyes képernyőelemekkel társítva vannak.



17.10. ábra. Vizuális programozás újrafelhasználással

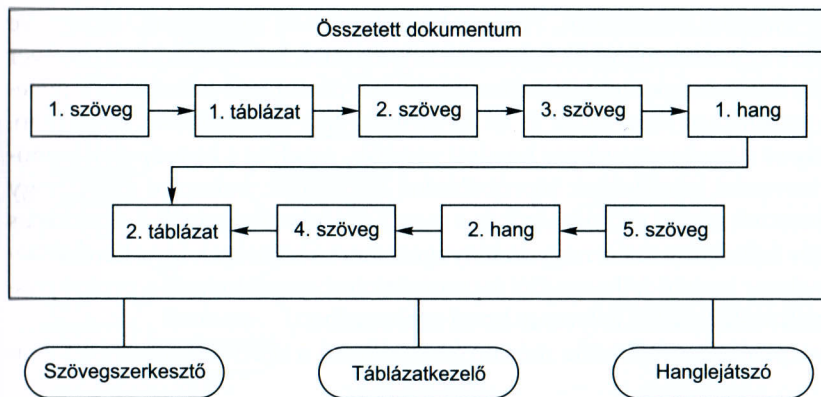
A Visual Basic nagyon kifinomult példája a szkriptnyelveknek (Ousterhout, 1998). A szkriptnyelvek nem típusos, magas szintű nyelvek, amelyeket arra terveztek, hogy komponensek integrálásával készíthessünk rendszereket. Korai szkriptnyelv a Unix shell (Bourne, 1978; Gordon és Bieman, 1995). A kifejlesztése óta sok más komoly szkriptnyelv jelent meg (Ousterhout, 1994; Lutz, 1996; Wall és mások, 1996). A szkriptnyelvek vezérlési szerkezeteket és grafikus eszközrendszereket is tartalmaznak, amelyek, ahogy azt Ousterhout (Ousterhout, 1998) illusztrálta, radikálisan csökkentették a rendszerfejlesztéshez szükséges időt.

A rendszerfejlesztés ezen megközelítése a viszonylag egyszerű alkalmazások gyors fejlesztését tette lehetővé, amelyet kisebb csapatok is el tudnak készíteni. Nagyobb rendszerek esetében, amelyeket nagyobb csapatok fejlesztenek, ez már nehezen szervezhető meg. Nincs explicit rendszer-architektúra és gyakran komplex függőségek léteznek a rendszer egyes részei között, ami nehézségeket okozhat, amikor változtatásokra van szükség. Ráadásul a szkriptnyelvek interakciós objektumok egy csoportjára vannak korlátozva, ezért a nem szabványos felhasználói felületeket nehéz lehet velük implementálni.

A *vizuális fejlesztés* a RAD olyan megközelítése, amely kidolgozott, újrafelhasználható szoftverkomponensekre alapoz. Egy másik újrafelhasználáson alapuló megközelítés olyan „komponenseket” használ fel újra, amelyek önmagukban is teljes alkalmazásrendszert alkotnak. Ezt néha COTS- (Commercial Off-the-Shelf) alapú fejlesztésnek nevezik – itt az alkalmazások már rendelkezésre állnak. Például ha egy rendszernek szüksége van szövegfeldolgozó képességekre, használhatunk egy standard szövegszerkesztő rendszert, mint például a Microsoft Wordöt. A COTS-alapú fejlesztésről mint újrafelhasználhatósági módszerről a 18. fejezetben lesz szó.

Hogy illusztráljam azt az alkalmazástípust, amely esetében érdemes használni a COTS-alapú fejlesztési megközelítést, tekintsük a 7. fejezetben bemutatott követelménykezelési folyamatot. Egy követelménykezelést támogató rendszernek szüksége van olyan eszközre, ami alkalmas a követelmények bevitelére, a követelmények tárolására, jelentések készítésére, a követelmények közötti kapcsolatok felfedezésére és ezen kapcsolatoknak nyomon követhetőségi táblázatban való kezelésére. Egy COTS-alapú megközelítés esetében a prototípus elkészíthető egy adatbázis (a követelmények tárolására), egy szövegszerkesztő (a bemenetek beolvasására és jelentések készítésére), egy táblázatkezelő (a nyomon követhetőségi táblázat kezelésére) és egy, a kapcsolatok feltárására alkalmas speciális kód összekapcsolásával.

A COTS-alapú fejlesztés a fejlesztő számára az alkalmazás minden funkcionalitásához hozzáférést biztosít. Ha az alkalmazás szkriptelő- vagy testreszabási eszközöket (például Excel makrókat) is biztosít, akkor ezek kihasználhatók az alkalmazás bizonyos funkcióinak fejlesztéséhez. Egy dokumentumkompozíciós metafora segíthet megérteni az alkalmazásfejlesztés ezen megközelítését. A rendszer által feldolgozott adatok egy dokumentumkompozícióba szervezhetők, ami úgy működik, mint valami tárolóeszköz különféle objektumok számára. Ezek az objektumok különböző típusú adatokat (táblázat, diagram, űrlap) tartalmazhatnak, amelyek különböző alkalmazásokkal feldolgozhatók. Az objektumok úgy



17.11. ábra. Alkalmazások összekapcsolása

vannak összekapcsolva és típussal ellátva, hogy ha az illető objektumhoz hozzáférés történik, akkor a hozzárendelt alkalmazás elindításra kerül.

A 17.11. ábra egy alkalmazásrendszert mutat be, ami olyan dokumentumkompozíciókból épül fel, amelyek szövegelemeket, táblázatokat és hangállományokat tartalmaznak. A szöveges elemeket a szövegszerkesztő, a táblázatokat a táblázatkezelő, a hangállományokat pedig a hanglejátszó alkalmazás dolgozza fel. Amikor a rendszer használója megpróbál elérni egy bizonyos típusú objektumot, a hozzárendelt alkalmazás meghívásra kerül, hogy a megfelelő felhasználói funkcionalitást biztosítsa. Például amikor hang típusú objektumokat próbálunk elérni, a hanglejátszó hívódik meg annak érdekében, hogy feldolgozza azokat.

Ennek a megközelítésnek a legnagyobb előnye, hogy az alkalmazásunk sok funkcionalitását gyorsan és költségghatékonyan tudjuk implementálni. Azoknak a felhasználóknak, akik már jól ismerik a rendszert alkotó alkalmazásokat, nem kell új dolgokat megtanulniuk. Mindamellett, ha nem ismerik az alkalmazások használatát, annak megtanulása nehézkes lehet, különösen azért, mert összezavarhatja őket az alkalmazások azon funkcionalitása, amely valójában nem is szükséges. Ugyanakkor teljesítménybeli problémákat is eredményezhet ez a megközelítés, mivel több alkalmazás rendszere közötti átváltásra van szükség. Az átváltásokkal járó teljesítménycsökkenés függ az operációs rendszer által biztosított támogatottságtól is.

17.4. SZOFTVERPROTOTÍPUS KÉSZÍTÉSE

Ahogy azt a fejezet bevezető részében már említettem, vannak olyan körülmények, ahol gyakorlati vagy szerződési szempontok miatt az inkrementális szoftverátadási folyamat nem használható. Ezekben a situációkban a rendszerkövetelmények hiánytalan megállapítása szükséges, amit a fejlesztőcsapat a szoft-

verrendszer alapjául használhat. Ahogy korábban már kifejtettem, előnyökre tehetünk szert egy inkrementális fejlesztési folyamattal, ha elkészítjük a szoftver prototípusát. Ezt a megközelítést néha eldobható prototípus-készítésnek nevezik, mert a prototípust nem adják át az ügyfélnek, és a fejlesztő sem tartja fenn.

A prototípus a szoftverrendszer kezdeti verziója, amelyet a koncepciók bemutatására, a tervezési lehetőségek kipróbálására készítenek, valamint azért, hogy többet tudhassunk meg a problémáról és a megoldás lehetőségeiről. A prototípus gyors, iteratív fejlesztése azért nagyon lényeges, mert a költségek így ellenőrizhetők, és a rendszer leendő felhasználói tapasztalatokat szerezhetnek a prototípussal már a szoftverfejlesztési folyamat korai szakaszában.

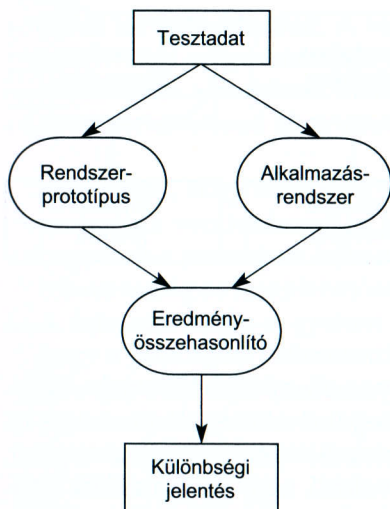
A szoftverprototípus többféle módon használható a szoftverfejlesztési folyamatban:

1. A követelménytervezési folyamatban a prototípus segíthet a követelmények kiderítésében és validálásában.
2. A rendszertervezési folyamatban a prototípus felhasználható bizonyos szoftvermegoldások vizsgálatára és a felhasználói felület megtervezésében.
3. A tesztelési folyamatban a prototípus felhasználható az ügyfél részére átadandó rendszerre visszacsatoló tesztek futtatására.

A rendszerprototípusok a felhasználók számára lehetővé teszik annak megismerését, hogy a rendszer mennyire is segíti a munkájukat. Új ötleteket kaphatnak a követelmények megadásához, és megtalálhatják a szoftver erősségeit és gyengeségeit. Ezután új rendszerkövetelményeket indítványozhatnak. Továbbá a prototípus fejlesztése alatt fény derülhet a javasolt követelményekkel kapcsolatos hibákra és hiányosságokra. A specifikációban leírt funkció akár hasznosnak és jól definiáltnak is tűnhet, de amikor ezt a funkciót más funkciókkal összekombinálják, gyakran arra jönnek rá a felhasználók, hogy az eredeti nézőpontjuk hibás vagy hiányos. A rendszer-specifikáció ezután megváltoztatható annak fényében, hogy a követelményekről kialakult új nézőpontjukat tükrözze.

A rendszer tervezése alatt a rendszerprototípus felhasználható a tervezési tapasztalatok alkalmazására, illetve a javasolt terv megvalósíthatóságának felülvizsgálatára. Például egy adatbázistervből készíthetünk egy prototípust, és tesztelhetjük, leellenőrizhetjük, hogy hatékonyabb adathozzáférést biztosít-e a legáltalánosabb felhasználói kérésekhez. A felhasználói felületek dinamikus természete miatt a szöveges leírások és diagramok nem megfelelőek a felhasználói felülettel szembeni követelmények megfogalmazására. Emiatt a gyors prototípus-készítés a végfelhasználók bevonásával az egyetlen lehetséges mód arra, hogy grafikus felhasználói felületeket fejlesszünk a szoftverrendszerek számára.

A rendszertesztelés jelentős problémája a tesztvalidálás, amikor is ellenőriznünk kell, hogy a teszt eredménye az-e, amire számítottunk. Amikor egy rendszerprototípus rendelkezésre áll, az eredmények ellenőrzéséhez szükséges munka visszacsatoló tesztek futtatásával csökkenthető (17.12. ábra). Ilyenkor ugyanazokra a tesztesetekre futtatjuk mind a prototípust, mind a rendszert. Ha mindkét rendszer ugyanazt az eredményt szolgáltatja, akkor a teszteset valószínűleg nem



17.12. ábra. Visszacsatoló tesztelés

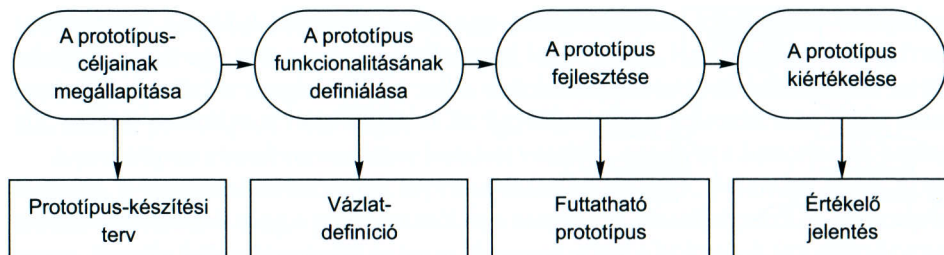
talált hibát. Ha viszont az eredmények különböznek, akkor az rendszerhibára utalhat, és a különbség okát fel kell deríteni.

Végezetül a szoftverfejlesztési folyamatok támogatásán kívül a prototípusok felhasználhatók még a felhasználói dokumentációk elkészítéséhez szükséges idő csökkentésére illetve oktatásra, ahol a felhasználókat ismertetik meg a rendszer használatával. Egy működő (jóllehet korlátolt funkcionalitású) rendszer gyorsan elkészíthető, hogy a vezetés felé demonstrálja az alkalmazás megvalósíthatóságát és hasznavehetőségét.

Egy 39 prototípus-készítési projektről szóló tanulmányban Gordon és Bieman (Gordon és Bieman, 1995) megállapította, hogy a prototípus-készítéssel járó előnyök a következők:

1. fokozott rendszer-használhatóság;
2. pontosabb megfelelés a felhasználók elvárásainak;
3. fokozott tervezési minőség;
4. fokozott karbantarthatóság;
5. kevesebb fejlesztési munka.

A tanulmányuk azt sugallja, hogy a prototípusok használatával járó fokozott használhatóság és jobb felhasználói követelmények nem kötelezően eredményeznek a rendszerfejlesztés kiadásaiban többletköltséget. A prototípus-készítés általában a szoftverfolyamat korai szakaszaiban jelent többletköltséget; a fejlesztési folyamat későbbi részeiben viszont csökkenti a költségeket. Ennek fő oka, hogy a fejlesztés alatt elkerülhetők az újraírási munkák, mert az ügyfelek kevesebb rendszerváltoztatást fognak javasolni. Mindamellett Gordon és Bieman azt is megállapította, hogy az átfogó rendszerteljesítmény néha gyengébb lesz, ha nem igazán hatékony prototípuskód kerül újrafelhasználásra.



17.13. ábra. A prototípusfejlesztés folyamata

A 17.13. ábrán a prototípusfejlesztés egy folyamatmodelljét láthatjuk. A prototípus-készítés céljait már a folyamat elején meg kell adni explicit módon. Ez lehet egy olyan rendszer fejlesztése, ami felhasználói felületeket prototipizál, esetleg funkcionális rendszerkövetelményeket validál, vagy alkalmazások megvalósíthatóságát demonstrálja a vezetés számára. Ugyanaz a prototípus nem fel lehet meg mindegyik célnak. Amennyiben a célok előre nem meghatározottak, a vezetés vagy a végfelhasználók félreérthetik a prototípus funkcióját. Következésképpen nem biztos, hogy a prototípus-fejlesztéstől várt előnyökhöz hozzájutnak.

A folyamat következő szakasza annak eldöntése, hogy mit vegyünk bele és, talán még ennél is fontosabb, hogy mit hagyjunk ki a rendszer prototípusából. Néhány funkcionáltságot kihagyhatunk a prototípusból annak érdekében, hogy csökkentsük a prototípus-készítés költségeit, illetve felgyorsítsuk az átadás ütemét. Úgy is dönthetünk, hogy enyhítjük a nemfunkcionális követelményeket, mint például a válaszütem és a memóriahasználatot. A hibakezelés figyelmen kívül hagyható vagy kezdetleges is lehet, kivéve azt az esetet, amikor a prototípus célja egy felhasználói felület létrehozása. A megbízhatóság és a program minőségének szintje akár csökkentett is lehet.

A folyamat végső fázisa a prototípus kiértékelése. Ebben a szakaszban gondoskodni kell a felhasználók képzéséről és a prototípus céljai alapján készíteni kell egy kiértékelési tervet. A felhasználóknak időre van szükségük egy új rendszer megismeréséhez, illetve annak rendeltetésszerű használatához. Amint a rendszert normálisan használni tudják, valószínűleg felfedeznek néhány, a követelményeikkel kapcsolatos hibát és hiányosságot.

Futtatható és eldobható prototípus fejlesztésekor általános probléma, hogy a prototípus használati módja nem egyezik meg azzal, ahogyan majd az átadott rendszert kell használni. Nem biztos, hogy a prototípus tesztelője egy tipikus rendszerfelhasználó, továbbá nem biztos, hogy a prototípus kiértékelése során a képzési idő elegendő lesz. Ha a prototípus lassú, a kiértékelők úgy módosíthatják a munkájukat, hogy kihagyhatják azokat a rendszerfunkciókat, amelyeknek lassú a válaszüteme. Amikor viszont a végső rendszerben ez már gyorsabb válaszütemmel működik, elképzelhető, hogy már másképp fogják azt használni.

A vezetőség gyakran nagy nyomást gyakorol a fejlesztőkre, hogy eldobható prototípusokat készítsenek, kiváltképp amikor a szoftver végső verziójának át-

adásánál késések adódnak. A vezetés sokkal inkább hisz abban, hogy egy befejezetlen vagy rossz minőségű rendszer jobb, mint a semmi, mintsem abban, hogy késhetnek a projekttel. Mindamellet ez a legtöbbször nem túl bölcs döntés, mégpedig a következő okok miatt:

1. Nem biztos, hogy lehetséges úgy beállítani a prototípust, hogy az megfeleljen azoknak a nemfunkcionális követelményeknek, amelyeket figyelmen kívül hagytunk a prototípus fejlesztésekor, mint például a teljesítmény, biztonság, robusztusság és megbízhatóság.
2. A fejlesztés során a gyakori változtatások elkerülhetetlenül oda vezetnek, hogy a prototípus dokumentálatlanná válik. Az egyetlen tervezési specifikáció a prototípus kódja. Ez nem elég a hosszú távú karbantartáshoz.
3. A prototípus fejlesztése alatt történő változtatások valószínűleg rontani fogják a rendszer struktúráját. Ezért a rendszer fenntartása nehézkessé és költségessé válhat.
4. A szervezeti minőségi szabványok a prototípus fejlesztésénél általában elhanyagoltak.

Az eldobható prototípusok nem szükségképpen futtathatók vagy használhatók a követelménytervezés folyamata alatt. Ahogy azt a 16. fejezetben írtam, a papírra vetett felhasználói felület makettek (Rettig, 1994) segíthetnek a felhasználóknak abban, hogy finomítsák a rendszer felhasználói felületének tervét, és átdolgozzák a működési forgatókönyveket. Ezek nagyon olcsón kifejleszthetők és pár nap alatt megépíthetők. Ennek a technikának kibővített változata az „Óz, a nagy varázsló” prototípus, ami csak a felhasználói felület fejlesztésével foglalkozik. A felhasználók ezzel a felülettel ismerkednek, de a kéréseik egy olyan személyhez jutnak először, aki feldolgozza ezeket, és ez alapján készíti el a megfelelő visszajelzéseket.

Kulcsfogalmak

- Ahogyan nő a nyomás a szoftver gyors átadására, egyre inkább használják az iteratív szoftverfejlesztést, mint szabványos fejlesztési technikát a kis- és közepeméretű rendszerek számára, különösképp az üzleti életben.
- Az agilis módszerek olyan iteratív fejlesztési módszerek, amelyek az inkrementális specifikációra, tervezésre és rendszer-implementálásra koncentrálnak. A fejlesztési folyamatba közvetlenül bevonják az ügyfelet is. A fejlesztési többletmunkák csökkentésével gyorsabb szoftverfejlesztést tesznek lehetővé.
- Az extrém programozás jól ismert agilis módszer, amely egy sor hasznos programozási gyakorlatot integrál, ilyen például a szisztematikus tesztelés, a folyamatos szoftvertökéletesítés és az ügyfél részvétele a fejlesztőcsapatban.
- Az extrém programozás egy különleges erőssége az automatikus tesztfejlesztés, még a programfunkció megírása előtt. Amikor egy új inkremenst integrálunk a rendszerbe, akkor minden tesztnek sikeresen le kell futnia.

- A gyors alkalmazásfejlesztés magában foglalja az olyan fejlesztői környezetek használatát, amelyek hatékony eszközöket biztosítanak a rendszer elkészítéséhez. Ezek között szerepelnek adatbázis-programozási nyelvek, űrlap- és jelentésgenerátorok, illetve kapcsolódási lehetőségek az irodai alkalmazásokhoz.
- Az eldobható prototípus-készítés egy iteratív fejlesztési folyamat, ahol a prototípust arra használjuk, hogy megvizsgáljuk a követelményeket és a tervezési lehetőségeket. Ez a prototípus nem arra készül, hogy a rendszer megrendelője üzembe helyezhesse.
- Amikor egy eldobható prototípust implementálunk, akkor a rendszer azon részeit fejlesztjük ki először, amelyeket a legkevésbé értünk. Ellentétben az inkrementális fejlesztési megközelítéssel, ahol éppen azokkal a részekkel kezdjük a rendszer fejlesztését, amelyeket a legjobban értünk.

További irodalom

Extreme Programming Explained. Ez volt az első XP-vel foglalkozó könyv, és még talán most is a leginkább olvasható. A könyv ezt a megközelítést az egyik megalkotó szemszögéből mutatja be, és a lelkesedése tisztán érezhető. (Beck, K., 2000, Addison-Wesley.)

„Get ready for agile methods, with care”. Átfogó kritika az agilis módszerekről, azok erősségeiről és gyengeségeiről, mérhetetlenül tapasztalt szoftvertervező szemszögéből. (Boehm, B., *IEEE Computer*, January 2002.)

„Scripting: Higher-level programming for the 21st century”. A szkriptnyelvek áttekintése a Tcl/Tk megalkotójától, aki a gyors alkalmazásfejlesztés ezen megközelítésének az előnyeit fejtegeti. (Ousterhout, J. K., *IEEE Computer*, March 1998.)

DSDM: Dynamic Systems Development Method. Egy gyors alkalmazásfejlesztési megközelítés leírása, amit néhányan egy agilis módszer korai megjelenésének tartanak. (Stapleton, J., 1997, Addison-Wesley.)

Feladatok

- 17.1. Magyarázza meg, hogy az új rendszerek gyors átadása gyakran miért fontosabb az üzletben, mint ezen rendszerek teljes funkcionalitása.
- 17.2. Magyarázza meg, hogy az agilis módszerek mögött húzóódó alapelvek hogyan vezetnek a szoftver gyorsabb fejlesztéséhez és átadásához.
- 17.3. Mikor nem ajánlott az agilis módszerek használata egy szoftverrendszer fejlesztésénél?
- 17.4. Az extrém programozás a felhasználói követelményeket mint történeteket írja le, minden egyes történetet egy-egy külön kártyára. Mutassa be a követelményleírás ezen módszerének az előnyeit és hátrányait.

- 17.5. Magyarázza meg, hogy az előrehozott tesztelést alkalmazó fejlesztés miért segít a programozónak jobban megérteni a rendszerkövetelményeket. Melyek a lehetséges nehézségek?
- 17.6. Adjon meg négy indokot arra, hogy nagyjából miért azonos a termelési rátája a párban dolgozó programozóknak és két külön dolgozó programozónak.
- 17.7. A munkahelye megbízza, hogy vizsgálja meg a szoftverfejlesztési folyamatban a prototípus-készítés megvalósíthatóságát. Készítsen jelentést a vezetőségnek, amelyben leírja azokat a projektípusokat, amelyek esetében érdemes prototípust készíteni, illetve megállapítja az ezzel járó költségeket és előnyöket.
- 17.8. Egy szoftvermenedzsert bevonnak egy olyan szoftvertervezést támogató rendszer fejlesztésébe, amely a szoftverkövetelmények formális szoftverspecifikációvá történő átalakítását teszi lehetővé. Magyarázzuk meg a következő fejlesztési stratégiák előnyeit és hátrányait:
- a) Készítsünk egy eldobható prototípust, értékeljük ki, és azután nézzük át újra a rendszerkövetelményeket. Ezután fejlesszük ki a végső rendszert C nyelven.
 - b) Fejlesszük ki a rendszert a már meglévő követelményeknek megfelelően Java nyelven és aztán alakítsuk át úgy, hogy az bármilyen felhasználói követelményváltoztatáshoz megfelelően alkalmazkodni tudjon.
 - c) Fejlesszük ki inkrementális fejlesztést alkalmazva a rendszert úgy, hogy egy felhasználót bevonunk a fejlesztőcsapatba.
- 17.9. Egy jótékonyági egyesület megkért, hogy készítsünk el számukra egy prototípusrendszert, ami nyomon követi a kapott adományokat. Ennek a rendszernek nyilván kell tartania az adományozók nevét, címét, az adomány mértékét és az adományozás dátumát. Ha egy adomány egy bizonyos mértéknél nagyobb, akkor az adományozó feltételeket szabhat (például csak bizonyos célokra lehet azt hasznosítani), és a rendszernek ezeket is számon kell tartania, illetve még azt is, hogy végül mire használták fel az adományt. Magyarázza meg, hogyan készítené el ezt a rendszert, szem előtt tartva azt is, hogy a jótékonyági egyesületnek van bizonyos számú fizetett munkása és önkéntese is. Az önkéntesek többsége nyugdíjas, akiknek nincs, vagy csak kevés számítógépes tapasztalata van.
- 17.10. Egy ügyfél számára kifejlesztett egy eldobható prototípusrendszert, aki nagyon elégedett vele. Azonban az ügyfél azt javasolja, hogy nincs szükség másik rendszerre, hanem a prototípust kellene csak átadni, és ő felajánlja, hogy kiemelkedő összeget fizet a rendszerért. Tudja, hogy a jövőben problémák adódhatnak a rendszer karbantartásával. Miként válaszolna ennek az ügyfélnek?