MISKOLCI EGYETEM

GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR

Szoftverfejlesztés

(MSc)

KÉSZÍTETTE: DR. MILEFF PÉTER

Miskolci Egyetem
Általános Informatikai Tanszék

2010

Tartalomjegyzék

1. A szoftver

A *szoftver* szót sokan egyenlőnek tekintik a számítógépes programokkal. Valójában ez túlságosan szigorú nézet. A szoftvert nem csak maguk a programok alkotják, hanem a hozzájuk kapcsolódó dokumentációk, illetve konfigurációs adatok, amelyek elengedhetetlenek ahhoz, hogy ezek a programok helyesen működjenek.

A szoftverterméknek két nagyobb csoportja létezik:

1. ***Általános termékek:*** ezek az egyedülálló rendszerek, amelyeket egy fejlesztő szervezet készít és ad el a piacon bármely vevőnek, aki azt képes megvásárolni. Gyakran úgy hivatkoznak ezekre, mint dobozos szoftverekre. Pl.: adatbázis-kezelők, szövegszerkesztők, a rajzoló csomagok és projektmenedzselési eszközök.

2. ***Egyedi igényekre szabott (rendelésre készített) termékek:*** az ilyen rendszerek egyéni megrendelők megbízásai alapján készülnek speciális megrendelői igények alapján. Pl.: az elektromos eszközök vezérlőrendszerei, a forgalomirányító és ellenőrző rendszerek.

A határ a két típus között gyakran elmosódhat. A következőkben megvizsgáljuk a **szoftverfolyamatot**.

A szoftverfolyamat tevékenységek és kapcsolódó eredmények sora, amelyek egy szoftvertermék előállításához vezetnek. A szoftverfolyamatok összetettek és nagyban függenek az emberi tevékenységektől és döntésektől. Emiatt a folyamatok automatizálására történő erőfeszítések segítségével csak korlátozott sikerek érhetők el. A CASE (számítógéppel segített szoftvertervezés) eszközök képesek a folyamatok bizonyos tevékenységeinek támogatására, de nincs lehetőség nagyobb mértékű automatizációra. Nincs ideális, minden számára megfelelő folyamat, különböző szervezetek a szoftverfejlesztést homlokegyenest különböző nézőpontokból közelítik meg. A folyamatokat úgy alakítják ki, hogy kiaknázzák a szervezeten belül az emberek különféle képességeit és a fejlesztő rendszer jellegzetességeit.

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

1. **Szoftverspecifikáció:** a szoftver funkcióit, illetve annak megszorításait definiálni kell.
2. **Szoftvertervezés és implementáció:** a specifikációnak megfelelő szoftvert készítjük el.
3. **Szoftvervalidáció:** ellenőrzési fázis, ahol a szoftvert validálni kell, hogy biztosítsuk, azt fejlesztettük, amit az ügyfél kíván.
4. **Szoftverevolúció:** a szoftvert úgy kell kialakítani, hogy megfeleljen a megrendelő kívánsága szerint történő változtatásoknak.

Habár nincs „ideális” szoftverfolyamat, számos terület van, ahol a szervezeten belüli szoftverfolyamatokon javíthatunk, mert gyakran elavult technikákat tartalmaznak, vagy esetleg nem is élnek a tervezési módszerek lehetőségeivel.

2. A szoftverfolyamat modelljei

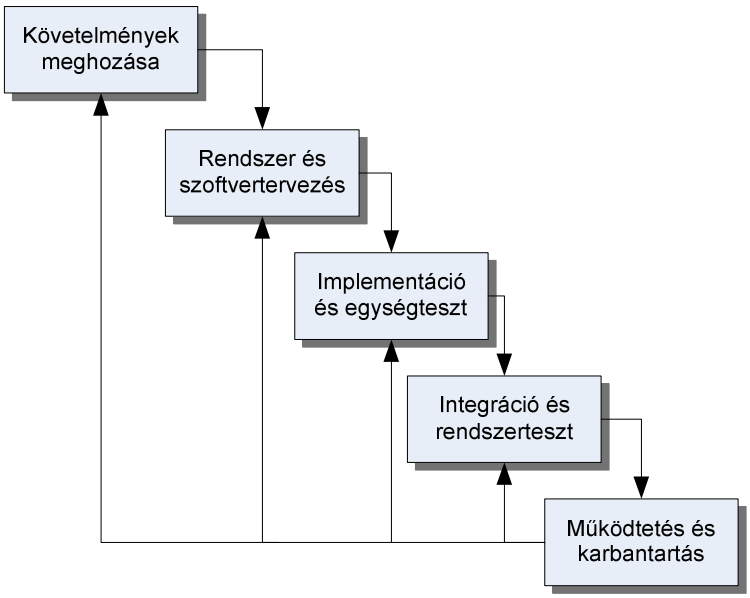
A szoftverfolyamat modellje a szoftverfolyamat absztrakt reprezentációja. Minden egyes modell különböző speciális perspektívából reprezentál egy folyamatot, de ily módon csak részleges információval szolgálhat magáról a folyamatról. Ezek az általános modellek nem a szoftverfolyamat pontos, végleges leírásai. Sokkalta inkább hasznos absztrakciók, amelyet a szoftverfejlesztés különböző megközelítési módjainak megértéséhez használhatunk. Az irodalomban legismertebb folyamatmodellek a következők:

1. **A vízesésmodell.** Ez a folyamat alapvető tevékenységeit a folyamat különálló fázisainak tekinti. Ezek a fázisok a követelményspecifikáció, a szoftvertervezés, az implementáció, a tesztelés stb.
2. **Evolúciós vagy iteratív fejlesztés.** Ez a megközelítési mód összefésüli a specifikáció, a fejlesztés és a validáció tevékenységeit.
3. **Komponens alapú fejlesztés.** Ez a megközelítés nagy mennyiségű újrafelhasználható komponensek létezésén alapszik.

Ezen modelleknek nem kizárólagos a használatuk a gyakorlatban, gyakran keverednek is egymással, főként nagy rendszerek fejlesztésekor.

2.1 A vízesésmodell

A szoftverfejlesztés folyamatának első publikált modellje, amely más tervezői modellekből származik. Az elnevezése onnan fakad, hogy a modellben az egyes fázisok lépcsősen kapcsolódnak egymáshoz, ami alapján vízesésmodellként vált ismertté. Ezt illusztrálja az 1. ábra.



1. ábra. A szoftver életciklusa

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

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

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

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

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

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

A fázisok eredménye tulajdonképpen egy **dokumentum**. A következő fázis addig nem indulhat el, amíg az előző fázis be nem fejeződött. A gyakorlatban persze ezek a szakaszok átfedhetik egymást. A szoftverfolyamat tehát **nem egyszerű lineáris modell**, hanem a fejlesztési tevékenységek iterációjának sorozata.

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

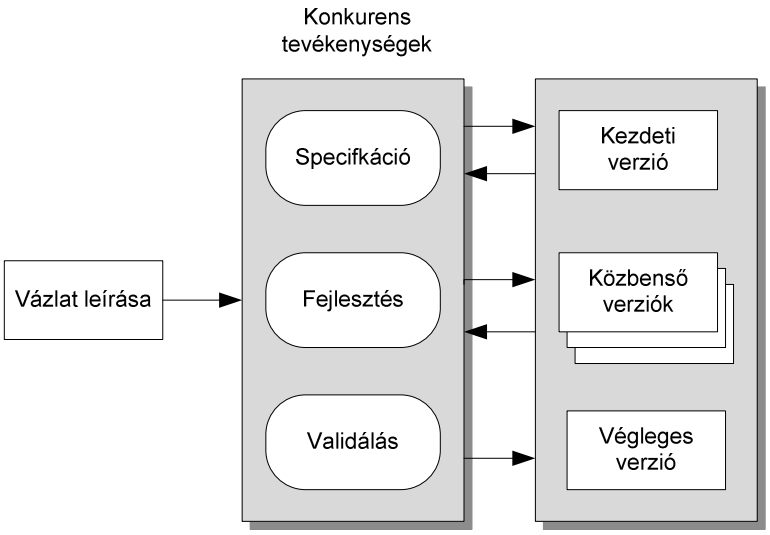
Az életciklus utolsó szakaszában a felhasználók használatba veszik a szoftvert. Ilyenkor derülnek ki az eredeti rendszerkövetelmények hibái és hiányosságai. Program és tervezési hibák kerülnek elő, továbbá felmerülhet új funkciók szükségessége is. Ezekből adódóan a rendszert át kell alakítani, amely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

Hátrányok:

A vízesésmodell problémáját a projekt szakaszainak különálló részekké történő nem flexibilis partícionálása okozza. A vízesésmodell csak akkor használható jól, ha már előre jól ismerjük a követelményeket.

2.2 Evolúciós fejlesztés

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



2. ábra. Evolúciós fejlesztés

Az evolúciós fejlesztésnek két különböző típusa ismert:

1. **Feltáró fejlesztés:** Célja az, hogy a megrendelővel együtt tárjuk fel a követelményeket, és alakítsuk ki a végleges rendszert. A fejlesztés a rendszer már ismert részeivel kezdődik. A végleges rendszer úgy alakul ki, hogy egyre több, az ügyfél által kért tulajdonságot társítunk a már meglévőkhöz.

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

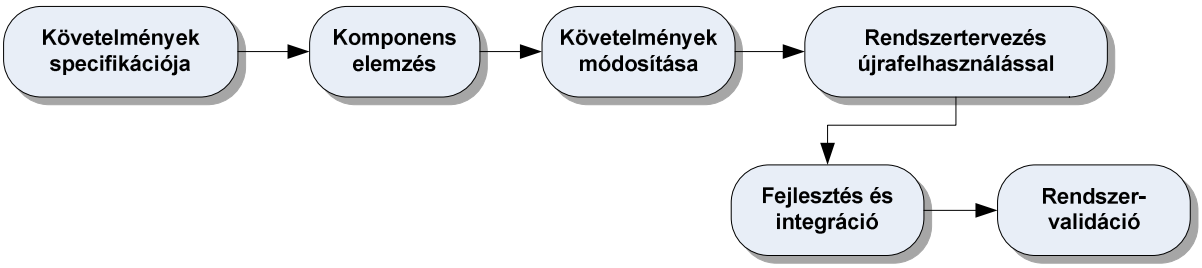
Az evolúciós megközelítés hatékonyabb a vízesésmodellnél, ha olyan rendszert kell fejleszteni, amely közvetlenül megfelel az ügyfél kívánságainak. További előnye, hogy a rendszerspecifikáció inkrementálisan fejleszthető. Mindazonáltal a vezetőség szemszögéből két probléma merülhet fel:

1. **A folyamat nem látható:** a menedzsernek rendszeresen szükségük van leszállítható részeredményekre, hogy mérhessék a fejlődést.
2. **A rendszerek gyakran szegényesen strukturáltak.** A folyamatos változtatások lerontják a rendszer struktúráját. A szoftver változásainak összevonása pedig egyre nehezebbé és költségesebbé válhat.

Kinek érdemes használni az evolúciós fejlesztési modell? Várhatóan rövid élettartalmú kis vagy közepes rendszerek esetén (kb. 500000 programsorig) célszerű az alkalmazása. Nagy, hosszú élettartalmú rendszerek esetén az evolúciós fejlesztés válságossá válhat.

2.3 Komponens alapú fejlesztés

A komponens alapú fejlesztés alapgondolata az újrafelhasználható komponensekből való építkezés. A szoftverfolyamatok többségében megtalálható valamelyest a szoftverek újrafelhasználása. Ilyen esetekben előkeresik a korábbi kódot (komponenst) és újra átdolgozva, esetleg általánosítva beledolgozzák a rendszerbe.



3. ábra. Újrafelhasználás orientált modell

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

1. **Komponenselemzés.** Egy adott követelményspecifikáció mellett keresést kell végrehajtani, hogy mely komponensek implementálták azokat. A legtöbb esetben nincs egzakt illeszkedés, a felhasznált komponens a funkciók 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 nem lehetséges, ott újra el kell végezni a komponenselemzést és alternatív megoldást kell keresni.

3. **Rendszertervezés újrafelhasználással.** Ez a szakasz felelős a rendszer szerkezetének tervezéséért. A tervezésben számon kell venni, hogy milyen komponenseket akarnak újrafelhasználni, és úgy alakítani a szerkezetet, hogy azok működhessenek. Amennyiben nincs elérhető újrafelhasználható komponens, akkor új szoftverek is kifejleszthetők.

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

A komponens alapú modell **előnye**, hogy csökkenti a kifejlesztendő szoftverek számát, így csökkentve a költségeket, illetve kockázati tényezőket. Sok esetben a rendszer így gyorsabban leszállítható. **Hátrány**: a követelményeknél elkerülhetetlenek a kompromisszumok, amelyek oda vezetnek, hogy a rendszer nem felel meg a felhasználó valódi kívánságainak.

3. Folyamatiteráció

A szoftverfolyamat nem egy egyszerű folyamat, hanem sokkal inkább a folyamattevékenységek rendszeresen ismétlődő folyamata, amely hatására a rendszert mindig átdolgozzuk az igényelt változások szerint. Ennek oka, hogy a nagy rendszerek esetében a változtatások elkerülhetetlenek, mert változhatnak a követelmények, az üzletmenet, és a külső behatások.

A folyamatiteráció támogatására több modell megjelent. A két legismertebb ezekből:

1. **Inkrementális fejlesztés.** Ekkor a szoftverspecifikáció, a tervezés, az implementálás, kis inkrementációs lépésekre van felosztva.
2. **Spirális fejlesztés.** Ekkor a rendszer fejlesztése egy belülről kifelé tartó spirálvonalat követ.

Az iteratív folyamat lényege, hogy a specifikációt a szoftverrel összekapcsolva kell fejleszteni.

3.3 Inkrementális fejlesztés

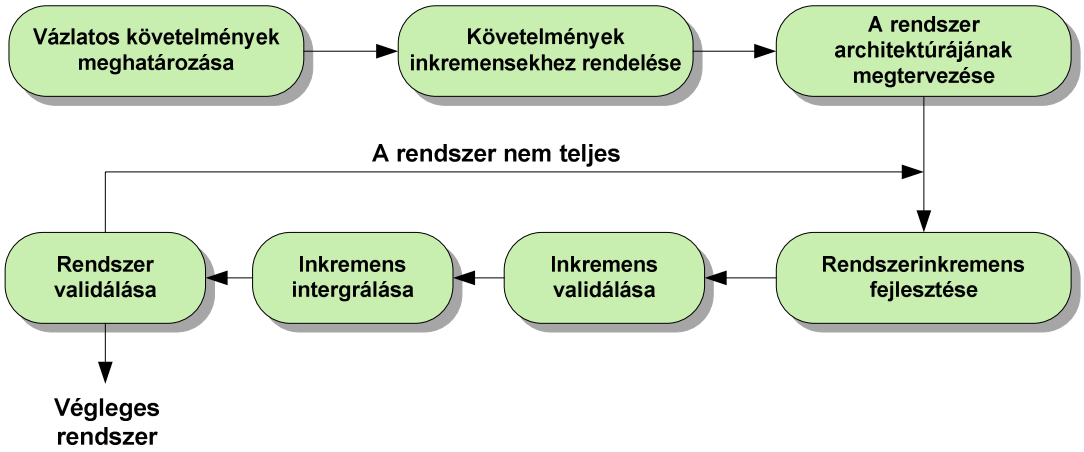
Az inkrementális megközelítési mód egy köztes megközelítés a vízesésmodell és az evolúciós fejlesztési modellek között. A vízesésmodell megköveteli az ügyféltől, hogy véglegesítse a követelményeket mielőtt a tervezés elindulna, a tervezőtől pedig azt, hogy válasszon ki bizonyos tervezési stratégiákat az implementáció előtt. A vízesésmodell előnye, hogy egyszerűen menedzselhető, mert külön választja az egyes fázisokat. Ezzel szemben viszont olyan robosztus rendszerek jöhetnek létre, amik esetleg alkalmatlanok a

változtatásokra. Az evolúciós megközelítésnél pedig megengedettek a követelményekkel és tervezésekkel kapcsolatos döntések elhagyása, ami pedig gyengén strukturált és nehezen megérthető rendszerekhez vezethetnek. A módszer lépései a 4. ábrán figyelhetők meg.

Egy inkrementális fejlesztési folyamatban a megrendelő meghatározza:

- nagy körvonalakban a rendszer által nyújtandó szolgáltatásokat,
- mely szolgáltatások fontosabban, melyek kevésbé.

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



4. ábra. Inkrementális fejlesztés

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

Amennyiben egy inkremens elkészült, a rendszer bizonyos funkcióit akár be is üzemeltethetik korábban. Így tapasztalatokat szerezhetnek a rendszerrel kapcsolatban, amely a későbbi inkrementációs lépésekben segítségre lehet a követelmények tisztázásában. Amennyiben az új inkremens elkészül, azt integrálni kell a már meglévő inkremensekkel. Ezzel a rendszerfunkciók köre egyre bővül. Az inkrementációs fejlesztés előnyei:

1. A megrendelőnek nem kell megvárnia míg a teljes rendszer elkészül. Már az első inkremens kielégítheti a legkritikusabb követelményeket, így a szoftver már menet közben használhatóvá válik.
2. A megrendelők használhatják a korábbi inkremenseket mint prototípusokat, ami által tapasztalatokat szerezhetnek.
3. Kisebb a kockázata annak, hogy a teljes projekt kudarcba fullad.
4. A fejlesztés során a magasabb prioritású inkremenseket szállítjuk le hamarabb, ezért mindig a legfontosabb szolgáltatások lesznek többet tesztelve. Ezért kisebb a hiba esélye a rendszer legfontosabb részeiben.

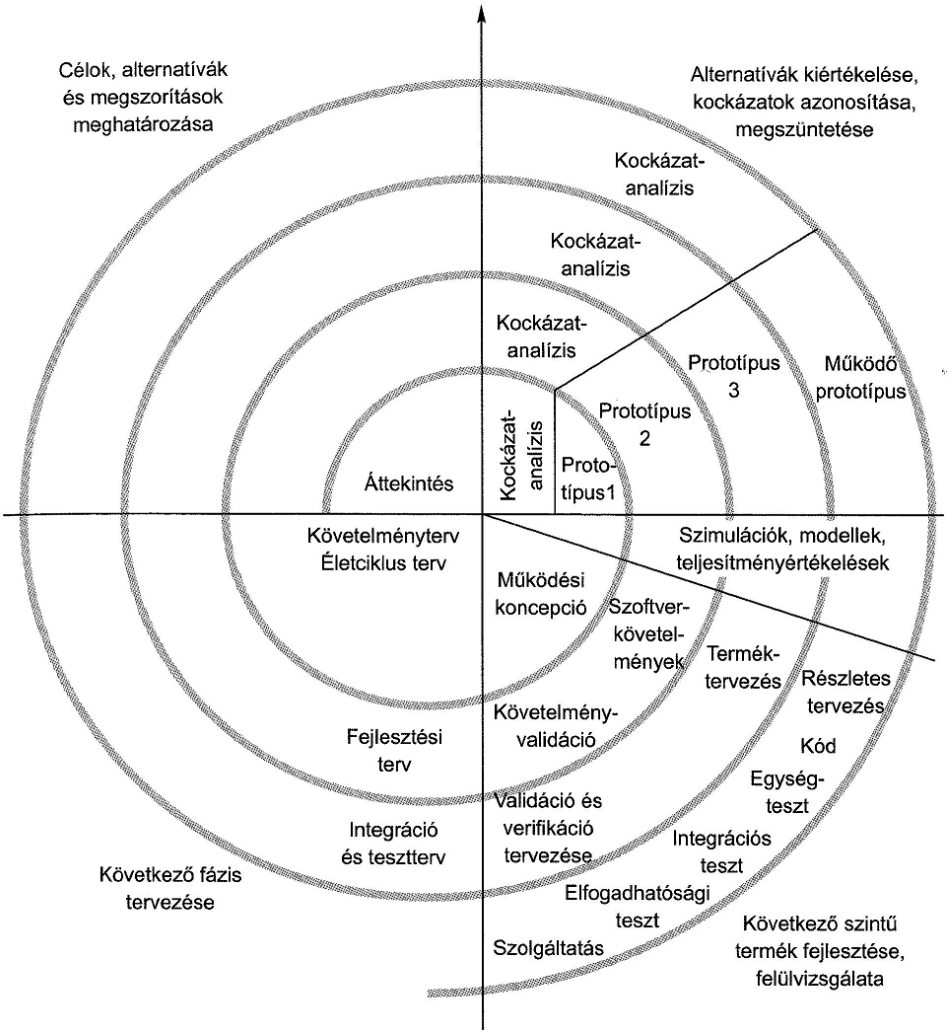
Mindezek ellenére az inkrementális fejlesztésnek is megvannak a maga hibái. Az inkremenseknek megfelelően kis méretűeknek kell lenni és minden inkrementációs lépésnek

szolgáltatni kell valami rendszerfunkciót. Így nehézkessé válhat a megrendelő követelményeit megfelelő méretű inkrementációs lépésekre bontani.

3.4 Spirális fejlesztés

A spirális fejlesztési modellt Boehm javasolta először már 1988-ban, amely azóta széles körben elterjedt az irodalomban és a gyakorlatban. **A szoftverfolyamatot** nem tevékenységek és közöttük található esetleg visszalépések sorozataként tekinti, hanem inkább **egy spirálként reprezentálja**. A spirál minden egyes körben a szoftverfolyamat egy-egy fázisát reprezentálja.

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



5. ábra. Boehm féle spirálmodell [1]

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 ha lehetséges.
2. **Kockázat becslése:** Minden egyes felismert kockázati tényező esetén részletes elemzésre kerül sor. Lépéseket kell tenni a kockázat csökkentése érdekében.
3. **Fejlesztés és validálás:** A kockázat kiértékelése után egy fejlesztési modellt kell választani a problémának megfelelően. Pl. evolúciós, vízesés, stb modellek.
4. **Tervezés:** A folyamat azon fázisa, amikor dönteni kell arról, hogy folytatódjon-e egy következő ciklussal, vagy sem. Ha a folytatás mellett döntünk, akkor fel kell vázolni a projekt következő fázisát.

Miben más a spirális fejlesztési modell az egyéb szoftverfolyamat-modelltől? Itt a modell explicite számol a kockázati tényezőkkel, amelyek problémákat okozhatnak a projektben. Ilyen például a határidő- és költségtúllépések.

Egy spirálciklus a célok meghatározásával kezdődik. Fel kell sorolni a megvalósítás lehetőségeit, és hozzá kell venni azok megszorításait is. Minden egyes célhoz meg kell határozni egy lehetséges alternatívát, amely azt eredményezi, hogy azonosításra kerülnek a projekt kockázati forrásai is. A következő lépés ezeknek a kockázatoknak a kiértékelése, majd pedig a tervezési fázis következik.

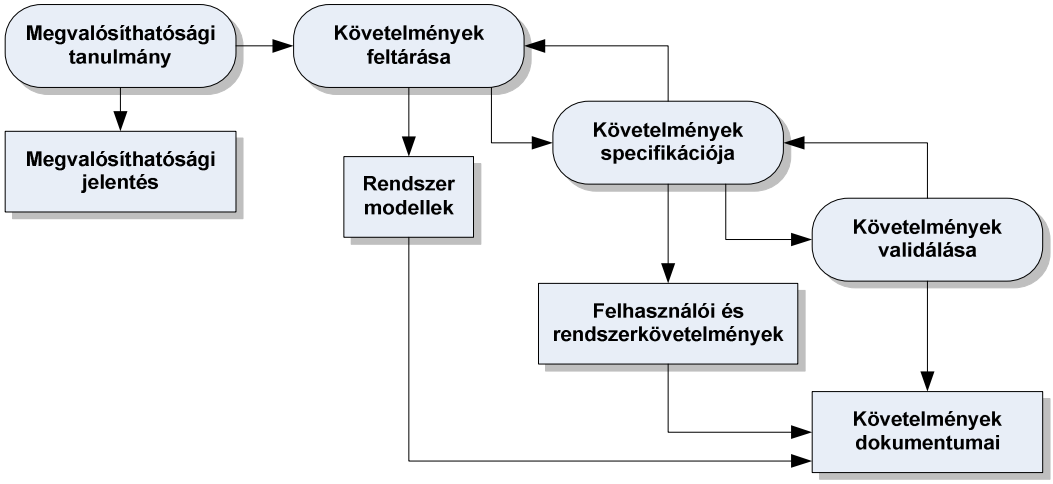
4. Folyamattevékenységek

Alapvetően négy különböző folyamattevékenységet szokás elkülöníteni: **specifikáció, tervezés és implementáció, validáció és evolúció**. Ezeket a különféle fejlesztési folyamatok különféleképpen szokták szervezni. A vízesésmodell esetében ezek egy szekvenciába szerveződnek, míg az evolúciós fejlesztésnél összefésülődnek. A továbbiakban ezeket a fázisokat vizsgáljuk meg.

4.1 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. Az ebben a szakaszban vétett hibák elkerülhetetlenül problémákhoz vezetnek majd a rendszertervezés későbbi szakaszában és az implementációban.

A követelmények tervezési folyamatát a következő ábra mutatja. A folyamat eredménye a követelmény dokumentum előállítása, amely a rendszer specifikációja. A követelmények két különböző szinten kerülnek kifejtésre a dokumentumban: a végfelhasználók és ügyfelek számára leírt követelmények, és a fejlesztők sokkal részletesebb rendszer-specifikációja.



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 rendszer költséghatékony-e, és hogy az kivitelezhető-e. A megvalósíthatóság elemzésének relatíve olcsónak és gyorsnak kell lennie. Eredménye a megvalósíthatósági jelentés.

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

3. ***Követelmény specifikáció***. *A* követelményspecifikáció az elemzési tevékenységek során összegyűjtött információk egységes dokumentummá alakítása. A dokumentumnak a követelmények két típusát kell tartalmaznia:

    - **A felhasználói követelmények** a rendszerkövetelmények absztrakt leírása, amelyek a végfelhasználóknak, illetve a megrendelőknek szólnak.
    - **A konkrét rendszerkövetelmények**, amelyek részletezik az elkészítendő rendszer által nyújtandó funkciókat.

4. ***Követelmény-validáció.*** A tevékenység ellenőrzi, hogy mennyire valószerűek, konzisztensek és teljesek a követelmények. A folyamat során fel kell tárni a követelmények dokumentumában található hibákat, és kijavítani.

Természetesen a követelménytervezés tevékenységeit nem kötelező szigorú sorrendben végrehajtani. Ekkor az elemzés, a meghatározás és a specifikáció tevékenységei összefésülhetők, egymást átfedhetik a folyamatban.

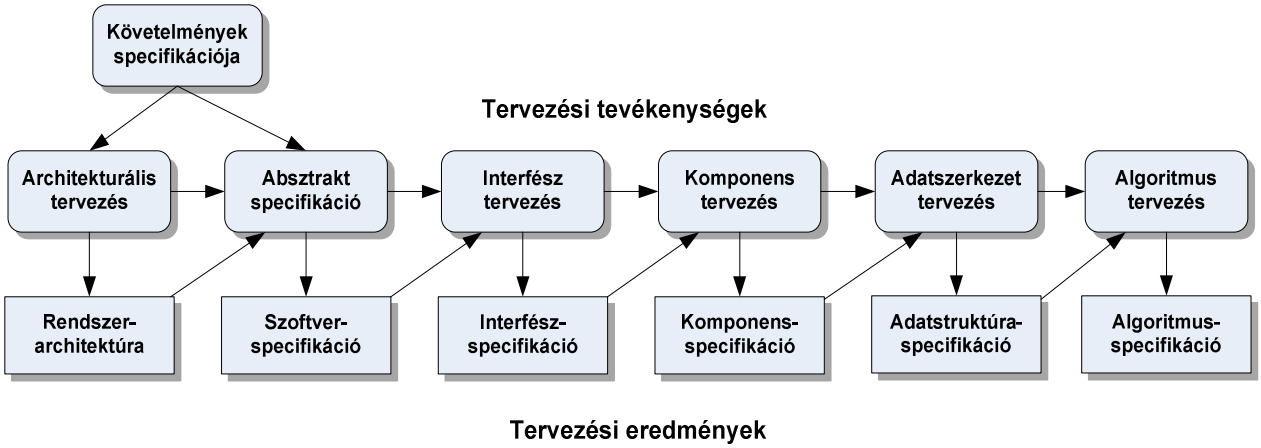
4.2 Szoftvertervezés és implementáció

A szoftverfejlesztés implementációs szakasza nem más, mint a rendszer-specifikáció

futtatható rendszerré történő alakítása. Ez mindig magában foglalja a szoftver tervezését és a programozást, esetleg a specifikáció finomítását.

A szoftver **tervezése** az implementálandó szoftver struktúrájának és az adatoknak, 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 tervezési folyamat tevékenységei átfedik egymást.



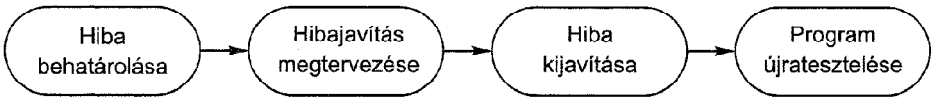
7. ábra. A tervezési folyamat általános modellje

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. ***Architekturális tervezés***. A rendszert felépítő alrendszereket és a köztük található kapcsolatokat azonosítani és dokumentálni kell.
2. ***Absztrakt specifikáció***. Minden egyes alrendszer esetében meg kell adni szolgáltatásaik absztrakt specifikációját és azokat a megszorításokat, 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 ismernénk a működését.
4. ***Komponens tervezése***. *A* szolgáltatások elhelyezése 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.

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.

A fejlesztés során 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. Ennek folyamata figyelhető meg a következő ábrán:



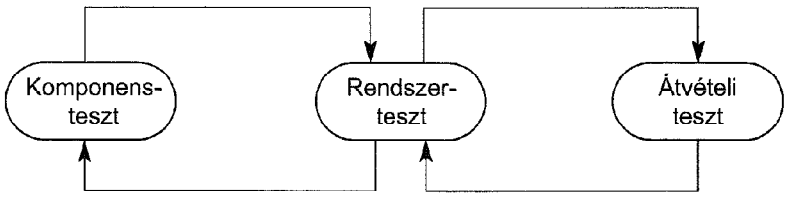
8. ábra. A belövés folyamata

A belövőnek hipotéziseket kell generálnia a program megfigyelhető viselkedéseire, aztán tesztelnie kell ezeket a hipotéziseket annak a reményében, hogy megtalálja az esetleges hibákat.

4.3 Szoftvervalidáció

A szoftvervalidációt általánosan úgy is nevezik, hogy verifikáció és a validáció (V & V). **Célja, hogy megmutassa, a rendszer konform a saját specifikációjával, és hogy a rendszer megfelel a rendszert megvásárló ügyfél elvárásainak**. Ez olyan ellenőrzési folyamatokat foglal magában, mint például szemléket, felülvizsgálatokat a szoftverfolyamat minden egyes szakaszában a felhasználói követelmények meghatározásától kezdve egészen a program kifejlesztéséig.

A kis programok kivételével a rendszerek nem tesztelhetők magukban mint monolitikus egységek. A következő ábra egy háromlépéses tesztelési folyamatot mutat be, ahol teszteljük a rendszer komponenseit, majd az integrált rendszert, és végezetül a teljes rendszert a megrendelő adataival. 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.



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.
3. ***Átvételi tesztelés.*** Ez a tesztelési folyamat legutolsó fázisa a rendszer használata előtt. A rendszert ilyenkor a megrendelő adataival kell tesztelni, amely olyan hiányosságokat vethet fel ami, más esetben nem derül ki.

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

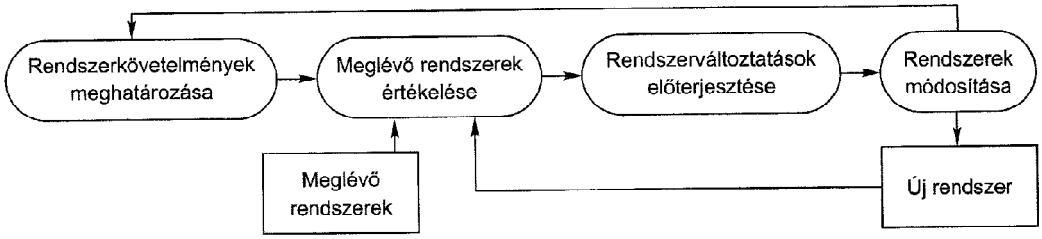
Amikor egy rendszer mint szoftvertermék piacra kerül, gyakran egy másik tesztelés is végbemegy, amelyet béta-tesztelésnek nevezünk. A **béta-tesztelés** magában foglalja a rendszer számos potenciális felhasználójához történő leszállítását, akikkel megegyezés történt a rendszer használatára, és ők jelentik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

4.4 Szoftverevolúció

A szoftverrendszerek flexibilitása az egyik legfontosabb ok, amiért egyre több és több szoftver egyesül nagy, összetett rendszerekben. A szoftverek esetében 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 hardverváltoztatások.

Korábbi felfogások szerint éles demarkációs vonal húzódik a szoftverfejlesztés folyamata és a szoftverevolúció (szoftverkarbantartás) között. **A szoftver karbantartása 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 napjainkban. Kevés szoftverrendszer tekinthető 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 mutatja be a következő ábra, ahol a szoftver életciklusa során folyamatosan változik, a követelmények megváltozásának és a felhasználói igényeknek megfelelően.



10. ábra. A szoftverevolúció folyamata

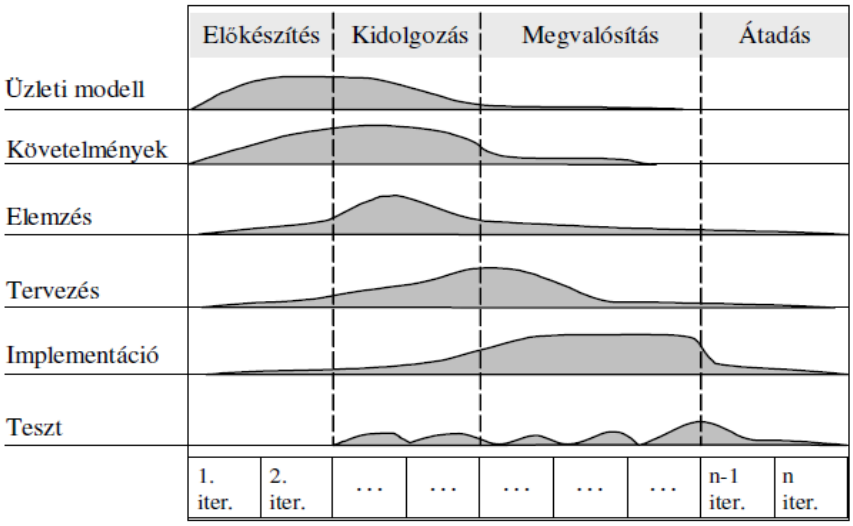
5. RUP (Rational Unified Process)

A Rational Unified Process (RUP) jó példája a modern folyamatmodelleknek, melyek az UML-ből és a hozzá kapcsolódó Unified Software Development Process-ből származnak. Az IBM által felvásárol Rational Software Corporation fejlesztett ki. Jó példája a hibrid modelleknek is, ugyanis mindegyik korábban tárgyalt általános folyamatmodellből tartalmaz elemeket, támogatja az iterációt, és jól illusztrálja a specifikáció és a tervezés tevékenységeit. A RUP felismeri, hogy a konvencionális folyamatmodellek a folyamatoknak csak egy egyszerű nézetét adják. A RUP nem egy kész, követendő eljárást ad minden projektre, sokkal inkább egy könnyen változtatható keretet azok kézbentartásához.

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

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

Az időbeliség alapján az RUP a rendszerfejlesztést négy nagyobb egységre, ***négy diszkrét fázisra*** bontja. Az RUP fázisai sokkal közelebb állnak az üzleti vonatkozásokhoz, mint a technikaiakhoz. A következő ábra bemutatja ezeket:



11. ábra. Fázisok és munkafolyamatok

5.1 RUP rendszerfejlesztési fázisok

Előkészítés

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

Kidolgozás

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

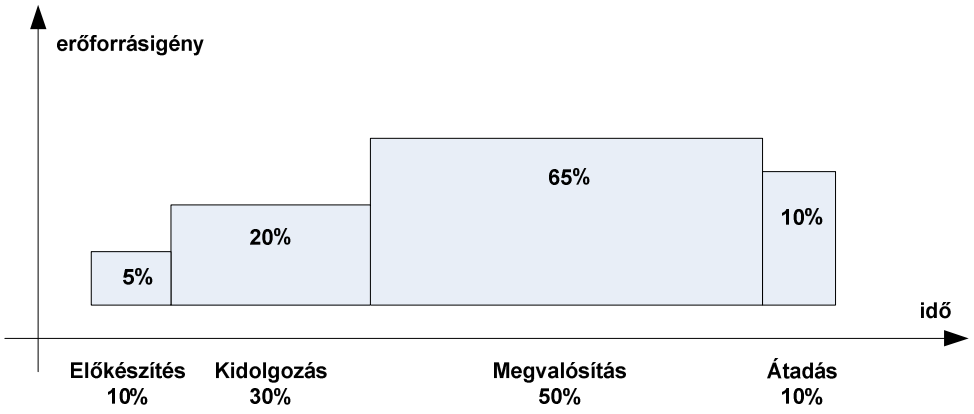
Megvalósítás

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

Átadás

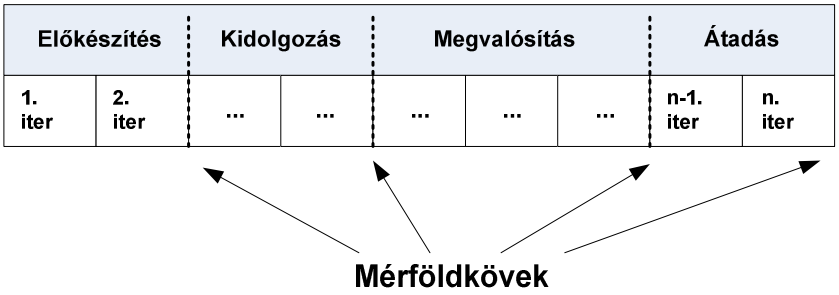
Az *Átadás (transition)* a rendszer bétaváltozatának kipróbálását jelenti, mely során néhány gyakorlott felhasználó teszteli a rendszert és jelentést készít annak helyességéről vagy a hibáiról és hiányosságairól. A megjelenő hibákat ki kell küszöbölni, a még hiányzó részeket ki kell fejleszteni.

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



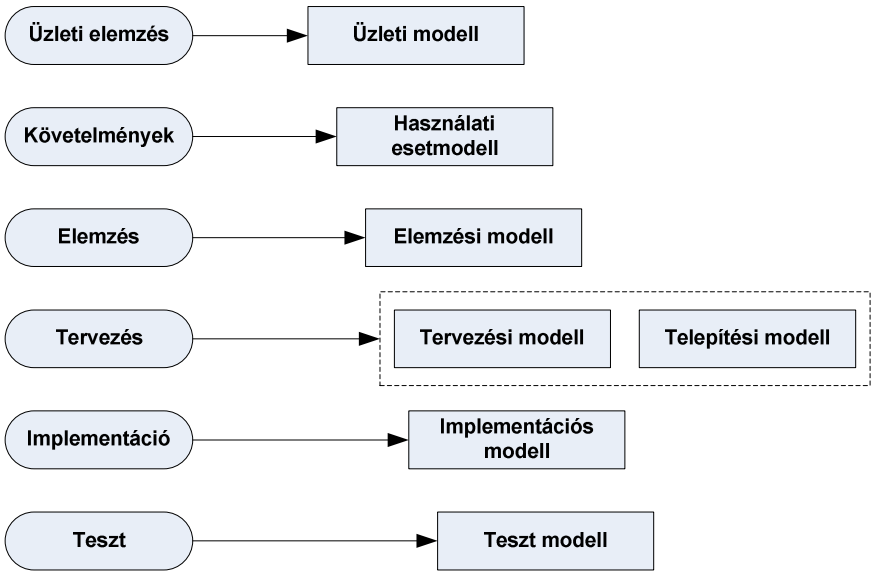
12. ábra. Fázisok idő- és erőforrásigénye

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



13. ábra. Mérföldkövek

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



14. ábra. Modellek

Kezdetben a fejlesztéshez egy megfelelő kiindulópontot keresünk. Az első tevékenységcsoportunk így az ***üzleti modellezés*** *(business model)*, mely során megkeressük a készítendő rendszer üzleti vagy más néven szakterületi környezetét, mely alapvetően az üzleti fogalmakat és folyamatokat jelentik, illetve az azokra hatást gyakorló üzleti munkatársakat.

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

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

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

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

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

6. Szoftverkövetelmények

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

A *követelmény* elnevezést a szoftveriparban nem használják következetes módon, hiszen elég általános fogalomról van szó. Ezért érdemes két szintre bontani a követelmények értelmezését:

1. ***A felhasználói követelmények*** diagramokkal kiegészített természetes nyelvű kijelentések arról, hogy mely szolgáltatásokat várunk el a rendszertől, és annak mely megszorítások mellett kell működnie. Ezek magas szintű, absztrakt követelmények. Ez a leírás az ügyfelek és a fejlesztők képviselői (menedzserek) számára készülnek, akik nem rendelkeznek részletes technikai ismerettel a rendszerről.

2. **A *rendszerkövetelmények*** a rendszer funkcióit, szolgáltatásait és működési megszorításait jelölik ki részletesen. A rendszerkövetelmények dokumentumának (melyet néhol funkcionális specifikációnak is hívnak) pontosnak kell lennie. Pontosan meg kell határoznia, mit kell implementálni. Ez a rendszer vásárlója és a szoftverfejlesztő közötti szerződés része lehet.

A rendszer-specifikáció különböző szintjei azért hasznosak, mert a rendszerről különböző típusú olvasók számára közölnek információt. A felhasználói követelmények inkább absztraktak, míg a rendszerkövetelmények részletezők, megmagyarázva a fejlesztendő rendszer által biztosítandó szolgáltatásokat és funkciókat.

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

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

- **Nemfunkcionális követelmények.** A funkciókra és szolgáltatásokra tett megszorítások. Gyakran a rendszer egészére vonatkoznak. Magukban foglalnak időbeli korlátozásokat, a fejlesztési folyamatra tett megszorításokat, szabványokat.

- **Szakterületi követelmények.** Ezek a követelmények a rendszer alkalmazási szakterületéről származnak és e szakterület jellegzetességeit és megszorításait tükrözik. Ezek lehetnek funkcionális vagy nemfunkcionális követelmények.

A valóságban a különböző követelménytípusok közötti különbségtétel nem olyan éles, mint ahogy ezek az egyszerű meghatározások sugallják.

6.1 Funkcionális követelmények

Egy rendszer funkcionális követelményei azt írják le, hogyan kellene működnie a rendszernek. Ezek a követelmények a fejlesztett szoftver típusától, a szoftver leendő felhasználóitól függenek. Ha felhasználói követelményekként vannak kifejezve, rendszerint egészen absztrakt leírások. Rendszerkövetelményként kifejtve viszont a rendszerfunkciókat részletesen írják le.

Elvben a rendszer funkcionális követelményt leíró specifikációjának teljesnek és ellentmondásmentesnek kellene lennie. A ***teljesség*** azt jelenti, hogy a felhasználó által igényelt összes szolgáltatást definiáljuk. Az ***ellentmondás-mentesség*** azt jelenti, hogy ne legyenek ellentmondó meghatározások. A gyakorlatban nagyméretű, összetett rendszereknél gyakorlatilag lehetetlen a követelmények teljességét és ellentmondás-mentességét elérni.

Ennek egyik oka, hogy nagyméretű, összetett rendszerek specifikációinak írásakor könnyű kifelejteni dolgokat, illetve hibát elkövetni. Másik ok, hogy a különböző kulcsfiguráknak eltérő – és gyakran ellentmondó – igényeik vannak.

6.2 Nemfunkcionális követelmények

A nemfunkcionális követelmények olyan követelmények, amelyek nem közvetlenül a rendszer által szállított specifikus funkciókkal foglalkoznak. Vonatkozhatnak olyan eredendő rendszertulajdonságokra, mint a megbízhatóság, a válaszidő és a tárfoglalás, stb. Ez azt jelenti, hogy ezek gyakran kritikusabbak, mint az egyedi funkcionális követelmények. Nem teljesítésük gyakran a teljes rendszert használhatatlanná teheti. Például ha egy repülőgéprendszer nem teljesíti a vele szemben támasztott megbízhatósági követelményeket.

Nemfunkcionális követelményeket felvethetnek felhasználói igények, költségvetési megszorítások, a szervezeti szabályzat, más szoftver- vagy hardverrendszerekkel való együttműködés igénye vagy olyan külső tényezők, mint a biztonsági szabályozások, adatvédelmi rendelkezések. Ezek alapján a nemfunkcionális követelményeket a következőképpen csoportosíthatjuk:

1. ***Termékre vonatkozó követelmények.*** Ezek a követelmények határozzák meg a termék viselkedését. Pl.: teljesítményre vonatkozó követelmények: cpu és memóriaigény; megbízhatósági követelmények; hordozhatósági követelmények és használhatósági követelmények.
2. ***Szervezeti követelmények.*** Ezek a követelmények a megrendelő és a fejlesztő szervezetének szabályzataiból és ügyrendjéből erednek. Pl.: felhasználandó folyamatszabványok; megvalósítási követelmények: a használt programozási nyelv

vagy tervezési módszer; szállítási követelmények: idő, és hely.

3. ***Külső követelmények.*** Minden olyan követelmény idetartozik, amely a rendszeren és annak fejlesztési folyamatán kívüli tényezőből származik. Ezek magukban foglalhatnak együttműködési követelményeket, amelyek meghatározzák, hogyan érintkezik a rendszer más szervezetek rendszereivel; törvényi követelményeket, továbbá etikai követelményeket.

A nemfunkcionális követelményekkel kapcsolatos általános probléma, hogy esetenként nehéz lehet verifikálni őket. Ennek oka, hogy ezek gyakran olyan általános elvi megkötések, mint például „a rendszernek könnyen használhatónak kell lennie”. Így amikor csak lehet, célszerű a nemfunkcionális követelményeket mennyiségileg, objektíven tesztelhető metrika segítségével kifejezni. A valós életben viszont a rendszer megrendelői számára gyakorlatilag lehetetlen céljaikat mennyiségi követelményekre váltani.

6.3 Szakterületi követelmények

A szakterületi követelmények inkább a rendszer alkalmazásának szakterületéből származnak, nem a rendszer felhasználójának egyéni igényeiből. Itt a legfőbb problémát az jelenti, hogy ezeket a követelményeket az alkalmazás szakterületén használt speciális terminológiával fogalmazzák meg, amihez a szoftvertervezők általában nem értenek. A szakterület szakértői kihagyhatnak információkat a követelményből, mivel az számukra teljesen nyilvánvaló, de a szoftver fejlesztőinek nem. Azonban ha ezek a követelmények nem teljesülnek, nem lehetséges a rendszert kielégítően működtetni.

6.4 Felhasználói követelmények

A rendszer felhasználói követelményeinek a funkcionális és nemfunkcionális követelményeket kell leírniuk úgy, hogy a rendszer azon felhasználói is megértsék azokat, akiknek nincsenek részletes technikai ismereteik. Éppen ezért a rendszernek csak a külső viselkedését írják le. A felhasználói követelményeket egyszerű nyelven, egyszerű táblázatok, űrlapok és könnyen érthető diagramok segítségével kell közreadni.

Ugyanakkor a természetes nyelven írt követelményeknél változatos problémák merülhetnek föl:

1. *Az egyértelműség hiánya.* Olykor nehéz a nyelvet pontos, egyértelmű módon használni anélkül, hogy a dokumentumot terjengőssé és nehezen olvashatóvá ne tennénk.
2. *Követelmények keveredése.* A funkcionális követelmények, nemfunkcionális követelmények, rendszercélok és tervezési információk nem különíthetők el tisztán.
3. *Követelmények ötvöződése.* Több különböző követelmény egyetlen követelményként fogalmazódik meg.

Ha a felhasználói követelmények túl sok információt tartalmaznak, az korlátozza a rendszerfejlesztő szabadságát abban, hogy újító megoldással szolgáljon a felhasználói problémákra, és nehezen érthetővé teszi a követelményeket. A felhasználói

követelményeknek egyszerűen a kulcsfontosságú igényekre kell összpontosítania.

6.5 Rendszerkövetelmények

A rendszerkövetelmények a felhasználói követelmények részletesebb leírásai. Megmagyarázzák, hogy a rendszernek hogyan kell biztosítania a felhasználói követelményeket. Alapul szolgálnak a rendszer megvalósítási szerződéséhez, és ezért az egész rendszer teljes és konzisztens meghatározását tartalmazniuk kell.

A rendszerkövetelmények ideális esetben egyszerűen a rendszer külső viselkedését és működési megszorításait írják le. A tervezés és az implementálás mikéntjével nem szabad foglalkozniuk. A természetes nyelvet gyakran használják rendszerkövetelmények specifikációjának és felhasználói követelményeknek a megírásához egyaránt. Ugyanakkor, mivel a rendszerkövetelmények a felhasználói követelményeknél részletesebbek, a természetes nyelvi specifikációk félrevezetőek és nehezen érthetőek lehetnek:

- A természetes nyelv megértése azon alapszik, hogy az író és az olvasó ugyanazokat a szavakat használja ugyanazokhoz a fogalmakhoz. Ez viszont nincs feltétlenül így, főleg a természetes nyelv többértelműsége miatt.
- A természetes nyelvű követelményspecifikáció túl rugalmas. Ugyanazt a dolgot teljesen különbözőféleképpen elmondhatjuk.
- A természetes nyelvű követelmények modularizálására nincs könnyű módszer. Lehet, hogy bonyolult az összes kapcsolódó követelményt megtalálni. Ahhoz, hogy felfedezzük egy változtatás következményét, előfordulhat, hogy minden követelményt meg kell néznünk.

E problémák miatt a természetes nyelven írt követelményspecifikációk félreérthetők. A rendszerkövetelményeket speciális jelölésekkel is kifejezhetjük. Beszélhetünk *strukturált természetes nyelvről, tervleíró nyelvről, grafikus jelölésekről, és matematikai specifikációkról*.

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

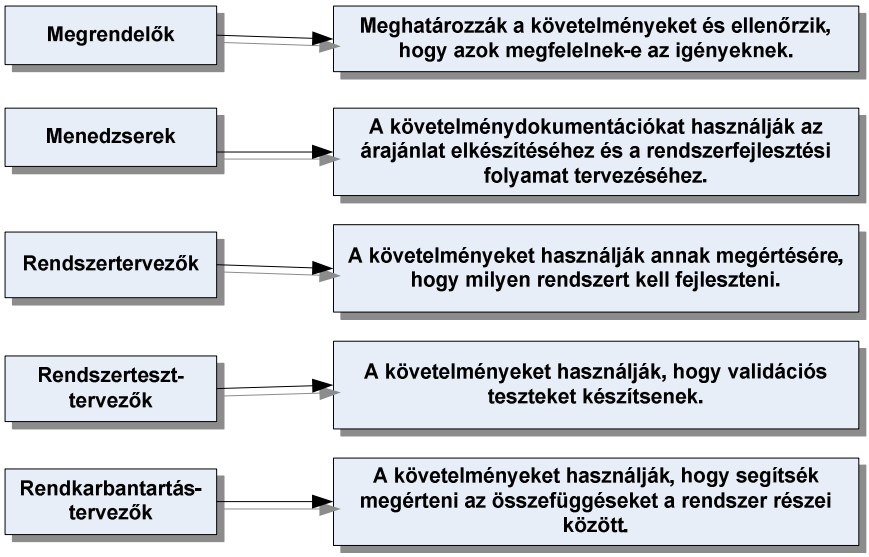
Tervleíró nyelv: PDL. A természetes nyelvű specifikáció többértelműségének kivédésére találták ki a programleíró nyelveket PDL (Program Description Language). A PDL olyan programozási nyelvből származó nyelv, mint a Java vagy az Ada. Tartalmazhat új, absztrakt konstrukciókat a kifejezőerő növelésére.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

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

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

6.6 Szoftverkövetelmények dokumentuma

A szoftverkövetelmények dokumentuma a hivatalos leírása annak, amit a rendszerfejlesztőknek meg kell valósítani. Ajánlott tartalmaznia mind a rendszer felhasználói követelményeit, mind a rendszerkövetelmények részletes specifikációját. A követelménydokumentum használóinak sora igen változatos lehet. A rendszerért fizető felsővezetőktől kezdve egészen a fejlesztőkig.



15. ábra. A követelménydokumentum használói

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

6.6.1 A dokumentum felépítése

A dokumentum felépítésére számos nagy szervezet definiált szabványokat. A legismertebb szabvány az IEEE/ANSI 830-1998-as. A szabvány nem ideális, de rengeteg jó tanácsot fogalmaz meg a követelmények lefektetésével kapcsolatban. A következő felsorolás egy IEEE szabványon alapuló lehetséges követelménydokumentum lehetséges szervezését mutatja:

1. Előszó
2. Bevezetés
3. Szójegyzék
4. Felhasználói követelmények definíciói

5. A rendszer felépítése
6. Rendszerkövetelmény specifikáció
7. Rendszermodellek
8. Rendszerevolúció
9. Függelék
10. Tárgymutató

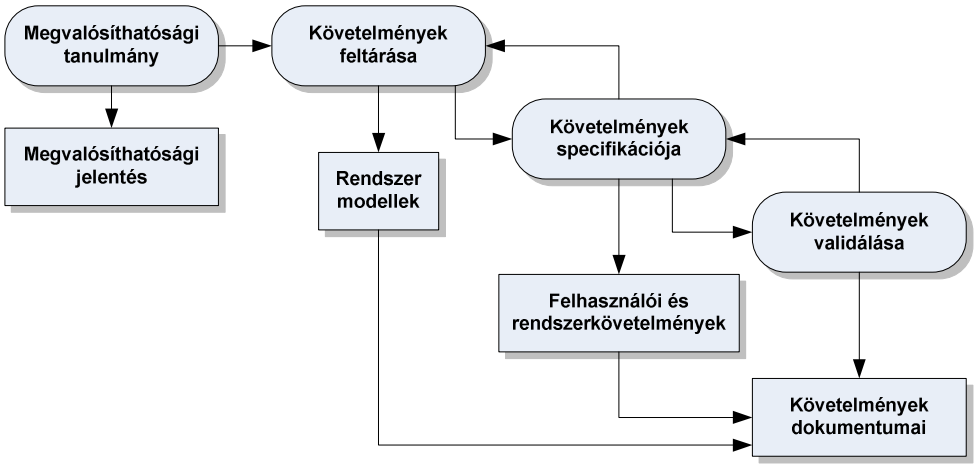
Tehát a dokumentációnak mindig egy egységes anyagnak kell lennie.

7. A követelmények tervezésének folyamatai

A követelmények tervezési folyamatának célja a rendszerkövetelmények dokumentumának létrehozása és karbantartása. A teljes folyamat négy magas szintű tervezési folyamatot foglal magába:

- a rendszer üzleti használatának felmérése (megvalósíthatósági tanulmány)
- a követelmények felderítése (feltárás és elemzés)
- a követelmények átalakítása valamilyen szabványos formátumra (specifikáció)
- annak ellenőrzése, hogy a követelmények a megrendelő által kívánt rendszert definiálják-e (validálás)

A következő ábra a követelmények tervezésének folyamatát mutatja be.



16. ábra. A követelménytervezés folyamata

7.1 Megvalósíthatósági tanulmány

A követelménytervezési folyamatnak minden esetben egy megvalósíthatósági tanulmánnyal kell kezdődnie. A tanulmány bemenetéül az üzleti követelmények kezdeti változatai, a rendszer körvonalazott leírása szolgál. A tanulmány eredményét egy jelentésben

foglalják össze. A tanulmány egy rövid, tömör dokumentum, amely az alábbi kérdésekre próbál választ adni:

1. Támogatja-e a rendszer a vállalat általános célkitűzéseit
2. Megvalósítható-e a rendszer a jelenlegi technológiával adott költségen belül és adott ütemezés szerint?
3. Integrálható-e más, már használatban lévő rendszerekkel?

Az a kérdés, hogy a rendszer hozzájárul-e az üzleti célokhoz igen kritikus, mert amennyiben nem járul hozzá, nincs valódi üzleti értéke. Bár a gyakorlatban számos vállalat fejleszt olyan rendszert, ami nem járul hozzá a céljaikhoz. Ennek oka lehet az, hogy a célkitűzések nincsenek világosan megfogalmazva, vagy nem sikerült definiálni a rendszer követelményeket.

7.2 Követelmények feltárása és elemzése

A követelménytervezés (szoftverspecifikációs folyamat) **második** nagy tevékenysége, az a folyamat, amely során a tervezett vagy meglévő rendszerekről információt gyűjtünk, majd ebből kiszűrjük a felhasználói és rendszerkövetelményeket. E tevékenység során együtt kell működni a szoftverfejlesztőknek a megrendelőkkel és a végfelhasználókkal, azért, hogy kiderítsék milyen szolgáltatásokat kellene biztosítani a rendszernek. Itt ki kell választani az úgynevezett **kulcsfigurákat** (olyan személy, akit a rendszer közvetetten, vagy közvetlenül érint)**,** olyan személyeket vagy csoportokat, akiknek közvetett vagy közvetlen befolyása lehet a rendszerkövetelményekre.

A feltárás és elemzés nehézségeinek több oka lehet:

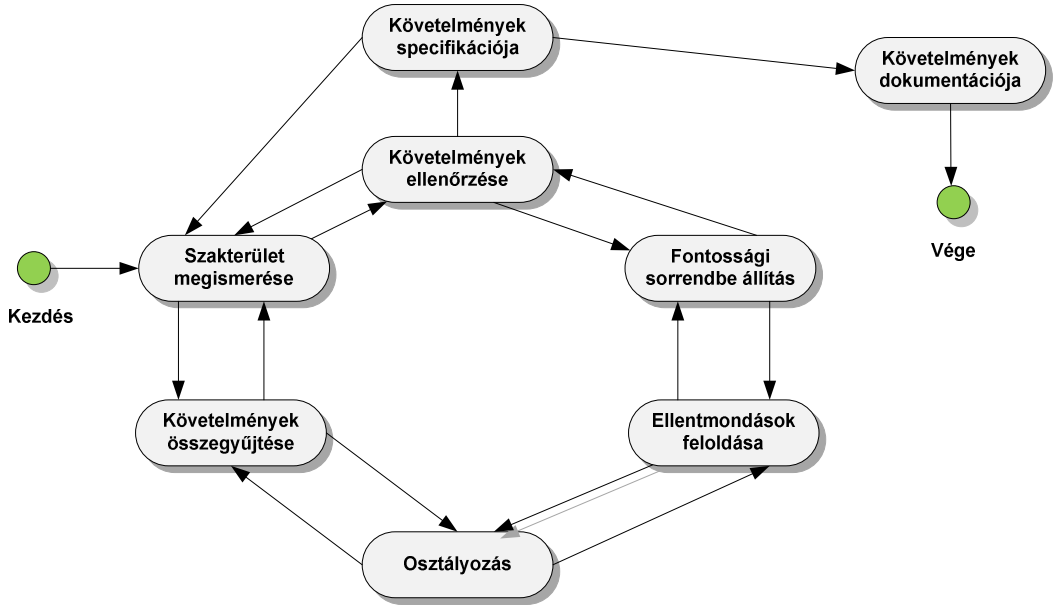
- A kulcsfigurák gyakran nem tudják pontosan, mit várnak el a számítógépes rendszertől. Bonyolult lehet számukra annak kifejezése, mit akarnak a rendszertől; valóságtól elrugaszkodott kívánságaik lehetnek.

- A rendszer kulcsfigurái a saját szakterületi fogalmaikkal fejtik ki a követelményeket.
- Az egyes kulcsfiguráknak különböző követelményeik vannak, ebben a követelménytervezőknek fel kell fedezniük a közös dolgokat és ellentmondásokat.

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

- A gazdasági és az üzleti környezet eközben dinamikusan változik, új kulcsfigurák jelenhetnek meg, újabb követelményekkel.

A feltárási és elemzési folyamat általános modellje:



17. ábra. A feltárás és elemzés általános modellje

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

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

7.2.1 Nézőpont-orientált szemlélet

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

A nézőpontok használhatók a kulcsfigurák és egyéb követelményforrások egyfajta osztályozására:

1. Az ***interaktor*** *nézőpontok* a rendszerrel közvetlenül érintkező személyeket vagy más rendszereket reprezentálnak.
2. A ***közvetett*** *nézőpontok* olyan kulcsfigurákat reprezentálnak, akik nem használják ugyan közvetlenül a rendszert, de a követelményeket valahogyan befolyásolják. Pl.: egy bank

vezetősége és a bank biztonsági személyzete.

3. A ***szakterületi*** *nézőpontok* a szakterületnek a rendszerkövetelményeket befolyásoló jegyeit és megszorításait jelképezik. Pl.: a bankok közötti kommunikációra kifejlesztett szabványok.

A nézőpontok jellemzően különbözőfajta követelményeket biztosítanak. Az *interaktor nézőpontok* részletes rendszerkövetelményeket biztosítanak, amelyek lefedik a rendszer eszközeit és interfészeit. A *közvetett nézőpontok* sokkal inkább magas szintű szervezeti követelményeket és megszorításokat jelentenek. A *szakterületi nézőpontok* rendszerint a rendszerre vonatkozó szakterületi megszorításokat nyújtanak.

A szoftvertervezési nézőpontok két szempontból fontosak. Egyrészt lehet, hogy a rendszert fejlesztő szoftvertervezők már dolgoztak hasonló rendszereken, és a korábbi tapasztalataik alapján javasolhatnak követelményeket. Másrészt a rendszert kezelő és karbantartó műszaki személyzetnek is lehetnek olyan követelményei, amelyek leegyszerűsítenék a rendszer támogatását.

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

7.2.2 Forgatókönyvek

Az emberek általában könnyebben kezelik a valós életbéli problémákat, mint az absztrakt leírásokat. **Ezek interakció-sorozatok leírásai**, különösen hasznosak akkor, ha további részletekkel szeretnénk kiegészíteni a követelmények körvonalazott leírását. Minden forgatókönyv egy vagy több lehetséges interakciót takar. Számos forgatókönyvtípust kifejlesztettek már, ezek mind különböző részletezettségű, különböző típusú információkat nyújtanak a rendszerről.

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

1. a rendszer, illetve a felhasználói elvárások;
2. a forgatókönyvbeli események normális menetének leírása;
3. leírást arról, mi romolhat el, és ezt hogyan kezeli a rendszer;
4. egyéb tevékenységek leírása, amelyek ugyanabban az időben mehetnek végbe;
5. a rendszer végállapotának leírását a forgatókönyv befejeződésekor.

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

7.2.3 Etnográfia

Az *etnográfia* megfigyelésen alapuló technika, amely felhasználható a társadalmi és szervezeti követelmények megértéséhez. Az elemző elmélyed abban a munkakörnyezetben, ahol a rendszert majd használni fogják. Megfigyeli a napi munkát, és jegyzeteket készít az aktuális feladatokról.

Az emberek gyakran nehéznek találják kifejezni munkájuk részleteit, ez természetükből fakad. A saját munkájukat megértik, de azt valószínűleg nem, hogy az milyen összefüggésben áll a szervezet többi munkájával. A társadalmi és szervezeti tényezők csakis akkor válhatnak világossá, ha egy tárgyilagos megfigyelő észreveszi őket.

Az etnográfiai tanulmányok a folyamat kritikus részleteit tárhatják fel, amelyek más feltárási technikáknál gyakran elmaradnak. Ugyanakkor, mivel a hangsúly a végfelhasználón van, ez a megközelítés nem alkalmas a szervezeti vagy a szakterületi követelmények felderítésére.

7.3 Követelmények validálása

A követelmények validálása azzal foglalkozik, hogy a követelmények valóban azt a rendszert definiálják, amit a megrendelő akar. A követelmények validálása átfedi az elemzést, mivel célja, hogy megtalálja a követelményekkel kapcsolatos problémákat. Azért fontos, mert a követelmény-dokumentumbeli hibák jelentős átdolgozási költségekhez vezethetnek. Az ilyenkor felderített hibák javításának költsége sokkal magasabb, mint a tervnek vagy kódolási hibáknak a kijavításáé. Ennek az oka abban áll, hogy a követelmények megváltozása általában azzal jár, hogy a rendszertervet és az implementációt is meg kell változtatni, a rendszert pedig újra tesztelni kell.

A validálási folyamat során ajánlott ellenőrzéseket végezünk el:

1. ***Validitás-ellenőrzések.*** A rendszereknek sokféle kulcsfigurája van, eltérő igényekkel, és a követelmények bármilyen csoportja elkerülhetetlenül a kulcsfigurák közösségének egészével kötött kompromisszum lesz.
2. ***Ellentmondás-mentességi ellenőrzések.*** A dokumentumban szereplő követelmények nem mondhatnak ellent egymásnak.
3. ***Teljesség-ellenőrzések.*** A követelménydokumentumnak javasolt tartalmaznia mindazon követelményt, amely a rendszer felhasználói által kért összes funkciót és megszorítást definiálja.
4. ***Megvalósíthatósági ellenőrzések.*** Létező technológiák ismereteit felhasználva el-lenőriznünk kell a követelményeket, hogy azok tényleg megvalósíthatók-e. Ezeknek az ellenőrzéseknek ki kell terjedniük a költségvetésre és a rendszerfejlesztés ütemtervére is.
5. ***Verifikálhatóság.*** Hogy csökkentsük a megrendelő és a vállalkozó közötti viták lehetőségét, mindig ellenőrizhető módon kell rögzíteni a rendszerkövetelményeket. Ez azt jelenti, hogy meg kell tudnunk írni egy olyan követelményhalmazt, amellyel bizonyítani lehet, hogy az átadott rendszer teljesíti az összes előírt követelményt.

A követelmények validálására számos technika létezik, melyek együtt vagy egyedileg használhatók:

1. ***Követelmények felülvizsgálata.*** A követelményeket módszeresen elemzi felülvizsgálók egy csoportja. Manuális folyamat, amely történhet formálisan és informálisan is. Az informális felülvizsgálat egyszerűen csak a vállalkozókat érinti, akik a követelményeket a kulcsfigurával vitatják meg. A formális követelmény-felülvizsgálatban a fejlesztő csoportnak „végig kell vezetnie" az ügyfelet a rendszerkövetelményeken.
2. ***Prototípus-készítés.*** Ebben a validálási folyamatban a rendszer egy végrehajtható modelljét mutatjuk be a végfelhasználóknak és a megrendelőknek, így tapasztalatokat szerezhetnek a modellel kapcsolatban, hogy lássák, vajon kielégíti-e a valós igényeket.
3. ***Teszteset generálása.*** A követelményeknek tesztelhetőeknek kell lenniük. Amennyiben a követelmények tesztjeit a validálási folyamat részeként tervezték kivitelezni, az gyakran követelményi problémákat fed fel. Ha a tesztet nehéz vagy lehetetlen megtervezni, az általában azt jelzi, hogy a követelmények nehezen implementálhatok, és tanácsos azokat újra átgondolni.

A követelmény validálás nehéz feladat, gyakorlott számítógépes szakemberek számára is igen kemény feladat egy ilyen absztrakt elemzés elvégzése, a rendszer felhasználói számára pedig még keményebb.

7.4 Szoftverprototípus készítése

A prototípus a szoftverrendszer kezdeti verziója, amelyet arra használnak, hogy bemutassák a koncepciókat, kipróbálják a tervezési opciókat, és hogy jobban megismerjék a problémát és annak lehetséges megoldásait. A prototípus gyors, iteratív fejlesztése azért nagyon fontos, mert a költségek így ellenőrizhetők.

A szoftverprototípus több helyen is használható a fejlesztés folyamatában:

- A követelménytervezés folyamatában.
- A rendszertervezési folyamatban.
- A tesztelési folyamatban.

A követelmény fázisban a prototípus fejlesztése alatt fény derülhet a követelményekkel kapcsolatos lehetséges hibákra, esetleg új ötletek kapcsán új rendszerkövetelményeket is indítványozhatnak.

A rendszertervezési folyamatban a rendszerprototípus felhasználható a tervezési tapasztalatok alkalmazására, illetve a javasolt terv megvalósíthatóságának felülvizsgálatára. Például a gyors prototípus készítés az egyetlen módja annak, hogy a felhasználók bevonásával grafikus felhasználói felületeket fejlesszünk ki.

A rendszertesztelés fázisában a prototípusok segítségével az eredmények ellenőrzéséhez szükséges munka *visszacsatoló tesztek* segítségével csökkenthető. Ilyenkor ugyanazokra a tesztesetekre futtatjuk a prototípust és a rendszert. Ha mindkét rendszer ugyanazt az eredményt adja, akkor a teszteset valószínűleg nem talált hibát.

A prototípuskészítés rendszerint a szoftverfolyamat korai szakaszában növeli a költségeket, később viszont jelentősen csökkenti, mert elkerülhetők az újraírási munkák.

A prototípuskészítés folyamata **négy** különböző fázissal írható le:

- A prototípus céljainak megállapítása

- A prototípus funkcionalitásának definiálása
- A prototípus fejlesztése
- A prototípus kiértékelése

A prototípuskészítés céljait érdemes az elején írásban megadni, mert e nélkül a vezetés vagy a végfelhasználók félreérthetik a rendeltetését. Ilyen cél lehet: az alkalmazás megvalósíthatóságának demonstrálása vagy a felhasználói felületek bemutatása, stb. Ugyanaz a prototípus nem szolgálhatja az összes célt.

A folyamat következő szakasza annak eldöntése, hogy mit tegyünk bele a prototípusba. Ezután következik a prototípus fejlesztése és végül az utolsó szakasz pedig a kiértékelés. Ennek folyamán gondoskodni kell a felhasználók képzéséről, mivel időbe telik, amíg megszokják az új rendszert, és csak ezután fedezhetik fel a követelménybeli hibákat és hiányosságokat.

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

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

8. Tervezés

A szoftvertervezés lényege a szoftver logikai szerkezetére vonatkozó döntések meghozatala. Bizonyos esetekben ezt a logikai szerkezetet egy olyan modellezőnyelv segítségével leírt modellel ábrázoljuk, mint amilyen az UML, más esetekben a tervet informális jelölésrendszerrel leírt vázlatokkal reprezentáljuk.

8.1 Architektúrális tervezés

A nagy rendszereket olyan alrendszerekre bontják, amelyek biztosítják az egymással kapcsolatban lévő szolgáltatásokat. Ezen alrendszerek azonosításának és az alrendszer vezérlésére és kommunikációjára szolgáló keretrendszer létrehozásának kezdeti tervezési folyamatát **architektúrális tervezésnek** nevezzük. Egy kreatív folyamat, ahol megpróbálunk előállítani egy rendszerszerkezetet, amely megfelel a funkcionális és nemfunkcionális rendszerkövetelményeknek.

A rendszer architektúrája befolyásolja a rendszer teljesítményét, robusztusságát, eloszthatóságát és karbantarthatóságát, ezért fontos, hogy a szoftvertervezőket rákényszerítsük arra, hogy a tervezés kulcsfontosságú aspektusaival már a folyamat korai

szakaszában foglalkozzanak. Az alkalmazás számára választott szerkezet olyan nemfunkcionális rendszerkövetelményektől is függhet, mint például a *teljesítmény*, *védettség*, *biztonságosság*, *rendelkezésre állás*, és *karbantarthatóság*.

Az architektúrális tervezés magában foglalja a rendszerek alrendszerekre bontását is. Az alrendszerek tervezése tulajdonképpen a rendszer durva szemcsézettségű komponensekre történő absztrakt felbontása, amely komponensek lehetnek önálló rendszerek. Az alrendszerek terveit általában blokkdiagram segítségével írjuk le, ahol a diagram dobozai az egyes alrendszereket reprezentálják.

8.1.1 Architektúrális tervezésési döntések

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

1. Létezik-e olyan általános alkalmazás architektúra, amely a tervezendő rendszer számára mintául szolgálhat?
2. Hogyan osztjuk szét a rendszert processzorokra?
3. Milyen architektúrális stílus lenne a rendszer számára megfelelő?
4. Milyen alapvető megoldásokat alkalmazunk a rendszer strukturálására?
5. Hogyan lehet a rendszer szerkezeti egységeit modulokra felbontani?
6. Milyen stratégiát kell alkalmazni az egységek működésének vezérlésével kapcsolatban?
7. Hogyan értékelik majd ki az architektúrális tervet?
8. Hogyan kell a rendszer-architektúrát dokumentálni?

Az architektúrális tervezés folyamatának eredménye egy architektúrális tervezési dokumentum, amely tartalmazza a rendszer grafikus reprezentációit és a hozzájuk kapcsolódó leíró szöveget. Tartalmaznia kell annak leírását, hogy a rendszert hogyan lehet alrendszerekre bontani, az egyes alrendszerek miképpen oszthatók modulokra. A kialakítandó architektúrális modellek többek között az alábbiak lehetnek:

1. A ***statikus szerkezeti modell*** azt mutatja meg, hogyan lehet az alrendszereket és komponenseket különálló egységként fejleszteni.
2. A ***dinamikus folyamatmodell*** azt ábrázolja, hogy a rendszer hogyan szervezhető futási idejű folyamatokba. Ez különbözhet a statikus modelltől.
3. Az ***interfészmodell*** az alrendszerek által publikus interfészeiken keresztül nyújtott szolgáltatásait írja le.
4. A ***kapcsolatmodellek*** az alrendszerek közötti kapcsolatokat (például adatáramlás) mutatják be.
5. Az ***elosztási modell*** azt adja meg, hogy az egyes alrendszereket hogyan kell elosztani a számítógépek között.

A rendszer-architektúrák leírására számos kutató az architektúraleíró nyelvek (architectural description languages, ADL) használatát és az UML nyelvet javasolja.

8.1.2 A rendszer felépítése

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

A rendszer felépítését számos modell segítségével írhatjuk le. Most ezekből nézünk meg a három legáltalánosabbat.

8.1.2.1 A tárolási modell

A rendszert felépítő alrendszereknek a hatékony együttműködés céljából információt kell cserélniük, amely alapvetően két módon történhet:

1. Minden megosztott adatot a minden alrendszer által elérhető, központi adatbázisban kell elhelyezni. **A megosztott adatbázison alapuló rendszermodellt *tárolási modellnek* nevezzük.**
2. Minden alrendszer saját adatbázist tart fent. Ekkor az egyéb alrendszerek közötti adatcsere üzenetküldés segítségével történik.

A nagy mennyiségű adatokkal dolgozó rendszerek osztott adatbázisok és tárolók köré szerveződnek. Ez a modell azon alkalmazások számára megfelelő, ahol az egyik alrendszerben keletkező adatok egy másik alrendszerben kerülnek felhasználásra.

A megosztott tárolók előnyei és hátrányai a következők:

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

2. Az alrendszereknek azonban azonos adattárolási modellel kell rendelkezniük. Ekkor kompromisszumokra van szükség az egyes eszközök között, amely azonban rossz irányban befolyásolhatja a teljesítményt. A közös sémának nem megfelelő adatmodellel rendelkező új alrendszerek integrálása nehézzé vagy lehetetlenné válik.

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

4. Az olyan tevékenységek, mint a biztonsági mentés, a védelem, a hozzáférés-szabályozás és a hiba utáni visszaállítás központosítottak, így az eszközök lényeges feladataikra összpontosíthatnak.

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

6. A megosztottság modellje a tárolási sémán keresztül látható. Ez egyenes utat biztosít az új eszközök integrálásához, amennyiben azok megfelelnek a közös adatmodellnek.

7. A tároló több gép közötti elosztása azonban bonyolult is lehet. Habár lehetőség van egy logikailag központosított tároló elosztására, problémák léphetnek fel az adatok redundanciájával és inkonzisztenciájával kapcsolatban.

8.1.2.1 A kliens-szerver modell

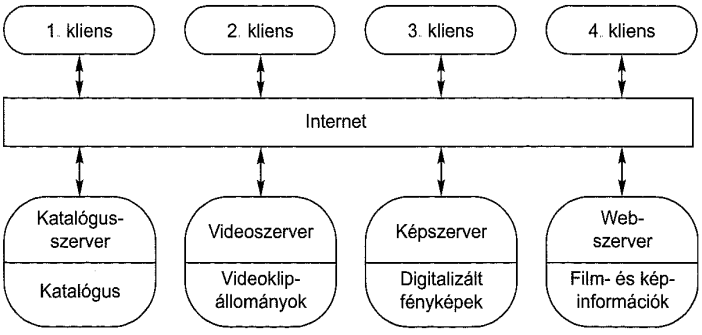
A kliens-szerver architektúra modellje egy osztott rendszer modellje, amely megmutatja, hogy az adat és a feldolgozás hogyan oszlik meg feldolgozóegységek között. A modell három fő komponensből, a kliensek és szerverek halmazából, valamint a hálózatból tevődik össze.

- **Szerverek halmaza:** amelyek más alrendszerek számára szolgáltatásokat nyújtanak. Ilyenek például a nyomtatószerverek.

- **Kliensek halmaza:** amelyek hozzáférnek a szerverek által biztosított szolgáltatásokhoz. Ezek általában önálló létjogosultsággal rendelkező alrendszerek. Egy kliensprogramnak számos példánya futhat egyidejűleg.
- **Hálózat:** amely lehetővé teszi, hogy a kliensek hozzáférjenek a szolgáltatásokhoz. Ez nem szükséges akkor, ha mind a kliensek, mind pedig a szerverek egyetlen gépen futnak.

A klienseknek szükségük van arra, hogy ismerjék az elérhető szerverek és az általuk biztosított szolgáltatások neveit, de a szervereknek nem kell tudniuk sem a kliens azonosságát, sem pedig azt, hogy hány kliens van. A kliensek a szerverek által biztosított szolgáltatásokat távoli eljáráshívásokkal érik el, egy kérés-válasz alapú protokoll segítségével, mint például a www esetén használt http protokoll. Lényegében a kliens elküld egy kérést a szervernek, és addig vár, amíg választ nem kap.

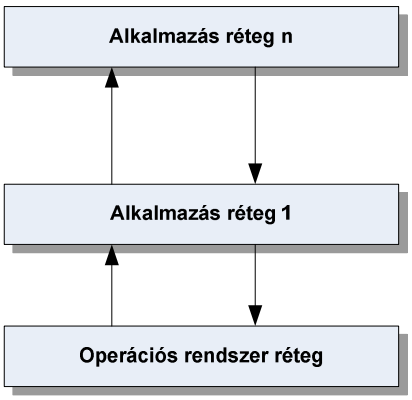
A kliens-szerver modell legfontosabb előnye osztott architektúrája. Hatékonyan használható sok, osztott feldolgozási egységből álló hálózati rendszerek esetén. Egy új szerver könnyen hozzáadható a rendszerhez és integrálható annak többi részével. A következő ábra egy példát mutat be a kliens-szerver architektúrára:



. ábra. Egy film és kép könyvtári rendszer architektúrája

8.1.2.2 Rétegzett modell

A rétegzett modell a rendszert rétegekbe szervezi, amelyek mindegyike valamilyen szolgáltatásokat biztosít. Minden ilyen rétegre úgy tekinthetünk, mint egy *absztrakt gépre,* amelynek gépi nyelvét a réteg által biztosított szolgáltatások definiálják. Ezt a „nyelvet". szolgáltatásokat használják az absztrakt gép következő szintjének megvalósítására. Példaként jól említhető a hálózati protokollok OSI-referenciamodellje. A következő ábra a rétegzett modell általános felépítését mutatja.



. ábra. Rétegzett rendszerek

A rétegalapú megközelítés segíti a rendszerek inkrementális fejlesztését. Mihelyt egy réteg kifejlesztésre került, a réteg által biztosított szolgáltatások egy része elérhetővé tehető a felhasználók számára. Egy réteg helyettesíthető egy másik, azzal ekvivalens réteggel, ha interfésze nem változik meg. Sőt egy réteg interfészének megváltozása vagy a réteg új lehetőségekkel történő bővítése csak a szomszédos réteget érinti. Csak a belső, gépfüggő rétegeket kell újraimplementálni ahhoz, hogy figyelembe vegyük egy másik operációs rendszer vagy adatbázis lehetőségeit.

A rétegzett megközelítés hátránya, hogy ily módon a rendszerek strukturálása igen bonyolulttá is válhat. Az alapvető adottságokat, amelyre minden absztrakt gépnek szüksége van a belső rétegek biztosíthatják. A külső réteg felhasználó által igényelt szolgáltatásainak „át kell ütniük" több szomszédos réteget is ahhoz, hogy hozzáférjenek a több réteggel alattuk elhelyezkedő réteg szolgáltatásaihoz. Ez felforgatja a modellt abban az értelemben, hogy egy külső réteg a továbbiakban már nem csupán egyszerűen a közvetlen megelőzőjétől függ.

8.1.3 Moduláris felbontás

A teljes rendszer-architektúra kiválasztásra után, dönteni kell az alrendszerek modulokra bontása során alkalmazott megközelítésről. Az alrendszerek és modulok nem különböztethetők meg egyértelműen, de érdemes azokat a következő módon elképzelni:

1. Egy **alrendszer** egy olyan önálló rendszer, amely működése nem függ más alrendszerek szolgáltatásaitól. Az alrendszerek modulokból épülnek fel.

2. Egy **modul** olyan rendszerkomponens, amely más modulok számára szolgáltatás(oka)t biztosít.

Az alrendszerek modulokra bontása során két fő stratégia ismeretes:

1. ***Objektumorientált felbontással*** a rendszert egymással kommunikáló objektumok halmazára bontjuk fel.
2. ***Funkcióorientált csővezetékek*** használatával a rendszert bemenő adatokat elfogadó és azokat kimenő adatokká alakító funkcionális modulokra bontjuk.

8.1.3.1 Objektumorientált felbontás

Egy objektumorientált architekturális modell a rendszert lazán kapcsolódó, jól definiált interfészekkel rendelkező objektumok halmazára tagolja. Az objektumok a többi objektum által biztosított szolgáltatásokat hívják. Az objektumorientált felbontás objektumosztályokkal, azok attribútumaival és műveleteivel foglalkozik. Implementációkor az objektumok ezekből az osztályokból jönnek létre, és az objektum műveleteinek koordinálásához valamilyen vezérlési modellt alkalmaznak.

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

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

8.1.3.1 Funkcionált csővezeték

A funkcióorientált csővezetékek használata esetén, amit más néven **adatfolyam-modellnek** is neveznek, funkcionális transzformációk dolgozzák fel bemenetüket és hoznak létre kimeneteket, közöttük áramlanak az adatok és sorban előrehaladva kerülnek átalakításra. Minden feldolgozási lépés transzformációként valósul meg. A bemeneti adatok ezeken a transzformációkon mennek keresztül, míg végül átalakulnak kimeneti adatokká. A transzformációk mind szekvenciálisan, mind pedig párhuzamosan végrehajthatók. Az adatok egyesével vagy kötegelve is feldolgozhatók.

Az interaktív rendszereket nehéz csővezetékelvű modell alapján megírni, míg az egyszerű szöveges be-, illetve kimenet modellezhető ily módon. Előnye: támogatja a transzformációk újrafelhasználhatóságát, könnyen érthető és bővíthető új transzformációkkal, implementálható konkurens és szekvenciális környezetben egyaránt. Hátránya: Az egyes transzformációknak vagy egyeztetniük kell egymással az átadandó adatok formátumát, vagy a kommunikációban részt vevő adatok számára egy szabványos formátumot kell kialakítani.

8.1.4 Vezérlési stílusok

Ahhoz, hogy az alrendszerek rendszerként működjenek, vezérelni kell őket, hogy szolgáltatásaik a megfelelő helyre a megfelelő időben eljussanak. Mivel a strukturális modellek nem tartalmaznak ilyen elemeket ezért a rendszer kiépítőjének az alrendszereket

valamilyen vezérlési modellnek megfelelően kell szerveznie. A vezérlési modellek az alrendszerek közötti vezérlési folyamatokkal foglalkoznak.

A szoftverrendszerekben két általános vezérlési stílust alkalmaznak:

1. ***Központosított vezérlés.*** *A* vezérlés teljes felelősségét egyetlen alrendszer látja el, amely beindítja és leállítja a többi alrendszert.

2. ***Eseményalapú vezérlés.*** Ahelyett, hogy a vezérlési információ egyetlen alrendszerbe lenne beágyazva, minden alrendszer válaszolhat egy külsőleg létrejött eseményre. Ezek az események vagy más alrendszerekből, vagy a rendszer környezetéből származhatnak.

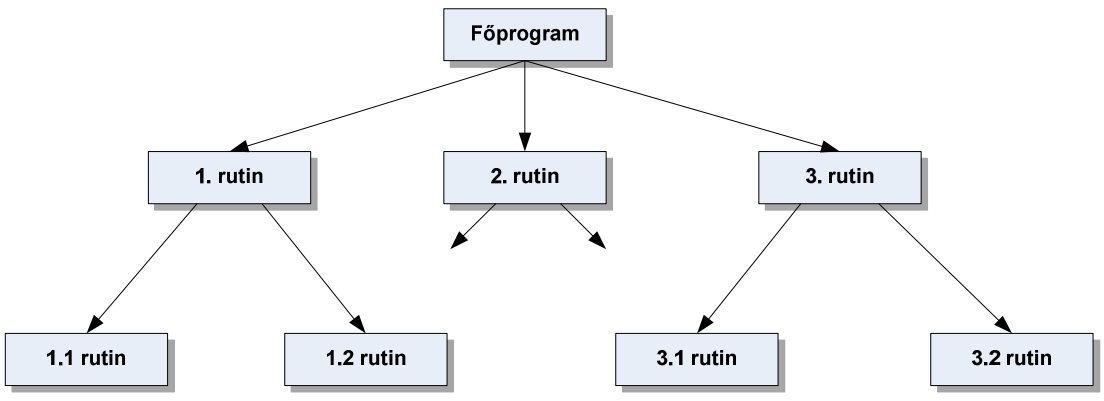
A vezérlési stílusok kiegészítik a strukturális stílusokat. Minden korábban bemutatott strukturális stílust meg lehet valósítani központosított és eseményalapú vezérléssel egyaránt.

8.1.4.1 Központosított vezérlés

A központosított vezérlési modellben létezik egy kitüntetett alrendszer, a **rendszer-vezérlő**, amely a többi alrendszer végrehajtásáért felelős. A központosított vezérlési modellek **két** csoportba oszthatók attól függően, hogy a vezérelt alrendszerek szekvenciálisan vagy párhuzamosan hajtódnak-e végre.

1. ***A hívás-visszatérés modell.*** Ez a megszokott fentről le alprogram modellje, ahol a vezérlés az alprogram-hierarchia csúcsán kezdődik és alprogramhívások segítségével jut el a fa alsóbb szintjeire. Ez a modell csak szekvenciális rendszerek esetén alkalmazható.
2. ***A kezelőmodell.*** Konkurens rendszerekre alkalmazható. Ebben egy kijelölt rendszerkezelő rendszerkomponens irányítja a többi rendszerfolyamat indítását, leállítását, valamint koordinálja azokat. A folyamat olyan alrendszer vagy modul, amely végrehajtható más folyamatokkal párhuzamosan. Ez a fajta modell használható szekvenciális rendszerek esetén is.

A hívásvisszatérés modell



ábra. A hívás-visszatérés modell

A vezérlés a hierarchiában magasabb szintű rutintól jut el az alacsonyabb szinten lévőhöz, majd visszatér a hívás helyére. A hívás-visszatérés modell modulszinten használható a tevékenységek és objektumok vezérlésére, mert sok objektumorientált rendszerben az objektumok műveletei (metódusok) eljárásként vagy függvényként vannak implementálva.

A modell merev és korlátozott természete egyben előny és hátrány is. Erőssége a vezérlési folyam elemzésének és bizonyos bemenet esetén a rendszer válasza kiszámíthatóságának viszonylagos egyszerűsége. Hátránya, hogy a normálistól eltérő működés esetén használata kényelmetlen.

A kezelőmodell

A rendszert vezérlő folyamat a rendszer állapotváltozói alapján dönti el, hogy az egyes folyamatokat mikor kell elindítani, illetve leállítani. Ellenőrzi, hogy a többi folyamat hozott-e létre feldolgozandó vagy feldolgozásra továbbküldendő információt. A vezérlő általában ciklikusan ellenőrzi az érzékelőket és folyamatokat, hogy bekövetkezett-e valamilyen esemény vagy állapotváltozás. Éppen ezért ezt a modellt **eseményciklus-**modellnek is szokás nevezni.

8.1.4.2 Eseményvezérelt rendszerek

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

1. ***Eseményszóró modellek.*** Ezekben a modellekben egy esemény minden alrendszerhez eljut, és bármelyik, az esemény kezelésére programozott alrendszer reagálhat rá. A vezérlési politika nincs beépítve az esemény- és üzenetkezelőbe. Az alrendszerek eldöntik, hogy mely eseményekre tartanak igényt, az esemény- és üzenetkezelő pedig biztosítja, hogy ezek az események eljussanak hozzájuk.
2. ***Megszakításvezérelt modellek.*** Ezeket kizárólag olyan valós idejű rendszerekben használják, ahol egy megszakítás-kezelő észleli a külső megszakításokat. Ezek aztán valamely más komponenshez kerülnek feldolgozásra.

8.2 Objektumorientált tervezés

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

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

- ***Objektumorientált elemzés:*** a szoftver objektumorientált modelljének kialakításával foglalkozik.

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

- ***Objektumorientált programozás****:* a szoftverterv objektumorientált programozási nyelven történő megvalósítása. Pl.: C++, D, Java.

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

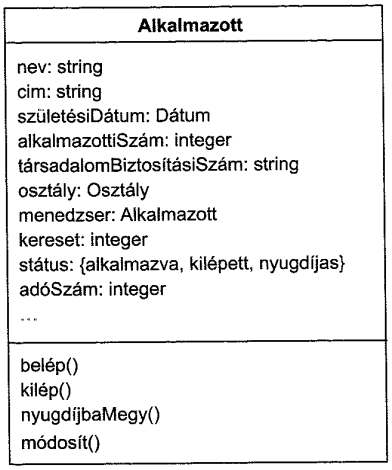
8.2.1 Objektumok és objektumosztályok

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

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

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

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



. ábra. Az Alkalmazott objektumosztály

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

```
// Egy puffer objektum módszerének hívása, amely visszaadja a pufferben található következő értéket
v = Buffer.GetNextElement();
// Puffer elemszámának a beállítása
v = Buffer.SetElements(20);
```

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

Az újabb OOP nyelvekben azonban, mint pl. a JAVA-ban vagy a C#-ban, léteznek a szálak, amelyek megengedik a konkurens módon végrehajtódó objektumok létrehozását és az **aszinkron kommunikációt** (a hívóobjektum folytatja a működését az általa igényelt szolgáltatás futása alatt is). Ezek a konkurens objektumok kétféleképpen implementálhatók:
1. ***Aktív objektumok:*** önmaguk képesek belső állapotukat megváltoztatni és üzenetet küldeni, anélkül, hogy más objektumtól vezérlőüzenetet kaptak volna. (Ellentétük a passzív objektum.) Az aktív objektumot reprezentáló folyamat ezeket a műveleteket folyamatosan végrehajtja, így soha nem függesztődik fel.

2. ***Szerverek****:* az objektum a megadott műveleteknek megfelelő eljárásokkal rendelkező párhuzamos folyamat. Az eljárások egy külső üzenetre válaszolva indulnak el és más objektumok eljárásaival párhuzamosan futhatnak. Mikor befejezték a tevékenységüket, az objektum várakozó állapotba kerül és további kéréseket vár.

A szervereket leginkább osztott környezetben érdemes használni, ahol a hívó és a hívott objektum különböző számítógépeken hajtódik végre. Az igényelt szolgáltatásra adott válaszidő megjósolhatatlan, ezért úgy kell megtervezni a rendszert, hogy a szolgáltatást igénylő objektumnak ne kelljen megvárni a szolgáltatás befejeződését. A szerverek persze egyedi

gépen is használhatók, ahol a szolgáltatás befejeződéséhez némi időre van szükség, pl. nyomtatás és a szolgáltatást több különböző objektum is igényelheti.

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

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

8.2.2 Objektumok élettartalma

A megvalósítandó objektum belső állapotát az attribútumainak pillanatnyi értéke határozza meg. Ezeket az adatokat az objektum élete során tárolni kell. A háttértárolón azon objektumok adatait kell tárolni, amelyek élettartama hosszabb, mint a program futási ideje. Ezeket **perzisztens** objektumoknak nevezzük. (Azokat az objektumokat pedig, amelyek élettartama a program futási idejénél nem hosszabb, **tranziens**-nek nevezzük.) Ha a program nagyszámú perzisztens objektummal dolgozik, érdemes egy adatbázis-kezelő rendszerrel kiegészíteni. A relációs adatbázisokat nagyszámú, de viszonylag kevés osztályhoz tartozó objektum tárolására érdemes igénybe venni, egyébként érdemesebb objektum-orientált adatbázis-kezelőt használni.

9. Gyors szoftverfejlesztés

A vállalatok ma globális, gyorsan változó környezetben működnek. Reagálnak az új lehetőségekre és piacokra, a gazdasági környezet változásaira. A szoftver része minden műveletnek, így kulcsfontosságú hogy egy új szoftvert gyorsan kifejlesszenek, kihasználva ezzel az új lehetőségek adta előnyöket. A szoftverrendszereknél így ma a legkritikusabb követelmény a gyors fejlesztés és üzembe helyezés. Sok vállalat hajlandó elengedni a minőségből és kompromisszumot kötni a szoftver gyors üzembe helyezése érdekében.

Mivel ezek a vállalatok folyamatosan változó környezetben tevékenykednek, sokszor gyakorlatilag lehetetlen, hogy a szoftverrel szemben stabil elvárásokat fogalmazzunk meg.

Ezek alapján nem tartoznak a gyors szoftverfejlesztéshez azok a szoftverfejlesztési folyamatok, amelyek az elvárások teljes specifikációján alapulnak. Az elvárások változásával szükségessé válhat a rendszerterv, illetve az implementáció átdolgozása.

Ez gyorsan változó üzleti környezetben komoly gondokat okozhat. Idővel a szoftver elkészül, de addigra a körülmények megváltozása miatt gyakorlatilag használhatatlanná válik. Ez idézte elő az olyan fejlesztési folyamatokat, amelyek a gyors szoftverfejlesztésre és átadásra összpontosítanak.

A gyors szoftverfejlesztési folyamatokat arra tervezték, hogy segítségükkel gyorsan készíthessünk használható szoftvereket. Ez általában olyan iteratív folyamat, ahol a specifikáció, a tervezés, a fejlesztés és a tesztelés átfedi egymást. A gyors szoftverfejlesztés alapvető jellemzői:

1. A specifikáció, a tervezés és az implementálás folyamata konkurens módon zajlik. Nincs részletes rendszer-specifikáció és a tervezési dokumentáció minimális. A felhasználói elvárások leírása csak a rendszer legfontosabb jellemzőit határozzák meg.
2. A rendszert lépésről lépésre fejlesztik. A végfelhasználók és a rendszer többi érintettje részt vesznek minden lépés specifikációjában. Indítványozhatnak változásokat a szoftverben és új követelményeket fogalmazhatnak meg.
3. A rendszer felhasználói felülete gyakran egy beépített fejlesztői környezet használatával készül. Ez gyors interfész elkészítést és gui elemek elrendezését eredményezi.

9.1 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 szemben támasztott 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 képviselik, akik nagyszámú különálló programból felépülő hosszú élettartalmú szoftverrendszerek fejlesztésével foglalkoznak.

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. Viszont ahogy a rendszerkövetelmények változnak elengedhetetlenül szükség van az átdolgozásra.

Ezzel az időigényes megközelítéssel való elégedetlenség vezetett oda, hogy a szoftverfejlesztők egy része az 1990-es években új, **agilis** (jelentése gyors) **módszereket** indítványozott. **Az indítvány célja az volt, hogy a fejlesztőcsapat magára a szoftverre koncentráljon, ne pedig annak tervezésére és dokumentálására**.

9.2 Az Agilis Kiáltvány

Az Agilis Fejlesztés egy módszertan-család, és nem egy konkrét megközelítése a szoftverfejlesztésnek. 2001-ben 17 ember, az akkor „*pehelysúlyú módszertanok*” néven

futó módszertanok jelentős képviselői, összeültek, hogy megbeszéljék, mi a közös a módszertanaikban. Megalkották az **Agilis Kiáltványt**, amit széles körben elfogadtak, mint az agilis fejlesztés kanonikus meghatározását.

Mi is az Agilis Kiáltvány?

A szoftverfejlesztés jobb módjait fedezzük fel azáltal, hogy csináljuk, és segítünk másoknak is csinálni. Ennek során az alábbi hangsúly-eltolódásokat találtuk:

- **Egyének és interakcióik, szemben az eljárásokkal és eszközökkel.**
- **Működő szoftver, szemben a teljeskörű dokumentációval.**
- **Együttműködés az ügyféllel, szemben a szerződésről való alkudozással.**
- **Változásokra való reagálás, szemben a terv követésével.**

Ez azt jelenti, hogy a jobb oldalon szereplő értékek is fontosak, de a bal oldalon lévőket fontosabbnak tartjuk.

Az agilis fejlesztési módszertan létrejöttét tehát a változás igénye idézte elő. Mégpedig az, hogy sokakban megfogalmazódott, hogy a **szoftverfejlesztés nem gyártás**. Bár napjainkban sok cég rá van kényszerítve külső nyomás, vagy más gazdasági érdekek miatt arra, hogy szinte futószalagon adja ki a dobozos szoftvereket (Pl. játékfejlesztés). Szükség van a változások gyors és rugalmas adaptálására és a arra, hogy megszabaduljanak a klasszikus módszerek hibáitól. **A cél:** Minél gyorsabban, minél költséghatékonyabban történjen a fejlesztés és az elvárt igényt minél jobban kielégítő végeredmény szülessen.

9.3 Az agilis fejlesztés működése

Az **agilis szoftverkészítés** (agile sw developement) egy elméleti keretrendszer. Többféle agilis fejlesztési eljárás van, de jórészt mindegyik a kis kockázatú fejlesztésre törekszik a rövid idejű fejlesztési ciklusok (ismétlések, iterációk) használatával. Minden ismétlés egy teljes szoftverfejlesztési ciklus: tervezés, követelményelemzés, kivitelezés, kódolás, tesztelés és dokumentálás. Egy ismétlés célja, hogy a ciklus végére egy letesztelt, elérhető kiadás jöjjön létre az alkalmazásból. Minden iteráció végén a fejlesztői csapat újraértékeli a projekt főbb céljait. Az agilis módszerek a szemtől-szembeni kapcsolatot részesítik előnyben az írásos dokumentációk helyett. Emiatt az agilis csoportok jellemzően egyetlen nagy irodában helyezkednek el (pl. scrum). Az agilis módszereknél a haladás legfőbb mérőszáma a működő szoftver.

Az agilis módszerek alapelvei:

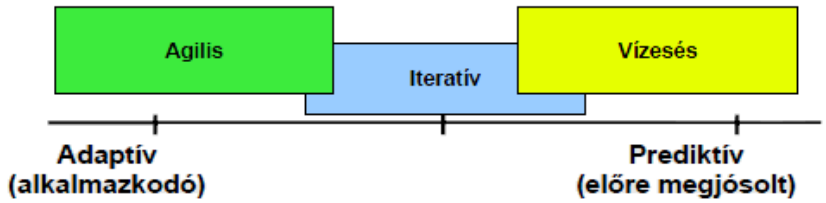
1. Hasznos szoftvertermékek gyors, folyamatos szállításából fakadóan elégedett megrendelők.
2. Működő szoftver szállítása gyakran (inkább hetes, mint havi periódusban)
3. Az előrehaladás mércéje a működő szoftver

4. A követelményekben még a késői változásoknak is örülnek.
5. Szoros, napi kommunikáció a fejlesztők és a megrendelő között
6. Személyes kapcsolattartás
7. A projekteket motivált, megbízható munkatársak vezetik
8. Folyamatos figyelem kíséri a műszaki színvonalat és a tervet
9. Egyszerűség
10. Önszerveződő csapatmunka
11. A változó körülményekhez való gyors alkalmazkodás

Agilis módszertan használata javasolt kis projektméret, kevés (de tapasztalt) fejlesztő esetén, alacsony kritikussági foknál, illetve gyakran változó követelmények esetén. Ellenjavallt a használata nagy projekteknél, elosztott fejlesztéseknél, létfontosságú projektekben.

9.4 Összehasonlítás más típusú módszertanokkal

Az Agilis módszertanokat a „terv-vezéret” vagy „fegyelmezett” módszertanok ellentétének szokták nevezni. Ez félrevezető, mert úgy hangzik, mintha az Agilis módszertanok „tervezetlenek” vagy „fegyelmezetlenek” lennének. Szerencsésebb megkülönböztetés, ha a számegyenes két végpontjának az „adaptív” és a „kiszámítható” módszertanokat tekintjük. Ebben az esetben az Agilis módszertanok a számegyenes „adaptív” végén lesznek.



1. ábra: A módszertanok számegyenese

Az adaptív módszertanok arra fókuszálnak, hogy a gyakran változó követelményekhez tudjanak alkalmazkodni. Ha egy projektben megváltoznak az igények, akkor egy adaptív csapat képes alkalmazkodni a változásokhoz. Egy adaptív csapat nehezen tudja megmondani, hogy mi fog történni a jövőben. Minél távolabbi pontról van szó a jövőben, annál bizonytalanabb lesz az adaptív módszertan elképzelése arról, hogy mi is fog akkor történni. Ha egy nagyon távoli időpontról van szó, akkor a csapat már csak a vállalat célkitűzéséről, vagy a tervezett költség-érték arányról tud beszámolni.

A kiszámítható módszertanok ennek ellenkezőjeként, arra fókuszálnak, hogy minél részletesebben megtervezzék a jövőt. Egy kiszámítható csapat pontosan meg tudja mondani bármelyik pillanatban, hogy mikor milyen feladatok és feature-ök lesznek készen a projektben. A prediktív csapatok nehezen váltanak irányt. A terv általában az eredetileg meghatározott cél elérésére van optimizálva, és az irányváltoztatás könnyen azzal a következménnyel járhat, hogy az elkészült munkát el kell dobni, és újrakezdeni. A kiszámítható csapatok gyakran változáskezelő bizottságot állítanak fel (change control board), hogy biztosítsák, hogy csak a legszükségesebb változások érintsék a projektet.

A legtöbb Agilis módszertan egyetért az iteratív fejlesztéssel abban, hogy a szoftvert rövid időközönként ki kell adni. Az Agilis fejlesztésben azonban ez a rövid időköz inkább hetekben mérhető, mintsem hónapokban. A legtöbb Agilis módszertan továbbá szigorú időkeretként tekinti ezt az időközt, és nem pedig tervezett célként. Az időkeret azt jelenti, hogy a határidő kőbe van vésve, és nem változhat. Ha a csapat kicsúszik a határidőből, akkor a feladatot nem sikerült megoldani, és vagy vissza kell vonni, vagy új feladatot csinálni belőle. Van olyan is, hogy a feladatot lehet egyszerűsíteni, hogy a csapat beleférjen a határidőbe.

A vízesés-modellel már kevesebb közös vonása van az Agilis technikáknak. A vízesés a legkiszámíthatóbb modell (elvileg). A haladást általában eredményként felmutatható dolgok formájában mérik - követelmény-specifikáció, terv-dokumentumok, tesztterv, kódellenőrzési jegyzőkönyvek, és effélék. A vízesés modell néha azon csúszik meg, hogy a ciklus végén komoly integrálási és tesztelési problémák jelentkeznek, és ez a munka eltarthat néhány hónapig vagy évig is. Ez gyakran okozza a modell bukását. Az Agilis módszertanok ellenben teljesen integrált és tesztelt programokat eredményeznek, de mindig csak egy lépést tesznek előre, hétről hétre. A hangsúly egy elnagyolt, de működő rendszer korai kifejlesztésén van, amit aztán lehet finomítgatni.

Sokan használják a „**cowboyos kódolás nevű módszert**” is. A cowboyos kódolás lényege, hogy nincs semmiféle módszer meghatározva, mindenki azt csinálja, amit jónak lát. Az Agilis módszertanok gyakori újratervezése, szemtől-szembe kommunikációja, és viszonylag laza dokumentumkezelése miatt sokan azt hiszik, hogy ez is cowboyos kódolás. Az Agilis csapatok azonban nagyon is használják a maguk jól meghatározott (gyakran fegyelmezett és rigorózus) módszertanját, tehát éles különbség van az Agilis kódolás és a cowboyos kódolás között.

Kritikák a módszerrel szemben:

Rengeteg kritika olvasható ezekről a módszertanokról, aminek részben az az oka, hogy ezek többségükben **még nem kiforrott**, **lezárt módszertanok**, ráadásul mindenki a maga szája íze szerint értelmezi őket. A jogosnak elismert kritikák három központi érv köré gyűlnek:

1. Csak gyakorlott, szenior fejlesztőkkel működik.
2. Nincs eléggé megtervezve a szoftver.
3. Túl sok kulturális változás kell ahhoz, hogy jól működjön.

Következésképpen az agilis módszereket olyan szerződésekre kell alapozni, ahol az ügyfél a rendszerfejlesztéshez szükséges időért fizet egy bizonyos követelmény helyett. Ha minden jól megy, akkor ez hasznot hoz mind az ügyfélnek, mind a fejlesztőnek. Az agilis módszertan nem alkalmas nagyobb kaliberű rendszerek fejlesztésére, ahol a fejlesztők különböző helyen dolgoznak, és ahol komplex kölcsönhatások léphetnek fel más hardver- és szoftverrendszerekkel. A módszer nem ajánlott kritikus rendszerek fejlesztésére sem. Az agilis szemlélet leginkább kis- és középvállalkozások, rendszereinek, illetve a személyi számítógépes termékek fejlesztésére a legalkalmasabb.

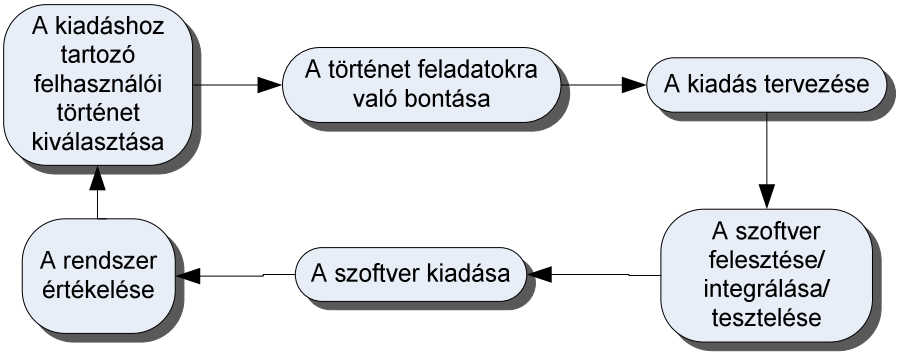
9.5 Az Extrém programozás

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

Az XP fő célja, hogy csökkentse a változások költségvonzatát. A hagyományos rendszerfejlesztési módszertanokban (pl. SSADM), a rendszerrel szemben támasztott követelmények adottak a projekt elején, és gyakran nem is változnak meg. Ez azzal jár, hogy minél később kell változtatni a követelményeken, ami pedig szoftverfejlesztési projektekben a legvégén sem szokatlan, annál magasabbak lesznek a költségek.

Az XP arra törekszik, hogy ezeket a költségeket csökkentse azáltal, hogy más alapvető értékeket, elveket, és gyakorlatot vezet be. Egy XP-t használó rendszerfejlesztési projekt sokkal rugalmasabb lesz a röptében bekövetkező változásokkal szemben.

Az extrém programozásban minden követelményt forgatókönyvként állítanak össze, amely közvetlenül feladatok soraként kerül implementálásra. A programozók párokban dolgoznak, és mindenre teszteket készítenek, még mielőtt megírnák a kódot. Minden tesztnek sikeresen le kell futnia, mielőtt az új kódot elhelyeznék a rendszerben. A rendszer kiadásai között csak kis idő telik el. A következő ábra az XP folyamatát mutatja be:



5. ábra: Az extrém programozás kiadási ciklusai

Az extrém programozás megfelel az agilis módszerek alapelveinek:

1. Az inkrementális fejlesztés a rendszer kisméretű, gyakori kiadásán keresztül valósul meg.
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.
3. A párban való programozás, a rendszerkód fölötti együttes tulajdonjog az embereket támogatja nem 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.

Az 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. Közösen kidolgoznak egy „történetkártyát”, ami magában foglalja az ügyfél kéréseit. A csapat ezután megpróbálja implementálni a forgatókönyvet a szoftver egy jövőbeli kiadásában.

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. Persze a követelmények változásával a nem implementált történetek változhatnak vagy elhagyhatók. Ha változások szükségesek egy olyan rendszerben, amelyet már átadtak, új történetkártyákat kell készíteni, és megint csak az ügyfél dönti el, hogy ezen a változásoknak elsőbbségük van-e az új funkcionalitással szemben vagy sem.

Az XP az iteratív fejlesztés „extrém” megközelítését használja. Naponta többször megjelenhet a szoftver új verziója, de az inkremensek nagyjából kéthetente kerülnek az ügyfélnek. Amikor létrehozzák a rendszer egy új verzióját, akkor minden létező automatizált teszten végig kell futtatni, beleértve az új funkcionalitásra készült teszteket is. Az új verzió csak akkor elfogadható, ha az összes teszt sikeresen lefutott.

Az XP figyelmen kívül hagyja a szoftvertervezés azon alapelvét, hogy a változtatásokra kell tervezni. Az XP szerint ez időpocséklás, mert gyakran nem a várt változtatások, hanem teljesen más követelmények valósulnak majd meg. Az előre nem várt változtatások implementálásával az a probléma, hogy ronthatják a szoftver struktúráját, ami miatt a megvalósíthatóságuk egyre nehezebbé válik. Az XP úgy oldja meg ezt a problémát, hogy a szoftver folyamatos kódátszervezését javasolja, ami azt jelenti, hogy a programozó csapat tökéletesítési lehetőségeket keres a szoftverben, és azokat azonnal implementálja. Így a szoftver mindig érthető és könnyen változtatható marad az új történetek implementálását követően is.

9.6 Tesztelés az XP-ben

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. A tesztelés kulcsjellemzői a következők:

1. Előrehozott teszteléssel történő fejlesztés
2. Inkrementális tesztfejlesztés a forgatókönyvek alapján
3. Felhasználók bevonása a tesztek fejlesztésébe és validálásába
4. Automatizált tesztelő eszközök használata.

Az XP egyik legfontosabb újítása az **előrehozott teszteléssel történő fejlesztés (Test-First Development)**. Az előre elkészített tesztek mind az interfészt, mind a viselkedési specifikációt meghatározza a fejlesztendő funkcionalitás számára. Ez csökkenti a követelményproblémák és interfészek félreértéseit. Ez a folyamat minden olyan folyamat esetén alkalmazható, ahol tisztán látható a rendszerkövetelmény és az implementálandó kód közötti kapcsolat. Az extrém programozásban mindig látható ez a kapcsolat, mert a követelményeket reprezentáló történetkártyák feladatokra vannak bontva, amelyek egy implementációs egységet képeznek.

Korábban említettük, hogy a felhasználói követelmények forgatókönyvekként vagy történetekként jelennek meg, és a felhasználó rangsorolja ezeket a fejlesztéshez. A

fejlesztőcsapat feladatokra bontja ezeket. Minden egyes feladathoz egy vagy több különálló egységteszt tervezhető, amely a feladatban leírt implementációt ellenőrzi.

Az ügyfél szerepe a tesztelési folyamatban az, hogy a történetekhez segítsen elfogadási teszteket fejleszteni, amelyeket a rendszer következő kiadásában kell implementálni.

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. A gyakorlatban az elfogadási teszek egész sorát szokták készíteni.

Az előrehozott teszteléssel történő fejlesztés és az automatikus teszteszközök használata az XP-irányzat fontos erősségei. Az előrehozott tesztelés azt jelenti, hogy a tesztelést, mint futtatható komponens hamarabb megírjuk, mint a feladatot. Így amint kész a szoftver implementációja, rögtön futtatható rajta a teszt. A tesztelő komponensnek függetlennek kell lennie, azaz szimulálnia kell a tesztelendő bemenetek megadását, és ellenőriznie kell, hogy a végeredmény megfelel-e a kimenet specifikációjának. Az automatikus teszteszköz olyan eszköz, amely elküldi ezeket az automatikus teszteket végrehajtásra.

Az előrehozott teszteléssel történő fejlesztés esetében a feladat implementálóinak átfogóan kell érteniük a specifikációt, hogy teszteket készítsenek a rendszerhez. Ez azt jelenti, hogy a specifikációban lévő félreérthetőségeket és hiányosságokat még az implementáció megkezdése előtt tisztázni kell. A módszer kikerüli a „*lemaradt teszt*” problémáját, ami akkor léphet fel, amikor a rendszer fejlesztője nagyobb léptékben dolgozik, mint a tesztelő, így az implementáció egyre jobban megelőzi a tesztelést. Így néha kihagyhatnak teszteket, hogy tartani tudják az ütemtervet.

Mindamellett az előrehozott teszteléssel történő fejlesztés nem mindig működik úgy, ahogy várnánk, mert a programozók jobban szeretnek programozni, mint tesztelni. Emiatt sokszor hiányos teszteket írnak. Vagy előfordulnak olyan esetek is, amikor a teszt megírása sokkal nehezebb, mint magának a feladatnak a megírása.

9.7 A Test-First development előnyei

Mint látni fogjuk, a TFD-nek számos előnyös tulajdonsága van, amit sajnos még ma sem minden fejlesztő ismer fel.

- **A tesztek csökkentik a bug-ok számát az új funkciókban.**
  Mivel a teszteket hamarabb írjuk, mint a tényleges programkódot, egy esetleges elgépelés nagy valószínűséggel el fog rontani néhány tesztet.

- **A tesztek jó dokumentációk.**
  Megfigyelések bizonyítják, hogy egy programozó sokkal hamarabb megérti egy programegység funkcióját egy szemléletes programkódból, mint folyamatos szövegben írt dokumentáció alapján.

- **A tesztek korlátozzák az osztályok feladatait.**

A kódolás során csak annyit kódolunk, amennyi a tesztek kielégítéséhez szükséges. Ennél fogva ami nem szerepel a tesztben, annak nem kell megjelennie a programkódban sem. Nem szükséges keretrendszereket és a későbbi bővítés számára fenntartott csatlakozási pontokat beiktatni a kódba. Törekedjünk inkább egyszerű design-ra, ami minden feladatot ellát.

- **A tesztek javítják a programkód minőségét.**
  A TFD miatt már kezdettől fogva úgy tervezzük a programunkat, hogy az tesztelhető legyen. Ezt csak jól tagolt, moduláris kóddal érhetjük el, aminek további hasznos következményei vannak.

- **A tesztek megvédnek a bug-ok újra bevezetésétől.**
  Ha a programban bug-ot találunk, akkor először megismételtetjük az egy új, eddig a tesztgyűjteményben még nem szereplő teszttel, majd kijavítjuk a hibát. Ha valaki módosítja a kódunkat, és újra bevezeti ezt a bug-ot, akkor a teszt azonnal jelezni fog.

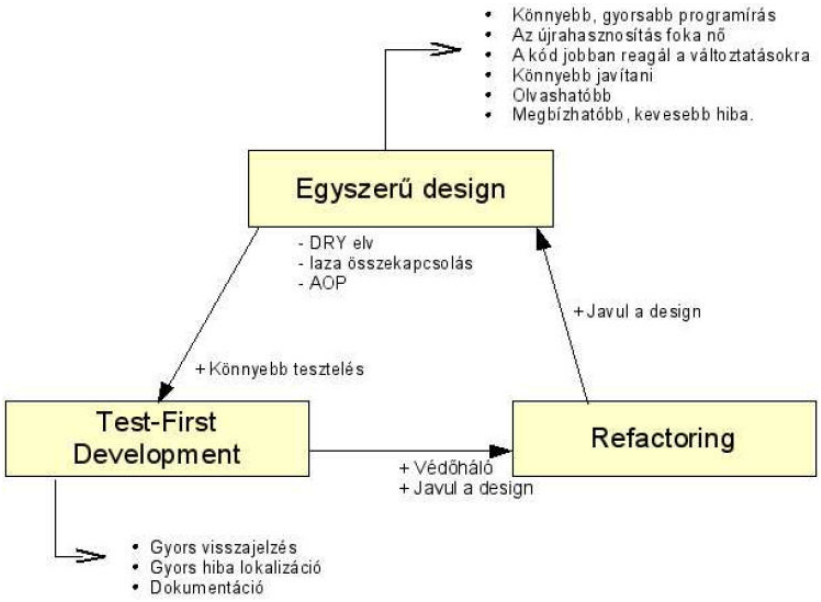
- **Felgyorsul a fejlesztés.**
  A kódolást tekintve a tesztkészítés lelassítja a programozót: a tesztesetek megfogalmazása, elkészítése időbe telik. Az egész fejlesztés sebessége viszont felgyorsul, mert kevesebb időt töltünk hibakereséssel, és az új funkciók implementálása is gyorsabban megy, mert nem kell állandóan azon aggódni, hogy elrontjuk a már kész kódot.

- **A tesztek csökkentik a félelemérzetet.**
  A programozók legnagyobb rémálma az, amikor egy jól működő funkcióba vezetünk be bug-ot, és az csak hetek múlva jelenik meg. A jó minőségű tesztgyűjtemény mint egy „védőháló” vigyáz a programozóra, és az esetek 90%-ában jelez, ha megbontottunk egy már jó funkciót. A programozó így nem fog habozni, ha egy jól tesztelt kódot kell módosítania.

9.8 Test-Driven Development

A Test-First fejlesztés csak azt követeli meg, hogy minden teszt sikeresen fusson le. A Test-Driven fejlesztés viszont megköveteli, hogy minden funkció implementálása után gondoljuk át, lehet-e egyszerűsíteni a rendszeren. Ha erre a kérdésre igen a válasz, akkor a refactoring technikájával ezt oldjuk is meg. Ha szigorúan csak a TFD elveihez ragaszkodnánk, a programunk a funkciók bővítése során egyre kaotikusabbá és bonyolultabbá válna, mivel semmi mással nem törődünk, csak azzal, hogy a teszteket kielégítsük. A refactoring lépés pontosan ezt akadályozza meg, így folytonosan be tudjuk tartani „egyszerű design” elvét.

Ez folyamat ráadásul egy pozitív visszacsatolási folyamat, azaz amellett hogy a rendszer önfenntartó, további artifact-öket termel. Az egyes lépések támogatják a soron következőt, és ez az, ami a Test Driven Development jelentőségét adja. A folyamatot a következő ábrán láthatjuk.



6. ábra: Test Driven Development folyamata

Az ábrán látható, hogy egy pozitív visszacsatolási rendszerről van szó, ugyanis:

1. Az új funkciók egyszerű teszteléséhez és implementálásához elengedhetetlen az egyszerű design.
2. Az egyszerűség eléréséhez a refactoring-ot alkalmazzuk.
3. A refactoring viszont elképzelhetetlen az automatizált tesztek nyújtotta „védőháló" nélkül.
4. Ezen túl a TDD komponenseinek további, a szoftverfejlesztő számára fontos haszna is van. (Pl. a gyors visszajelzés, öndokumentáló, tiszta kód, stb.)

Ha bármit is kiveszünk a TDD-ből, a kör megbomlik, és szertefoszlik a módszerminden előnye.

9.9 Az XP ellentmondásos részei

A legellentmondásosabb, legproblémásabb része a dolognak a változás-menedzsment. Mivel az ügyfél közvetlenül kommunikál a programozókkal, leginkább szóban, így két rossz dolog történhet. Az egyik, hogy az összevissza csapongó változásai költséges átdolgozásokhoz vezetnek, a másik az ún. „becsúszó featuritis", vagyis hogy az ügyfél szép lassan egyre többet és többet követel. Az XP azért nem szereti lepapírozni az ügyfél kéréseit, mert azt mondják, hogy ez a rugalmasság kárára megy, és az esetek túlnyomó részében tovább tart a papírozás, mint maga a munka.

Az ügyfél képviselője szerves része a projektnek. Ezt sokaknak stresszt okoz, hiszen minden hiba és tévedés azonnal nyilvánvaló az ügyfél számára is. Ugyanakkor, ha az ügyfél képviselője rosszul végzi a munkáját, akkor azon az egész projekt megbukhat. Ugyanakkor, ha az ügyfél nincs jelen, általában mégis mindenki azt kívánja, hogy bárcsak jelen lenne.

10. Verifikáció és validáció

Az implementációs folyamat közben és után az éppen fejlesztett programot ellenőrizni kell, hogy megfelel-e megfelel a specifikációnak és kielégíti a megrendelő igényeit. Az ellenőrzőfolyamatok neve ***verifikáció*** és ***validáció***. Ezek a folyamatok a fejlesztés minden lépésében jelen vannak.

> **Validáció:** A megfelelő terméket készítetjük el?
> **Verifikáció:** a terméket jól készítetjük el?

A verifikáció magába foglalja annak ellenőrzését, hogy a szoftver megfelel-e a specifikációnak, azaz eleget tesz-e a funkcionális és nem funkcionális követelményeknek. Általánosabban: a szoftver megfelel-e a vásárló elvárásainak.

A validáció ennél kicsit általánosabb fogalom. Végcélja az, hogy megbizonyosodjunk arról, hogy a szoftverrendszer „megfelel-e a célnak”. Azaz teljesül-e a vásárló elvárása, amibe beleértendő olyan nem funkcionális tulajdonságok is, mint a hatékonyság, hibatűrés, erőforrásigény.

A V & V folyamaton belül a rendszer ellenőrzésére két egymást kiegészítő technika használható:

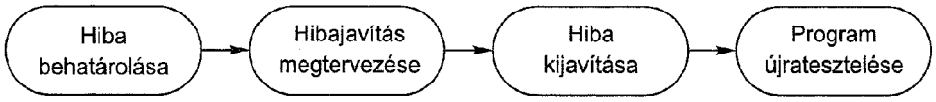
- **Statikus:** szoftverátvizsgálások, amelyek a rendszer reprezentációját elemzik: Követelmény dokumentumot, terveket, és forráskódot.
- **Dinamikus:** szoftvertesztelés, amely csakis az implementáció fázisában végezhető el. A tesztadatok segítségével ellenőrzi, hogy az megfelelő teljesítményt nyújt-e.

Az átvizsgálási folyamat a szoftverfolyamat bármely lépésében használhatók. A követelményekkel kezdődően minden olvasható reprezentáció átvizsgálható. Az átvizsgálási technikák közé tartoznak a *programátvizsgálások*, az *automatizált forráskódelemzés* és *formális verifikáció*. A rendszert csak akkor tesztelhetünk ha elkészült egy végrehajtható változatának prototípusa. Az inkrementális fejlesztés előnye, hogy a különböző inkremensek külön tesztelhetők, és az egyes funkciók már hamarabb tesztelhetők.

A szoftvertesztelés, mint dinamikus V & V a gyakorlatban inkább alkalmazott technika. Ekkor a programot a valós adatokhoz hasonló adatokkal teszik próbára. A kimenetek eredményei lehetőséget adnak anomáliák, problémák feltárására. Ezen tesztelés két fajtája ismert:

- **Hiányosságtesztelés:** célja a program és a specifikációja között meglévő ellentmondások felderítése. Az ilyen teszteket a rendszer hiányosságainak feltárására tervezik, nem a valós működés szimulálására.
- **Statisztikai (vagy validációs) tesztelés:** a program teljesítményének és megbízhatóságának tesztelése valós körülményeket szimulálva. Annak megmutatása, hogy a szoftver megfelel-e a vásárlói igényeknek.

A V & V folyamatokat gyakran a belövési folyamattal ötvözik. **A belövés az a folyamat, amely behatárolja és kijavítja a szoftverben talált hiányosságokat.** A folyamata az alábbi ábrán látható.



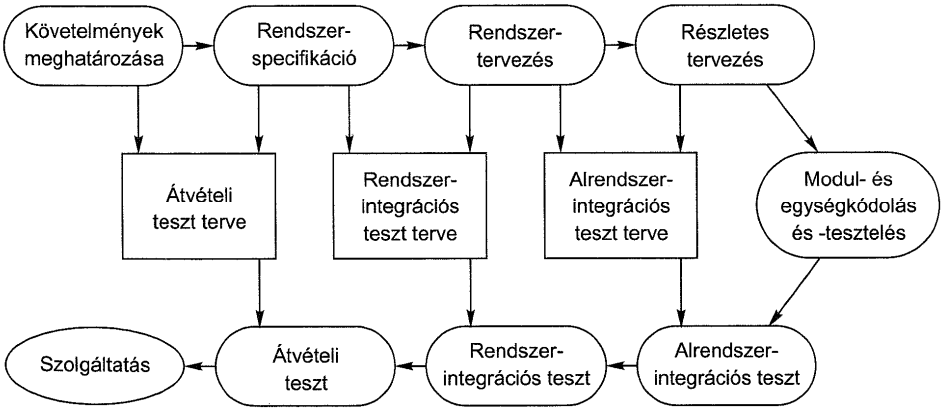
18. ábra. A belövés folyamata

A belövésre nem létezik egyszerű módszer, mert a hibák behatárolása a programban nem mindig könnyű, hiszen a hiba nem mindig ott található meg, ahol a sikertelenség bekövetkezett. Bizonyos esetekben szükség lehet új tesztek megtervezésére, amelyek megismétlik az eredeti hibát és interaktív belövőeszközök (pl. Debugger) használatára.

Bármely a rendszerben talál hiba után az összes tesztet meg kell ismételni, azonban a gyakorlatban ezt nem alkalmazzák, mert nagyon költséges.

10.1 Verifikáció- és validációtervezés

A V & V költséges folyamat, bonyolult valós idejű rendszereknél akár a költségek 50%-át is elérheti, ezért gondosan meg kell tervezni. **Egy szoftverrendszer verifikációjának és validációjának tervezését még a fejlesztési folyamat elején el kell kezdeni.** Az alábbi ábrát V modellnek is szokás nevezni, amely megmutatja, hogy hogyan kapcsolják össze a tesztterveket a tesztelési és fejlesztési tevékenységeket.



. ábra. V Modell

A V & V tervezési folyamatoknak egyensúlyt kell kialakítani a verifikáció és a validáció statikus és dinamikus technikái között.

10.2 Statikus technikák

A szoftver átvizsgálása

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

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

A programkód átvizsgálását egy legalább 4 emberből álló csapatnak érdemes végeznie. A szerző, az olvasó, a tesztelő és a moderátor. A csapat tagjai szisztematikusan elemzik a kódot és rámutatnak az esetleges hibákra. Maga az átvizsgálás max. 2 óra, amely során az olvasó felolvassa a kódot. Végül a szerző módosítja a programot. Ezután vagy újra átvizsgálják a kódot, vagy a moderátor úgy dönt, hogy ez nem szükséges. Az átvizsgálás folyamatát a szokásos programozási hibák – nyelvtől függő – listájára kell alapozni.

A szoftver átvizsgálás eredményessége ellenére sok szoftverfejlesztő cégnél nehéz bevezetni, mert a tesztelésben jártas szoftvertervezők vonakodva fogadják el a módszer hatékonyságát.

Automatizált statikus elemzés

A statikus programelemzők olyan szoftvereszközök, amelyek a program forrásszövegének vizsgálatával derítik fel a lehetséges hibákat és anomáliákat. Ehhez nincs szükség a program futtatására. Észlelhetik, hogy az utasítások formailag helyesek-e, a nem használt kódrészleteket, inicializálatlan változókat, stb. Kiegészítik a nyelv fordítóprogramja által adott lehetőségeket.

Néhány statikus elemzéssel felismerhető hibafajta és anomália (az anomália nem feltétlenül programhiba, lehet következmény nélküli is): Adathibák, Vezérlési hibák, Input/output hibák, Interfészhibák.

11. Szoftvertesztelés

Korábban már áttekintettük a szoftvertesztelés, más néven verifikáció és a validáció általános eljárását, amely a különálló programegységek, mint például függvények vagy objektumok tesztelésével kezdődik. Ezek később alrendszerekbe és rendszerekbe integrálódnak, majd az egységek közötti interakciók kerülnek tesztelésre. Végül, miután a rendszer elkészült, a vásárló végrehajthat egy elfogadási tesztsorozatot annak ellenőrzésére, hogy a rendszer teljesíti-e az előírásokat.

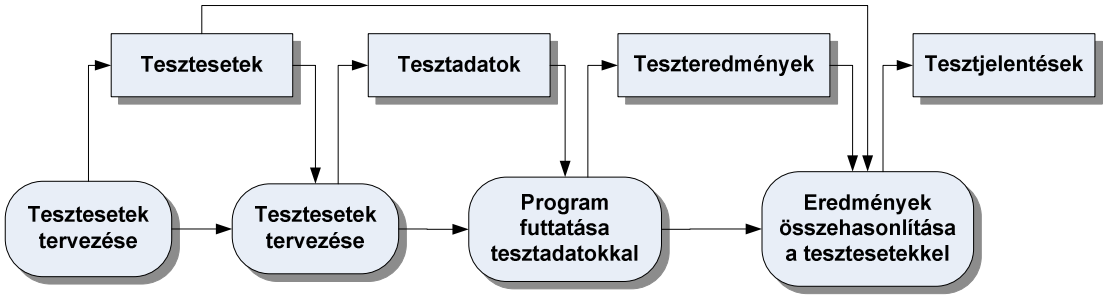
A szoftvertesztelésnek tehát két célja van:

- **Bizonyítani a fejlesztő és a vásárló számára, hogy a szoftver megfelel a vele szemben**

támasztott követelményeknek. Egyedi szoftver esetén ez azt jelenti, hogy a felhasználói és rendszerkövetelményeket leíró dokumentum minden követelményére vonatkozóan legalább egy tesztnek lennie kell. Általános szoftver esetén pedig minden egyes rendszertulajdonságra kell lenni egy tesztnek.

- **Felfedezni a szoftverben azokat a hibákat és hiányosságokat, amelyekben a szoftver viselkedése helytelen, nemkívánatos, vagy nem felel meg a specifikációnak.** A hiányosságtesztben felfedett hibák kiirtásával foglalkozik.

A tesztelés folyamatának általános modellje látható következő ábrán.



. ábra. A szoftvertesztelési folyamat modellje

A teszteset nem más, mint a teszthez szükséges inputok és a rendszertől várt outputok specifikációja. A tesztadatok kifejezetten a rendszer tesztelésére létrehozott inputok. A tesztadatok néha automatikusan generálhatók, az automatikus teszteset-generálás viszont lehetetlen. A tesztek outputjait csak azok tudják előre megjósolni, akik értik, hogy a rendszernek mit kellene csinálnia.

Beszélhetünk **kimerítő tesztelés**ről is, ahol az összes lehetséges program-végrehajtási szekvenciát teszteljük. A gyakorlatban nem praktikus, ezért a tesztelésnek a lehetséges tesztesetek egy részhalmazán kell alapulnia. Erre irányelveket kell kidolgozni a szervezetnek, nem pedig a fejlesztőcsoportra hagyni.

11.1 Rendszertesztelés

A rendszertesztelés két vagy több, rendszerfunkciót vagy jellemzőt megvalósító komponens integrálását és az integrált rendszer tesztelését jelenti. A rendszertesztelés iteratív fejlesztési folyamatban a vásárló számára leszállítandó inkremens, míg vízesés jellegű folyamat során a teljes rendszer tesztelését jelenti.

A legtöbb bonyolult rendszer esetében a rendszertesztelést két külön fázisra bonthatjuk:

1. ***Integrációs tesztelés:*** ahol a tesztelést végző csapat hozzáfér a rendszer forráskódjához. Leginkább a rendszer hiányosságainak megtalálásával foglalkozik.

2. ***Kiadástesztelés:*** ahol a rendszer egy felhasználók számára kiadható verziója kerül tesztelésre. Itt a tesztcsapat azt vizsgálja, hogy a rendszer megfelel-e a követelményeknek.

A kiadástesztelés általában fekete doboz tesztelés, ahol a teszt csapatnak egész egyszerűen csak azt kell bemutatnia, hogy a rendszer jól működik-e, avagy sem. Ha ebbe a kiadástesztelésbe a vásárlót is bevonják, akkor ezt ***elfogadási tesztelésnek*** nevezzük. Ha a verzió elég jó, a vásárló elfogadhatja használatra.

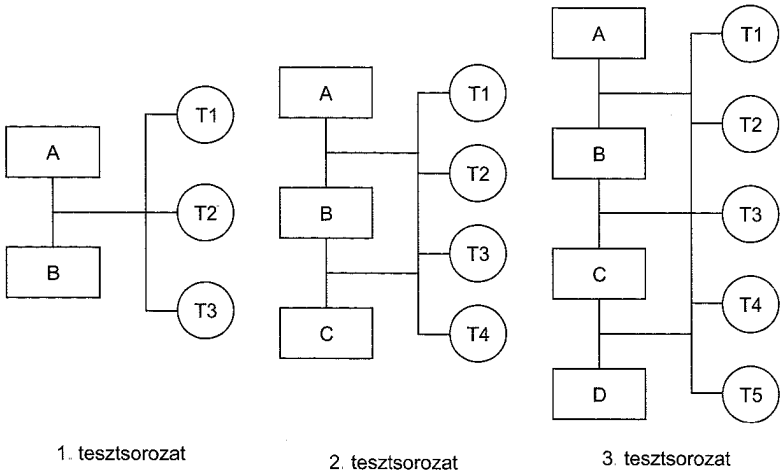
Az integrációs tesztelésre alapvetően úgy kell tekintenünk, mint rendszerkomponensek egy csoportjából vagy fürtjéből álló befejezetlen rendszerek tesztelésére. A kiadástesztelés a rendszer azon kiadásának tesztelésével foglalkozik, amelyet le szeretnénk szállítani a vásárlóknak.

11.1.1 Intergrációs tesztelés

A rendszer-integráció folyamata magában foglalja a rendszer komponenseiből történő felépítését és az eredményül kapott rendszer tesztelését a komponensek együttműködéséből adódó problémák felderítésére. Az integrációs tesztelés azt ellenőrzi, hogy ezek a komponensek valóban képesek-e együttműködni, megfelelően vannak-e meghívva és interfészeiken keresztül a megfelelő adatokat a megfelelő időpontban küldik-e át.

A rendszer-integráció során azonosítani kell a rendszer különböző funkcionalitásait biztosító komponensek csoportjait, majd további kód hozzáadásával integrálni kell őket. Sok esetben elsőként a rendszer váza készül el, és ehhez adódnak hozzá a komponensek. Ezt ***fentről lefelé*** történő integrációnak nevezzük. Más esetekben először a gyakran a közös szolgáltatásokat (mint a hálózati és az adatbázis-elérés) biztosító komponensek integrálását végezzük először, majd hozzáadjuk a funkcionális komponenseket. Ez a ***lentről felfelé*** történő integráció. A gyakorlatban sokszor ezek keverékét alkalmazzák.

A rendszer integrálása és tesztelése során mindig inkrementális megközelítést kell alkalmazni. Ennek oka az, hogy az integráció folyamata során felmerülő hibák megtalálása nehéz feladat, mert rendszerkomponensek között komplex együttműködés zajlik. Ennek folyamatát mutatja be a következő ábra:



. ábra. Inkrementális integrácós tesztelés

Az integráció megtervezésekor mindig döntenünk kell a komponensek integrációjának sorrendjéről. Olyan esetekben, amikor a vásárló nincs bevonva a folyamatba, a leggyakrabban használt funkcionalitást megvalósító komponenseket célszerű elsőként integrálni, azaz azokat, amelyeket a legtöbbet teszteltük.

Regressziós tesztelés: Egy meglévő tesztsorozat újbóli lefuttatását jelenti. Akkor van rá szükség, ha egy új komponenst integrálunk és tesztelünk. Az új komponens intergrálása megváltoztathatja a korábbi, már tesztelt komponensek közötti együttműködések mintáját. Olyan hibákat is felfedezhetünk, amelyek az egyszerűbb konfiguráció tesztelésénél nem jelentkeztek. Ezért nem szabad csakis az új komponensre vonatkozó teszteket futtatni, hanem a korábbiakat is.

Igen költséges folyamat, és gyakorlat alkalmazásához automatizált támogatásra van szükség. Pl.: JUnit tesztelés.

11.1.2 Kiadástesztek

A kiadástesztelés az a folyamat, amelynek során a rendszervásárlóknak leszállítandó kiadás (verzió) tesztelése történik. A folyamat elsődleges célja, hogy megmutassa, a rendszer megfelel a követelményeinek funkcionalitás, teljesítmény, üzembiztonságra vonatkozóan.

Egy **fekete doboz folyamat**, ahol a teszteket a rendszer specifikációjából származtatják. A rendszert fekete dobozként kell tekinteni, viselkedésére csak bemeneteinek és kimeneteknek a vizsgálatával lehet következtetni. Más néven ezt *funkcionális tesztelésnek* is nevezik.

A kiadások tesztelése során meg kell próbálni „elrontani" a szoftvert oly módon, hogy kiválasszuk azokat a bemeneteket, amelyek nagy valószínűséggel rendszerhibát okoznak.

11.1.3 Teljesítménytesztelés (stressztesz)

Amikor teljesen integráltuk a rendszert, lehetséges annak eredendő tulajdonságainak tesztelése, mint pl. teljesítmény és a megbízhatóság tesztelése. Teljesítményteszteket azért készítünk, mert biztosítani szeretnénk, hogy a tervezett terhelés mellett a rendszer képes dolgozni. Ez általában olyan tesztek sorozata, ahol a terhelés addig nő, amíg a rendszer teljesítménye elfogadhatatlanná nem válik. A rendszereket általában megadott terhelésre tervezik. A stressztesz a tervezett maximális terhelésen túl is folytatja a tesztet, mindaddig, amíg a rendszer hibázik. Ennek két oka van:

1. **Teszteli a rendszer viselkedését szélsőséges körülmények között**. Ilyen körülmények között fontos, hogy a túlterhelés ne okozzon adatvesztést vagy a felhasználói szolgáltatások váratlan eltűnését.

2. **Terheli a rendszert, amellyel olyan hiányosságok is napvilágra kerülnek, amelyek normális körülmények között nem jelennek meg.** Ezek a hiányosságok normál használat során nem okoznának rendszerhibát, de felléphet a normál körülmények váratlan

kombinációja, amit a stressztesztelés előidéz.

A stressztesztelés különösen elosztott rendszereknél lényeges. Ezek a rendszerek gyakran nagy teljesítményromlást mutatnak, amikor nagyon leterheltek. A hálózat ilyenkor telítődik a koordinációs adatokkal, amiket a folyamatok küldenek egymásnak, és mivel a többi folyamattól szükséges adatokat eközben várják.

11.2 Komponenstesztelés

A komponenstesztelés (amit néha *egységtesztelésnek* is neveznek) a rendszer önálló komponenseinek tesztelése. Ez egy hiányosságtesztelési folyamat, mivel célja a vizsgált komponensekben lévő hibák felderítése. A legtöbb rendszerben a komponens teszteléséért annak fejlesztője felelős.

A tesztelhető komponensek különbözők lehetnek:

1. Önálló függvények vagy objektumok esetén módszerek.
2. Több attribútumot és módszert tartalmazó objektumosztályok.
3. Különböző objektumokból és/vagy függvényekből készült összetett komponensek, amelyek funkcionalitását interfészeken keresztül érhetjük el.

11.2.1 Interfésztesztelés

Az interfésztesztelésre akkor kerül sor, amikor egy nagyobb rendszer létrehozásához modulokat és alrendszereket integrálunk, amelyek egymással interfészeken keresztül kommunikálnak. Ez a fajta tesztelés különösen fontos az objektum-orientált és a komponens alapú programozásnál, amikor osztályokat, objektumokat és komponenseket újrafelhasználunk. Itt a hibák inkább az objektumok között interakciók eredményei nem az egyedi objektumok viselkedéséből adódnak.

A programkomponensek között különböző típusú interfészek léteznek, következésképpen különböző típusú interfészhibák fordulhatnak elő.

1. ***Paraméter-interfészek.*** Ezek olyan interfészek, ahol az adat- vagy néha a függvényreferenciák továbbítódnak az egyik komponenstől a másikhoz.
2. ***Osztott memóriájú interfészek.*** Olyan interfész, ahol egy memóriablokk van megosztva az alrendszerek között. Az adatokat az egyik alrendszer a memóriába írja, ahonnan egy másik alrendszer kiolvassa.
3. ***Procedurális interfészek.*** Ezek olyan interfészek, ahol az egyik alrendszer a más alrendszerek által hívható eljárások egy halmazát bezárja.
4. ***Üzenettovábbító interfészek.*** Ezek olyan interfészek, ahol egy alrendszer valamilyen szolgáltatást kér egy másik alrendszertől úgy, hogy üzenetet továbbít hozzá. A szolgáltatás lefuttatásával kapott eredményeket pedig egy válaszüzenet tartalmazza.

Komplex rendszereknél a gyakori interfészhibák:

- **Interfész téves alkalmazása:** főleg a paraméter interfészeknél fordul elő, rossz típusú, rossz sorrendű, nem megfelelő számú paraméter átadás.
- **Interfész félreértelmezése:** A hívó komponens félreértelmezi a hívott specifikációját. pl. a bináris keresést meghívjuk egy rendezetlen tömbbel, így a keresés hibás lesz.
- **Időzítési hibák:** valós idejű rendszereknél fordul elő, amelyek osztott memóriájú vagy üzenettovábbító interfészt használnak. Az adat előállítója és feldolgozója eltérő sebességgel üzemelhet, így a feldolgozó idejétmúlt adatot kaphat.

Az interfészhibák tesztelése nehéz, mert néhányuk csak szokatlan feltételek között jelentkezik. Pl.: egy objektum fix hosszúságú struktúraként implementál egy sort. A hívó objektum feltételezheti, hogy a sor végtelen adatstruktúraként lett megvalósítva, és nem ellenőrzi a túlcsordulást. Ez a hiba csak akkor derül ki a tesztelés során, ha a teszteset kierőlteti a túlcsordulást.

11.3 Teszteset tervezés

A tesztesettervezés a rendszer- és komponenstesztelés azon része, amikor a rendszer tesztelését végző tesztesetek tervezése történik. A folyamat célja a program hiányosságainak hatékony felderítésére alkalmas tesztesetek kialakítása.

A tesztesetek tervezésekor különböző lehetőségek közül választhatunk:

1. **Követelmény alapú tesztelés.** Amikor a teszteseteket a rendszer követelményeinek tesztelése céljából készítjük.
2. **Partíciós tesztelés.** Input és output partíciókat hozunk létre, és úgy tervezzük meg a teszteket, hogy a rendszer minden partícióból lefuttatja az inputokat, és minden partícióba generál outputot.
3. **A strukturális tesztelés.** A program részeit vizsgáló teszteket a program szerkezetének ismeretében készítjük el.

A tesztesetek tervezésének egyik alapelve, hogy a követelményekből kiindulva a legmagasabb szintű tesztekkel kezdjünk, majd a részletesebb tesztekkel folyamatosan, a partíciós és a strukturális tesztek segítségével bővítsünk.

11.3.1 Követelményalapú tesztelés

A követelményalapú tesztelés olyan szisztematikus teszteset tervezést jelent, amikor az egyes követelményekből tesztsorozatokat származtat. Ez a tesztelési mód inkább validáció, mintsem hiányosságtesztelés - azt próbáljuk bemutatni, hogy a rendszer megfelelően megvalósítja a követelményekben leírtakat.

11.3.2 Partíciós tesztelés

Egy program inputjai általában különböző osztályokba esnek. Ezek rendelkeznek valamilyen közös jellemzővel, pl. pozitív vagy negatív számok, szóköz nélküli sztringek, stb. A programok általában az osztály minden tagjára hasonló módon viselkednek. Ezért nevezik ezeket ekvivalencia – osztályoknak vagy tartományoknak. Ekvivalencia-osztály például az érvénytelen és az érvényes inputok halmaza is.

A hiányosságtesztelés szisztematikus megközelítése az ekvivalencia-osztályok azonosításán alapul. Először meg kell határozni az osztályokat, majd minden osztályból teszteseteket kell választani. Az osztály határáról és közepéről is érdemes teszteseteket választani. Az ekvivalencia-osztályok a programspecifikáció vagy a felhasználói dokumentáció alapján azonosíthatók.

11.3.3 Struktúra teszt

A struktúrateszt esetében a teszteket a szoftver struktúrájának és implementációjának ismeretében készítjük. Ezt a megközelítést néha hívják ***fehér doboz*** tesztelésnek. Többnyire kis programegységekre, objektumokra, alprogramokra alkalmazzák. Az algoritmusról szerzett tudás alapján pontosabban tudjuk azonosítani az ekvivalencia-osztályokat. Pl.: a keresőeljárás legyen a bináris keresés, ahol a sorozatot egy rendezett tömbként adjuk át. A kód vizsgálatával láthatjuk, hogy a keresési teret három részre lehet osztani: középső elem, nála kisebbek és nála nagyobbak.

11.4 Tesztautomatizálás

A tesztelés a szoftverfolyamat drága és fáradságos szakasza. Ennek eredményeképpen az elsők között kifejlesztett szoftvereszközök a tesztelő eszközök voltak. Ilyen keretrendszer a JUnit, amely Java-osztályok olyan halmaza, amit a felhasználó kibővíthet annak érdekében, hogy létrehozzon egy automatizált tesztkörnyezetet. Minden egyedi tesztet objektumként kell megvalósítani, és a teszt végrehajtója futtatja az összes tesztet. A teszteket úgy kell megírni, hogy jelezzék, hogy a tesztelt rendszer elvárt módon viselkedik-e.

12. Kialakuló technológiák

12.1 Szolgáltatásorientált architektúra

A szolgáltatásorientált architektúra (service-oriented architecture, SOA) lényegében az elosztott rendszerek fejlesztésének módja, ahol a rendszerek komponensei különálló szolgáltatások. Ezek a szolgáltatások földrajzilag egymástól távol elhelyezkedő számítógépeken futhatnak. A szolgáltatások kommunikációjának és az adatcserének a támogatására számos szabványos protokoll született. Következésképpen egy szolgáltatás

lényege, hogy a szolgáltatás biztosítása független a platformtól, az implementációs nyelvtől és a szolgáltatást igénybe vevő alkalmazástól.

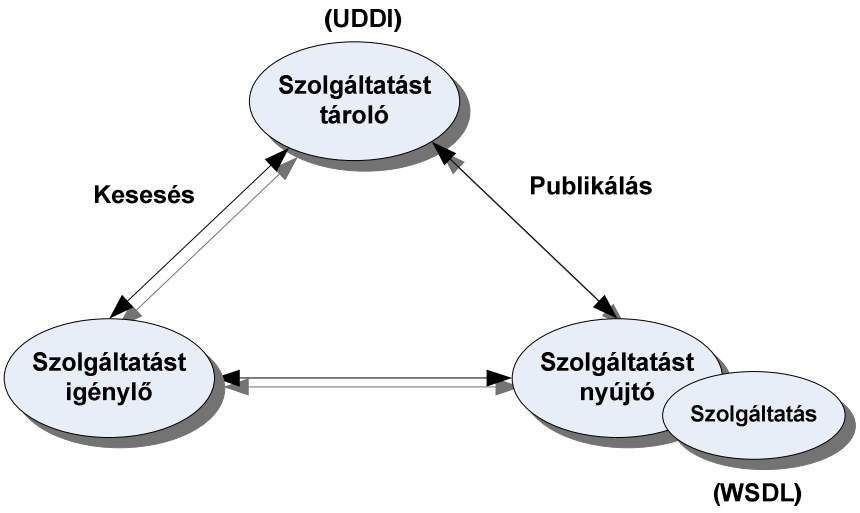
Szolgáltatás: lazán csatolt, újrafelhasználható szoftverkomponens, amely olyan diszkrét funkcionalitást zár be, amely elosztható és programozói eszközökkel elérhető.

A www fejlődése azt eredményezte, hogy a kliensszámítógépek saját szervezetükön kívül elhelyezkedő távoli szerverekhez is hozzáférnek. Ha az információikat HTML-formátumra hozták, akkor azok elérhetővé váltak. Azonban ez a hozzáférés kizárólag csak webböngészőn keresztül volt lehetséges, és az információtárolókhoz történő közvetlen hozzáférés nem volt megvalósítható.

Ennek a problémának a kiküszöbölése érdekében alakult ki a **webszolgáltatások** fogalma. A webszolgáltatás segítségével az információk elérhetővé válnak más programok számára is, egy web-szolgáltatás-interfész definiálásával és publikálásával. Ez az interfész definiálja az elérhető adatokat, és megadja, hogy hogyan lehet azokhoz hozzáférni.

Webszolgáltatás: olyan szolgáltatás, amely szabványos internetes és XML-alapú szabványokkal érhető el.

A következő ábra a szolgáltatásorientált architektúra általános felépítését mutatja be:



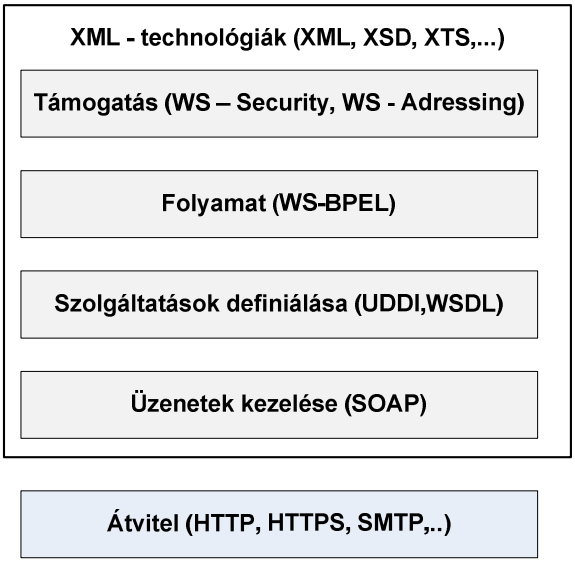
. ábra. Szolgáltatásorientált architektúra

A szolgáltatások gyártói megtervezik és implementálják a szolgáltatásokat, majd egy WSDL nevű nyelven specifikálják azokat. Ezenkívül a szolgáltatásokról információkat publikálnak egy általánosan elérhető tárolóban az UDDI publikációs szabvány segítségével. A szolgáltatások igénybe vevői (néha szolgáltatáskliensnek is nevezik őket), akik szeretnének használni egy szolgáltatást, keresést hajtanak végre az UDDI-tárolóban, hogy megtalálják azt, valamint annak nyújtóját. Ezután alkalmazásukat összekapcsolhatják az adott szolgáltatással, és kommunikálhatnak azzal. A kommunikációhoz általában a SOAP nevű protokollt használják.

A szolgáltatásorientált architektúra hatalmas fejlődést jelent, különösen az üzleti alkalmazásrendszerek szempontjából. Mert:

- Rugalmasságot biztosít: a szolgáltatások elkészíthetők helyileg, illetve igénybe vehetők külső szolgáltatóktól
- A szolgáltatások bármilyen nyelven implementálhatók, így lehetővé teszi, hogy a vállalat különböző részein esetlegesen használt különböző platformok és implementációs technológiák együttműködhessenek.
- Ami a legfontosabb, a szolgáltatások segítségével a cégek és más szervezetek képesek az együttműködésre, valamint egymás üzleti funkcióinak használatára.

A szolgáltatásorientált architektúrák nem szenvednek inkompatibilitási problémáktól, mert a fejlesztéseket a kezdetektől aktív szabványosítási folyamat követte. Alapvető szabványok a webszolgáltatás támogatására:



. ábra. Webszolgáltatás szabványok

A webszolgáltatás-orientált architektúrák szabványai röviden a következők:

- ***SOAP:*** Ez egy üzenetcsere-szabvány, amely a szolgáltatások közötti kommunikációt támogatja. Definiálja a szolgáltatások üzeneteinek szükséges és opcionális komponenseit.

- ***WSDL:*** Webszolgáltatás definíciós nyelv (Web Service Definition Language, WSDL) definiálja, hogyan kell a szolgáltatások gyártóinak elkészíteniük szolgáltatásaik interfészeit.

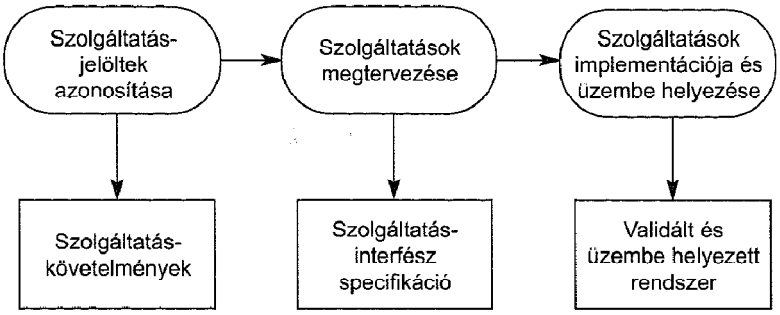
- ***UDDI:*** Az Univerzális leírás, felfedezés és integráció (Universal Description, Discovery and Integration, UDDI) szabványa. Definiálja, hogy milyen komponenseket kell tartalmaznia a szolgáltatások specifikációinak, hogy könnyen felfedezhetők legyenek. Ez információkat tartalmaz a szolgáltatás gyártójáról, a szolgáltatásról, a szolgáltatás leírásának helyéről. Az UDDI-tárolók segítségével fedezhetik fel a felhasználók az elérhető szolgáltatásokat.

- ***WS-BPEL:*** Szabványos munkafolyamat nyelv, amely különböző szolgáltatásokat tartalmazó folyamatprogramok definiálására használatos.

12.1.1 Szolgáltatások tervezése

A szolgáltatástervezés az a folyamat, melynek során szolgáltatásorientált alkalmazásokban újrafelhasználható szolgáltatásokat fejlesztünk ki. A szolgáltatások tervezőinek biztosítaniuk kell, hogy a szolgáltatás egy olyan újrafelhasználható absztrakciót reprezentál, amely különböző rendszerek számára is hasznos lehet. A szolgáltatástervezés folyamatának három logikai állomása van:

1. **Szolgáltatásjelöltek azonosítása:** Itt megkeressük azokat a lehetséges szolgáltatásokat, amelyeket a későbbiekben implementálhatunk, és definiáljuk a szolgáltatások követelményeit.
2. **Szolgáltatások megtervezése:** Ebben a szakaszban tervezzük meg a logikai és a WSDL szolgáltatás-interfészeket.
3. **Szolgáltatások implementációja és üzembe helyezése:** Itt implementáljuk és teszteljük a szolgáltatásokat, majd a felhasználók számára hozzáférhetővé tesszük őket.



. ábra. A szolgáltatástervezés folyamata

12.1.1.1 Szolgáltatásjelöltek azonosítása

A szolgáltatásoknak az üzleti logikát kell támogatniuk. A szolgáltatásjelöltek azonosítási folyamatának feladata, hogy megértse és elemezze a szervezet üzleti folyamatait, majd eldöntse, milyen szolgáltatások szükségesek a folyamatok támogatásához. Alapvetően a szolgáltatásoknak három típusa különböztethető meg:

1. ***Segédszolgáltatások.*** Ezek olyan általános funkcionalitást implementálnak, amit különböző üzleti folyamatok használhatnak. Pl.: egy pénznemátváltó szolgáltatás, amelynek feladata egy pénznem (például dollár) átváltása egy másikra (például euró).

2. ***Üzleti szolgáltatások.*** Ezek a szolgáltatások egy bizonyos üzleti folyamathoz

kapcsolódnak. Pl.: üzleti szolgáltatás lehetne például a hallgatók regisztrálása egy kurzusra.

3. ***Koordinációs vagy folyamatszolgáltatások.*** Ezek a szolgáltatások sokkal általánosabb üzleti folyamatot támogatnak, amelyben több különböző faktor és tevékenység is megjelenhet. Koordinációs szolgáltatásra jó példa egy vállalat rendelési szolgáltatása, amelyben az ügyfelek leadhatják rendeléseiket és fizethetnek.

A szolgáltatásjelöltek azonosítása során célunk olyan szolgáltatások kiválasztása, amelyek logikai egységet alkotnak, függetlenek és újrafelhasználhatók. A folyamat végeredménye az azonosított szolgáltatások halmaza, valamint a hozzájuk kapcsolódó követelmények.

12.1.1.2 Szolgáltatás-interfész tervezése

A szolgáltatásjelöltek kiválasztása után a szolgáltatástervezés folyamatának következő lépése a **szolgáltatás-interfészek megtervezése**. Ez magába foglalja a szolgáltatáshoz kapcsolódó műveletek és paramétereik definiálását. A szolgáltatás-interfész tervezésének **három** lépése a következő:

1. **Az interfész logikai tervezése** során azonosítjuk a szolgáltatásokhoz tartozó műveleteket, meghatározzuk a műveletek bemeneteit és kimeneteit, továbbá a műveletekhez tartozó kivételeket. A szolgáltatás követelményeiből indul ki.

2. **Az üzenetek megtervezése** az a lépés, amelyben eldől, milyen lesz az üzenetek struktúrája, és hogy a szolgáltatás milyen üzeneteket küld és fogad.

3. **A WSDL-fejlesztés:** a logikai és üzenetterveinket átültetjük egy WSDL nyelven íródott absztrakt interfészleírásba. A legtöbb szolgáltatásorientált fejlesztést támogató környezet (például az ECLIPSE környezet) tartalmaz olyan eszközöket, amelyek a logikai interfészleírást lefordítják a megfelelő WSDL-reprezentációba.

12.1.1.3 Szolgáltatások implementációja és üzembe helyezése

Az interfészek megtervezése után a következő lépés a **szolgáltatások implementálása** és üzembe helyezése. Ez általában szabványos programozási nyelvek segítségével történik, pl. C#, Java, melyek tartalmaznak könyvtárakat a szolgáltatásfejlesztés széles körű támogatására.

Az implementáció után, az üzembe helyezést megelőzően, a **szolgáltatást tesztelni kell**. Ehhez hozzátartozik a bemenetek elemzése és osztályozása, a kombinációkat tükröző üzenetek elkészítése, és annak ellenőrzése, hogy a megfelelő kimenő üzenetek érkeztek-e. Érdemes kivételeket generálni minden tesztben, hogy lássuk, a szolgáltatás képes-e megbirkózni az érvénytelen bemenetekkel. Ma már több olyan eszköz is beszerezhető, amely alkalmas a szolgáltatások vizsgálatára és tesztelésére, valamint képes teszteseteket generálni WSDL-specifikációból.

A folyamat befejező lépése **a szolgáltatások üzembe helyezése**. Ez azt jelenti, hogy a szolgáltatást használatra elérhetővé tesszük egy webszerveren. A legtöbb szerverszoftver esetén ez egyszerűen megy, csak egy megadott könyvtárba kell telepíteni a végrehajtható szolgáltatást tartalmazó állományt. Ha a szolgáltatást publikusnak szánjuk, akkor még egy UDDI-leírást is készítenünk kell.

Forrásmunkák:

1. **Ian Sommerville:** Szoftverrendszerek fejlesztése, 2007 bővített, második kiadás.
2. **Vég Csaba:** Rational Unified Process – áttekintés. http://www.logos2000.hu/docs/RUP.pdf
3. **Szabolcsi Judit:** Szoftvertechnológia, 2008.
4. **Juhász Sándor Ferenc:** Az Extreme Programming programozás technikai elvei.