

23 SZOFTVERTESZTELÉS

TÉMAKÖRÖK

A fejezet célja a szoftvertesztelő folyamatok ismertetése és tesztelési technikák bemutatása. A fejezet elolvasása után:

- érteni fogjuk a validációs tesztelés és a hiányosságtesztelés közötti különbségeket;
- ismerni fogjuk a rendszertesztelés és a komponenstesztelés alapelveit;
- ismerni fogjuk a teszteset-generáláshoz alkalmazható háromféle stratégiát;
- érteni fogjuk a tesztautomatizálást támogató szoftvereszközök legfőbb jellemzőit.

TARTALOM

- 23.1. Rendszertesztelés
- 23.2. Komponenstesztelés
- 23.3. Tesztesettervezés
- 23.4. Tesztautomatizálás

A 4. fejezetben kifejtettem egy általános tesztelési eljárást, amely a különálló programegységek, mint például függvények vagy objektumok tesztelésével kezdődik. Ezek később alrendszerekbe és rendszerekbe integrálódnak, majd az egységek közötti interakciók kerülnek tesztelésre. Végül, miután a rendszer elkészült, a vásárló végrehajthat egy elfogadási tesztsorozatot annak ellenőrzésére, hogy a rendszer teljesíti-e az előírásokat.

A tesztelési folyamat ezen modellje megfelelő nagy rendszerek fejlesztése során, de kisebb, illetve szkripteléssel vagy újrafelhasználással fejlesztett rendszerek esetén ez a folyamat gyakran kevesebb lépésből áll. A tesztelési folyamat absztraktabb megközelítése látható a 23.1. ábrán. A két alapvető tesztelési tevékenység a komponenstesztelés – a rendszer részeinek tesztelése – és a rendszertesztelés – a rendszer egészként történő tesztelése.

A komponenstesztelési fázis célja a különálló programkomponensek tesztelésével végzett hibafelderítés. Ezek a komponensek lehetnek függvények, objek-

**23.1. ábra.** Tesztelési fázisok

tumok vagy a 19. fejezetben leírt újrafelhasználható komponensek. A rendszer-tesztelés során ezeket a komponenseket alrendszerekbe vagy a teljes rendszerbe integráljuk. Ebben a fázisban a rendszertesztelés arra összpontosít, hogy a rendszer megfelel-e a vele szemben támasztott funkcionális és nemfunkcionális követelményeknek, és hogy nem viselkedik-e nem várt módon. A rendszertesztelés szükségszerűen előhoz olyan, a komponensekben rejlő hibákat, amelyeket a korábbi tesztek során nem találtunk meg.

Ahogy arról már a 22. fejezetben említést tettem, a szoftvertesztelésnek két célja van:

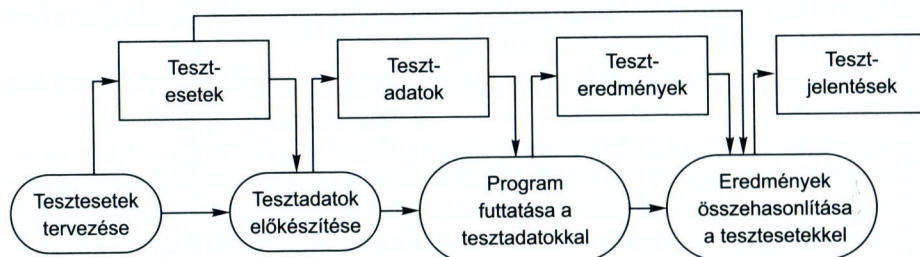
1. *Bizonyítani a fejlesztő és a vásárló számára, hogy a szoftver megfelel a vele szemben támasztott követelményeknek.* Egyedi szoftver esetén ez azt jelenti, hogy a felhasználói és rendszerkövetelményeket leíró dokumentum minden követelményére vonatkozóan legalább egy tesztnak lennie kell. Általános szoftvertermékek esetén azt jelenti, hogy minden, a termék kiadásába tartozó rendszer-tulajdonságra lennie kell tesztnak. A 4. fejezetben taglaltak alapján, bizonyos rendszerek tartalmazhatnak explicit átvételi vizsgálati lépést is, ahol a vásárló formálisan ellenőrzi, hogy a leszállított rendszer megfelel-e a specifikációjában leírtaknak.
2. *Felfedezni a szoftverben azokat a hibákat és hiányosságokat, amelyekben a szoftver viselkedése helytelen, nemkívánatos, vagy nem felel meg a specifikációnak.* A hiányosságtesztelés mindenfajta nemkívánatos rendszerviselkedésnek (mint például rendszerösszeomlás, más rendszerekkel folytatott nemkívánatos interakciók, rossz számítások, adatkorruptió) a kiirtásával foglalkozik.

Az első cél a validációs teszteléshez vezet, ahol azt várjuk a rendszertől, hogy egy annak elvárt használatát tükröző tesztesethalmazra helyesen működjön. A második cél a hiányosságteszteléshez vezet, ahol a teszteseteket úgy tervezték meg, hogy hiányosságokat fedjenek fel. A tesztesetek szándékosan pontatlanok lehetnek, és nem kell tükrözniük a rendszer normál működését. Validációs tesztelés esetén egy teszt akkor sikeres, ha azt alkalmazva a rendszer helyesen működik. Hiányosságtesztelés esetén viszont a tesztet akkor nevezzük sikeresnek, ha az felderít egy olyan hiányosságot, amely a rendszer helytelen működését okozza.

A tesztelés nem képes azt megmutatni, hogy a szoftvernek nincsenek hiányosságai, vagy minden körülmények között a megadott módon fog viselkedni. Lehetséges, hogy egy ellenőrzött teszt a rendszer további problémáit is felderíti. Ahogy Edsger Dijkstra, a szoftvertervezés történetének korai vezéralakja ékesen szólóan kijelentette (Dijkstra és mások, 1972), „a tesztelés a hibáknak csak a jelenlétét, nem pedig a hiányát tudja megmutatni”.

Összességében a szoftvertesztelés célja meggyőzni a rendszer fejlesztőit és vásárlóit arról, hogy a szoftver elég jó ahhoz, hogy üzemszerűen használni lehessen. A tesztelés az a folyamat, amely a megbízhatóságot építi be a szoftverbe.

A tesztelés folyamatának általános modellje látható a 23.2. ábrán. A teszteteset nem más, mint a teszthez szükséges inputok és a rendszertől várt outputok specifikációja, valamint egy ismertető, hogy mit tesztlünk. A tesztadatok kifejezetten a rendszer tesztelésére létrehozott inputok. A tesztadatok néha automatikusan generálhatók, az automatikus tesztetes-generálás viszont lehetetlen. A tesztek outputjait csak azok tudják előre megjósolni, akik értik, hogy a rendszernek mit kellene csinálnia.



23.2. ábra. A szoftvertesztelési folyamat modellje

A kimerítő tesztelés, ahol az összes lehetséges program-végrehajtási szekvenciát teszteljük, nem praktikus. Ezért a tesztelésnek a lehetséges tesztetek egy részhalmazán kell alapulnia. A szervezeteknek ki kell fejleszteniük az irányelveket e részhalmazok kiválasztásához, és nem szabad azt a fejlesztőcsoportra hagyni. Ezek alapulhatnak általános tesztelési irányelveken, például hogy minden programutasítást legalább egyszer le kell futtatni. A tesztelési irányelvek alapulhatnak rendszer-üzemeltetési tapasztalatokon, és a működő rendszer sajátosságainak tesztelésére összpontosíthatnak:

1. A menükön keresztül elérhető összes rendszerfunkciót tesztelni kell.
2. Ugyanazon menün keresztül elérhető funkciókombinációkat (mint például szövegformázás) tesztelni kell.
3. Ahol felhasználói inputot várunk, az összes funkciót tesztelnünk kell, helyes és helytelen inputtal egyaránt.

Tapasztalatból tudjuk, hogy a fontosabb szoftvertermékeknel, mint például a szövegszerkesztők vagy táblázatkezelők, hasonló irányelveket használnak a termék tesztelésékor. Ha a szoftver tulajdonságait elkülönítve tekintjük, rendben működnek. A probléma Whittaker szerint (Whittaker, 2002) ott kezdődik, amikor a jellemzők kombinációját nem teszteljük együtt. Bemutat egy példát is, amelyben egy gyakran használt szövegszerkesztő esetén egy többhasábos elrendezés és a lábjegyzetek együttes használata a szöveg helytelen elrendezését eredményezi.

A V & V folyamat részeként a menedzserek döntenek, hogy a tesztelés egyes szakaszaiért ki a felelős. A legtöbb rendszernél a programozók felelősséget vállalnak az általuk fejlesztett komponensek teszteléséért. Miután ez kész, a munka átkerül egy integrációs csapathoz, akik a különböző fejlesztőktől érkező modulokat integrálják, elkészítik a rendszert és tesztelik a rendszer egészét. Azonban kritikus rendszereknél sokkal formálisabb folyamatot használnak, ahol független tesztelők felelnek a tesztelési folyamat összes lépéséért. A kritikus rendszerek tesztelésekor a tesztek elkülönítve fejlesztik ki, és a teszteredményekről részletes feljegyzéseket készítenek.

A fejlesztők által végzett komponenstesztelés azon az intuitív értelmezésen alapszik, hogy a komponenseknek hogyan kellene működniük. A rendszertesztelésnek azonban írott rendszer-specifikációkon kell alapulnia. Ez lehet részletes rendszerkövetelmény-specifikáció, mint ahogy a 6. fejezetben kifejtettem, vagy lehet az implementálandó rendszer jellegzetességeinek felhasználóorientált specifikációja. Az integrációs tesztekért egy különálló csoport felel. Ahogy arról a 4. fejezetben már szó volt, a rendszertesztlő csapat a felhasználói és rendszerkövetelmény-dokumentációkat használja a rendszertesztlési tervek kifejlesztéséhez (4.10. ábra).

A tesztelésről szóló legtöbb cikk a komponensteszteléssel kezdődik, és a rendszertesztteléssel folytatódik. Ebben a fejezetben megfordítottam ezt a sorrendet, mert egyre több esetben a szoftverfejlesztés újrafelhasználható komponensek integrációját és a létező szoftverek konfigurálását és a megadott követelmények szerinti átdolgozását jelenti. Ilyen esetekben gyakorlatilag csak rendszertesztlésről beszélhetünk, különálló komponenstesztelésről nem.

23.1. RENDSZERTESZTELÉS

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

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

1. *Integrációs tesztelés*, ahol a tesztelést végző csapat hozzáfér a rendszer forráskódjához. Ha egy problémára fény derül, az integrációs csapat megpróbálja megtalálni a probléma forrását, és azonosítani a nyomkövetendő komponenseket. Az integrációs tesztelés leginkább a rendszer hiányosságainak megtalálásával foglalkozik.
2. *Kiadástesztelés*, ahol a rendszer egy felhasználók számára kiadható verziója kerül tesztelésre. Itt a tesztcsoport azt vizsgálja, hogy a rendszer megfelel-e a követelményeknek, valamint a rendszer megbízhatóságának biztosításával foglalkozik. A kiadástesztelés általában fekete doboz tesztelés, ahol a teszt-

csapatnak egész egyszerűen csak azt kell bemutatnia, hogy a rendszer jól működik-e, avagy sem. A problémákat továbbítják a fejlesztőcsapatnak, akik a program nyomkövetését végzik. Ha ebbe a kiadástesztelésbe a vásárlót is bevonják, akkor ezt *elfogadási tesztelés*nek nevezzük. Ha a verzió elég jó, a vásárló elfogadhatja használatra.

Az integrációs tesztelésre alapvetően úgy kell tekintenünk, mint rendszerkomponensek egy csoportjából vagy fürtjéből álló befejezetlen rendszerek tesztelésére. A kiadástesztelés a rendszer azon kiadásának tesztelésével foglalkozik, amelyet le szeretnénk szállítani a vásárlóknak. Ezek természetesen átfedhetik egymást, különösen abban az esetben, ha inkrementális fejlesztést alkalmazunk és a kiadásra kerülő rendszer még nincs teljesen készen. Általában azt mondhatjuk, hogy az integrációs tesztelés számára a rendszerben található hiányosságok felderítése, míg a rendszertesztelés számára a rendszer követelményeknek való megfelelése ellenőrzése az elsődleges, azonban a gyakorlatban mindkét folyamat során végzünk validációs és hiányosságtesztelést is.

23.1.1. Integrációs tesztelés

A rendszer-integráció folyamata magában foglalja a rendszer komponenseiből történő felépítését (29. fejezet) és az eredményül kapott rendszer tesztelését a komponensek együttműködéséből adódó problémák felderítésére. Az integrált komponensek lehetnek kulcsrakész, egy egyedi rendszer számára átdolgozott újrafelhasználható, illetve újonnan kifejlesztett komponensek. Nagy rendszerek esetében valószínűleg mindhárom fajta komponenssel találkozunk. Az integrációs tesztelés azt ellenőrzi, hogy ezek a komponensek valóban képesek-e együttműködni, megfelelően vannak-e meghíva és interfészeiken keresztül a megfelelő adatokat a megfelelő időpontban küldik-e át.

A rendszer-integráció során azonosítani kell a rendszer különböző funkcionálisait biztosító komponensek csoportjait, majd további kód hozzáadásával integrálni kell őket, hogy együtt tudjanak működni. Sok esetben elsőként a rendszer váza készül el, és ehhez adódnak hozzá a komponensek. Ezt *fentről lefelé* történő integrációnak nevezzük. Más esetekben először a gyakran használt, közös szolgáltatásokat (mint a hálózati és az adatbázis-elérés) biztosító infrastrukturális komponensek integrálását végezzük először, majd hozzáadjuk a funkcionális komponenseket. Ez a *lentől felfelé* történő integráció. A gyakorlatban sokszor ezek keverékét alkalmazzák, és mind az infrastrukturális, mind pedig a funkcionális komponenseket inkrementumokban adják a rendszerhez. A fentről lefelé és a lentől felfelé történő integráció során általában egyaránt további kódot kell kifejleszteni ahhoz, hogy az egyéb komponenseket szimulálni tudjuk, és lehetővé tegyük a rendszer futtatását.

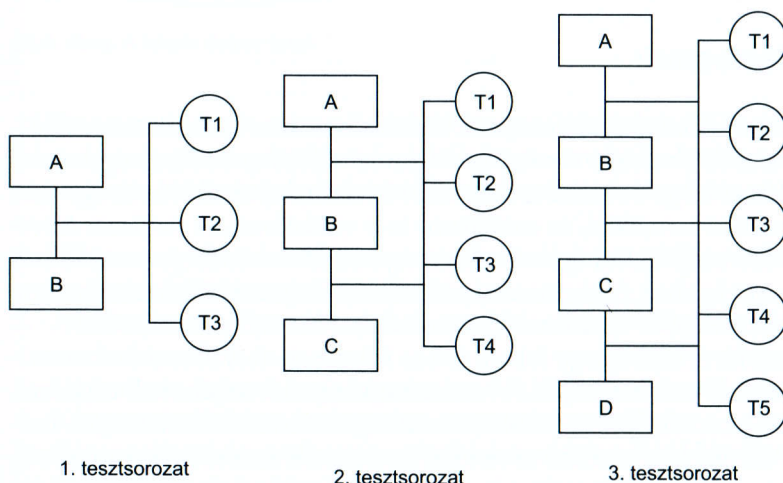
Az integrációs tesztelés során adódó fő nehézség a folyamat során feltárt hibák lokalizálása. A rendszerkomponensek között komplex együttműködés zajlik, és amikor egy rendellenes outputot találunk, nagyon nehéz lehet megtalálni a

hiba forrását. A hibák lokalizálásának megkönnyítéséhez a rendszer integrálása és tesztelése során mindig inkrementális megközelítést kell alkalmazni. Kezdetben csak minimális rendszer-konfigurációt kell integrálni, és ezt kell tesztelni. Ezt követően lehet a minimális konfigurációhoz komponenseket adni, majd tesztelni kell minden egyes inkrementum hozzáadása után.

A 23.3. ábrán látható példában A, B, C és D komponensek, T1–T5 pedig a kapcsolódó testsorozatok, amelyekkel a rendszer jellemzőit tesztelik. Először a T1, T2 és T3 testsorozatot futtatjuk az A és B komponensekből összeállított (minimális) rendszeren. Ha ezek hibákat tárnak fel, akkor azokat kijavítjuk. Integráljuk a C komponenst, és ismételjük meg a T1, T2 és T3 teszteket azért, hogy kizárjuk az A és B komponensek közötti nem várt együttműködés lehetőségét. Ha e tesztek során problémák merülnek fel, akkor ezek valószínűleg az új modullal való együttműködés miatt vannak. A probléma forrását lokalizáltuk, így egyszerűsödik a hiba behatárolása és javítása. A T4 testsorozatot is lefuttatjuk a rendszeren. Végül a D komponenst is integráljuk, és teszteljük a meglévő és új (T5) tesztekkel.

Az integráció megtervezésekor döntenünk kell a komponensek integrációjának sorrendjéről. Egy olyan folyamatban, mint amilyen például az XP, a vásárló be van vonva a fejlesztési folyamatba, és ő dönt arról, hogy az egyes rendszer-inkrementumokba milyen funkcionalitás kerüljön, így a rendszer-integrációt a vásárló elsőbbségi szempontjai vezérlik. Más esetekben, amikor kulcsrakész és speciálisan kifejlesztett komponensek integrációját végezzük, az ügyfelet nem lehet bevonni, így az elsőbbségi szempontokat az integrációt végző csapat határozza meg.

Ilyen esetekben a leggyakrabban használt funkcionalitást megvalósító komponenseket célszerű elsőként integrálni, azaz azokat, amelyeket a legtöbbet teszteltük. Például a LIBSYS könyvtári rendszerben elsőként a keresőfunkciót érdemes integrálni, mert így a felhasználók már egy minimális rendszerben is rá tudnak azokra a dokumentumokra keresni, amelyekre szükségük van. Ezután a doku-



23.3. ábra. Inkrementális integrációs tesztelés

mentumok letöltését végző részt érdemes hozzáadni a rendszerhez, majd fokozatosan a további funkciókat megvalósító komponensek integrációja szükséges.

Természetesen a valóságban ez nem olyan egyszerű, mint ahogy azt ez a modell sugallja. A rendszer eszközeinek implementációja számos komponensre kiterjedhet. Ezért egy új eszköz tesztelése számos különböző komponens integrációját igényelheti. A tesztelés pedig hibákat tárhat fel a komponensek és az egyéb rendszerrészek közötti együttműködésben. A hibák javítása esetleg nehézkes, mert hatással lehet a rendszer eszközeit implementáló egész komponenscsoportra. Továbbá ha egy új komponenszt integrálunk és tesztelünk, akkor ez megváltoztathatja a korábbi, már tesztelt komponensek közötti együttműködések minőségét. Olyan hibákat is felfedezhetünk, amelyek az egyszerűbb konfiguráció tesztelésénél nem jelentkeztek.

Éppen ezért fontos, hogy az új inkrementumok integrálásakor nemcsak az új rendszerfunkciók ellenőrzését végző új tesztekkel kell végrehajtani, hanem a korábbi inkrementumok tesztjeit is újra le kell futtatni. Egy meglévő tesztsorozat újbóli lefuttatását *regressziós tesztelésnek* nevezzük. Ha a regressziós tesztelés fényt derít bizonyos hibákra, akkor ellenőrizni kell, hogy ezek a problémák az előző inkrementumban vannak-e jelen, csak az új inkrementum hozta őket elő, vagy esetleg a hozzáadott inkrementum miatt vannak jelen.

A regressziós tesztelés igen költséges folyamat, és gyakorlat alkalmazásához automatizált támogatásra van szükség. A 17. fejezetben tárgyalt extrém programozás esetén az összes teszt végrehajtható kódként van leírva, amely specifikálja a teszt bemenetét és elvárt kimeneteit, és automatikus módon kerül ellenőrzésre. Ha ezt egy olyan automatizált tesztkörnyezetben használjuk, mint amilyen a JUnit (Massol és Husted, 2003), akkor ezek a tesztek automatikusan újra lefuttathatók. Az extrém programozás egyik alapelve, hogy minden új kódrészlet integrálásakor a teljes tesztsorozatot végig kell futtatni, és az új kódot mindaddig nem szabad elfogadni, amíg minden teszt sikeresen le nem futott.

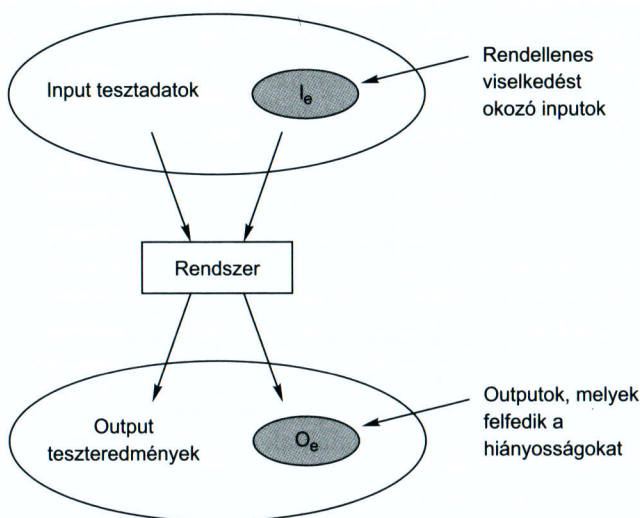
23.1.2. Kiadásteszték

A kiadástesztelés az a folyamat, amelynek során a rendszervásárlóknak leszállítandó kiadás (verzió) tesztelése történik. A folyamat elsődleges célja, hogy növelje a bizalmat abban, hogy a rendszer megfelel követelményeinek. Ha ez így van, akkor válik kiadható termékkel, és szállítható le a vásárlónak. A rendszer követelményeinek való megfelelést igazolandó, meg kell mutatni, hogy rendelkezik a specifikációban rögzített funkcionalitásra, teljesítményre és üzembiztonságra vonatkozó jellemzőkkel, és üzemszerű használat esetén rendben működik.

A kiadástesztelés általában egy fekete doboz folyamat, ahol a tesztek a rendszer specifikációjából származtatják. A rendszert fekete dobozként kell tekinteni, viselkedésére csak bemeneteinek és az azokhoz kapcsolódó kimeneteknek a vizsgálatával lehet következtetni. Más néven ezt *funkcionális tesztelésnek* is nevezik, mert a tesztelő a szoftvernek csupán a funkcionalitásával foglalkozik, nem pedig az implementációjával.

Egy fekete doboz típusú tesztelés alatt álló rendszer modelljét a 23.4. ábra illusztrálja. Ez a megközelítés egyaránt alkalmazható a függvény- vagy objektumalapú rendszerekre. A tesztelő inputokat ad át a komponensnek vagy a rendszernek, és a megfelelő kimeneteket megvizsgálja. Ha a kimenet nem az, amit előre vártunk (vagyis benne van az O_e halmazban), akkor a teszt felfedezett egy, a szoftverrel kapcsolatos problémát.

A kiadások tesztelése során meg kell próbálni „elrontani” a szoftvert oly módon, hogy a 23.4. ábrán látható I_e halmazból választunk teszteseteket. Ezzel az a célunk, hogy kiválasszuk azokat a bemeneteket, amelyek nagy valószínűséggel rendszerhibát okoznak (kimenetei az O_e halmazban vannak). A döntést a valószínűleg sikeres hiányosságtesztekkel kapcsolatos korábbi tapasztalataink és különböző tesztelési irányelvek segíthetik.



23.4. ábra. A fekete doboz teszt

Bizonyos szerzők, mint például Whittaker (Whittaker, 2002), tesztelési tapasztalataikat olyan irányelvekben foglalták össze, amelyek növelik a sikeres hiányosságtesztelés valószínűségét. Lássunk néhány ilyen irányelvet:

1. Válasszunk olyan bemeneteket, amelyek rákényszerítik a rendszert arra, hogy az összes hibaüzenetet legenerálja.
2. Alkalmazzunk olyan inputokat, amelyek a bemeneti pufferek túlcsoordulását eredményezik.
3. Ugyanazon inputot vagy inputsorozatot több alkalommal is ismételjük meg.
4. Kényszerítsük ki az érvénytelen outputok generálását.
5. Kényszerítsük ki azt, hogy a számítási eredmények túl kicsik vagy túl nagyok legyenek.

A rendszer követelményeivel szembeni megfelelésének ellenőrzésére a legjobb módszer a forgatókönyv-alapú tesztelés, amely során forgatókönyveket kell kigondolni, és ezekből kell teszteseteket kifejleszteni. A korábbi fejezetekben tárgyalt LIBSYS könyvtári rendszer használatának mikéntjét a következő példa írja le:

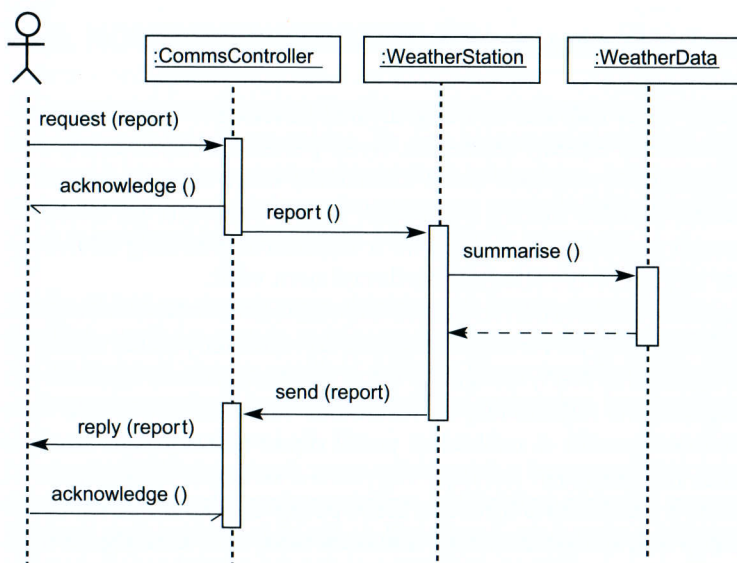
Egy Skóciában amerikai történelmet hallgató diák azt a feladatot kapja, hogy írjon dolgozatot „A határvidék mint nemzetalakító tényező az amerikai Nyugaton, 1840 és 1880 között” címmel. Ehhez sok könyvtárból lesz szüksége különböző forrásokra. Belép a LIBSYS-rendszerbe, és a keresési funkció segítségével megtudja, hogy hozzáférhet-e korabeli dokumentumokhoz. Különböző amerikai egyetemi könyvtárakban megtalálja őket, közülük néhányat letölt. Egy dokumentumhoz azonban csak akkor juthat hozzá, ha egyeteme megerősíti, hogy ő valóban diák, és nem üzleti célokra szeretné a dokumentumot használni. A LIBSYS ilyen engedélyek igénylését kezelő modulja rögzíti kérését, majd ha megkapja az engedélyt, a dokumentum letöltésre kerül a regisztrált könyvtár szerverére, majd kinyomtatják számára. A hallgató üzenetet kap a LIBSYS-től, amelyben arról tájékoztatják, hogy majd e-mailben értesítik, hogy mikor mehet a kinyomtatott dokumentumért.

Ebből a forgatókönyvből kiindulva számos tesztet készíthetünk, amelyeket majd a LIBSYS-rendszer újabb kiadásának megjelenítése előtt kell lefuttatni:

1. Helyes és helytelen bejelentkezési információkkal tesztelni kell a bejelentkezési mechanizmust, hogy meggyőződjünk arról, hogy a rendszer beengedi a hitelesített felhasználókat, míg a többieket nem.
2. Tesztelni kell a keresési funkciót is, hogy ellenőrizni tudjuk, hogy az ismert adatforrásokra kiadott lekérdezések valójában találnak eredményként dokumentumokat.
3. A megjelenítő rész tesztelésére is szükség van, ezzel azt ellenőrizzük, hogy a dokumentumok információi megfelelően jelennek-e meg.
4. Tesztelni kell a letöltési jogosultság kérelmezési mechanizmusát.
5. A letöltött és kinyomtatott dokumentum hozzáférhetőségéről szóló e-mail üzenetküldést is tesztelni kell.

Minden ilyen teszthez el kell készíteni egy tesztsorozatot, amely egyaránt tartalmaz érvényes és érvénytelen bemeneteket, és generál érvényes és érvénytelen kimeneteket. A forgatókönyv-alapú tesztelést úgy érdemes végezni, hogy előbb a legvalószínűbb forgatókönyveket teszteljük, a kivételes esetekre vonatkozó forgatókönyveket csak később vesszük elő. Így azokra a részekre összpontosítunk, amelyeket a leggyakrabban használnak.

Ha a rendszer követelményeinek leírására használati eseteket alkalmaztunk, ezek a használati esetek és a hozzájuk kapcsolódó szekvenciadiagramok a rendszer tesztelésének alapjául szolgálhatnak. Ezeket mind az integrációs, mind pedig a kiadásteszt során felhasználhatjuk, amit a 14. fejezetben megismert meteorológiai állomás rendszerén keresztül szemléltetünk.



23.5. ábra. Időjárási adatok begyűjtésének szekvenciadiagramja

A 23.5. ábra a meteorológiai állomás által végrehajtandó műveletek sorrendjét mutatja, amikor az válaszol a térképrendszer adatgyűjtési kérésére. Ezt a diagramot használhatjuk a tesztelendő műveletek azonosítására és a tesztek lefuttatásához szükséges tesztesetek megtervezéséhez. Ezért egy jelentéskérés kiadása a következő módszersorozat futtatását eredményezi:

CommsController:request → WeatherStation:report → WeatherData:summarise

A szekvenciadiagram arra is felhasználható, hogy meghatározzuk a teszthez szükséges inputokat és outputokat:

1. Egy beérkező jelentéskérésnek rendelkeznie kell a hozzá tartozó nyugtázással, és a kérésre válaszolva vissza kell adni egy jelentést. A tesztelés során pedig készíteni kell összesített adatokat, hogy ellenőrizni lehessen, a jelentés megfelelő szerkezetű-e.
2. A WeatherStation-höz érkező jelentéskérés egy generált összesített jelentést eredményez. Ezt elkülönítve tesztelhetjük olyan nyers adatok előállításával, amelyek megfelelnek a CommsController tesztelésére elkészített összesítéssel, továbbá ellenőrizhetjük, hogy a WeatherStation objektum helyesen készíti-e el ezt az összesítést.
3. Ezen nyers adatok a WeatherData objektum tesztelésére is felhasználhatók.

Természetesen a 23.5. ábrán látható szekvenciadiagramot egyszerűsítettük, így nem ábrázolja a kivételeket. Egy teljes forgatókönyvteszt esetén viszont ezeket is számításba kell venni, és biztosítani kell, hogy az objektumok ezeket is helyesen kezeljék.

23.1.3. Teljesítménytesztelés

Miután egy rendszert teljes mértékben integráltunk, lehetséges a rendszer eredendő tulajdonságainak (2. fejezet) tesztelése, ilyen például a teljesítmény és a megbízhatóság. Teljesítménytesztet azért készítünk, mert biztosítani szeretnénk, hogy a tervezett terhelés mellett a rendszer képes dolgozni. Ez általában tesztek olyan sorozatát foglalja magában, ahol a terhelés mindaddig állandóan nő, amíg a rendszer teljesítménye elfogadhatatlanná nem válik.

Az egyéb tesztfajtákhoz hasonlóan a teljesítménytesztelés is azt a célt szolgálja, hogy egyrészt bemutassa, hogy a rendszer megfelel követelményeinek, másrészt bizonyos esetleges hibákra és hiányosságokra fényt derítsen. Annak tesztelésére, hogy sikerült-e megfelelni a teljesítményre vonatkozó követelményeknek, létre kell hozni egy működési profilt. A működési profil olyan teszt sorozat, amely a rendszer feladatainak reprezentatív mintája, vagyis ha a rendszer tranzakcióinak 90 százaléka A típusú, 5 százaléka B típusú, a többi pedig C, D és E típusú, akkor a működési profilt úgy kell kialakítani, hogy a tesztek túlnyomó többsége A típusú legyen, ellenkező esetben ugyanis nem fogunk pontos képet kapni a rendszer teljesítményéről. A működési profilokat és a megbízhatósági tesztekben betöltött szerepüket a 24. fejezetben taglalom.

Ez a megközelítés azonban a hiányosságtesztelés szempontjából nem szükségképpen a legjobb. A tapasztalatok szerint a hiányosságok felderítésének hatékony módja, ha a tesztet úgy tervezzük, hogy azok a rendszer határait érintsék. A teljesítménytesztelésben ez azt jelenti, hogy a rendszer tervezési határain túlmutató kérésekkel nyomás alá helyezzük a rendszert (ezért hívják *stressztesztelésnek*).

Például egy tranzakciófeldolgozó rendszert úgy is tervezhetek, hogy másodpercenként akár 300 tranzakciót is fel tudjon dolgozni; egy operációs rendszert pedig akár 1000 különálló terminál kezelésére tervezhetek. A stressztesztelés a tervezett maximális terhelésen túl is folytatja ezeket a tesztet mindaddig, amíg a rendszer hibázni nem fog. Az ilyen típusú tesztelésnek két funkciója van:

1. Teszteli a rendszer viselkedését szélsőséges körülmények között. Események nem várt kombinációja olyan körülményeket okozhat, ahol a rendszerre nehezedő terhelés meghaladja a maximális előre látható terhelést. Ilyen körülmények között fontos, hogy a túlterhelés ne okozzon adatvesztést vagy a felhasználói szolgáltatások váratlan eltűnését.
2. Terheli a rendszert, amellyel olyan hiányosságok is napvilágra kerülnek, amelyek normális körülmények között nem jelennek meg. Ezek a hiányosságok normál használat során nem okoznának rendszerhibát, de felléphet a normál körülmények váratlan kombinációja, amit a stressztesztelés előidéz.

A stressztesztelés különösen a processzorok hálózatán alapuló osztott rendszereknél lényeges. Ezek a rendszerek gyakran nagyméretű teljesítményvesztést mutatnak, amikor nagyon leterheltek. A hálózat telítődik a koordinációs adatokkal, amelyeket a különböző folyamatoknak kell cserélniük, így mind lassabban és lassabban lesznek, amíg a többi folyamatból a szükséges adatokat várják.

23.2. KOMPONENSTESZTELÉS

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

A tesztelhető komponensek különbözők lehetnek:

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

Az önálló függvények vagy módszerek a komponensek lehető legegyszerűbb fajtájába tartoznak, ezek tesztelése különböző aktuális paraméterekkel történő hívások sorozatával történik. A következő szakaszban kifejtett tesztesettervezés segítségünkre lehet a tesztek megtervezése során.

Az objektumosztályok tesztelésekor a teszteket úgy kell megtervezni, hogy azok az objektum összes jellemzőjét lefedjék, azaz egy ilyen objektumosztály-tesztelésnek tartalmaznia kell:

1. az objektumhoz kapcsolódó összes művelet különálló tesztelését;
2. az objektumhoz kapcsolódó összes attribútum beállítását és vizsgálatát;
3. az objektum összes lehetséges állapotának a vizsgálatát. Ez azt jelenti, hogy az összes olyan eseményt szimulálni kell, amely állapotváltozást okoz az objektumban.

Gondoljunk például a meteorológiai állomásra a 14. fejezetből, amelynek az interfésze a 23.6. ábrán látható. Ennek csak egy egyszerű attribútuma van, amely egy azonosító. Ez egy konstans, amely a meteorológiai állomás telepítésekor kerül beállításra. Ezért csak olyan tesztre van szükség, ami ellenőrzi, hogy be van-e állítva vagy sem. Viszont tesztesetek definiálása szükséges a `reportWeather`, `calibrate`, `test`, `startup` és `shutdown` módszerekhez. Ideális esetben a módszereket elkülönítve kell tesztelni, de bizonyos esetekben néhány tesztsorozatra is szükség van. Például a `shutdown` teszteléséhez le kell futtatni a `startup` módszert.

WeatherStation
identifier
reportWeather () calibrate (instruments) test () startup (instruments) shutdown (instruments)

23.6. ábra. A meteorológiai állomás objektuminterfésze

A meteorológiai állomás állapotainak teszteléséhez egy állapotmodellt kell használni (14.14. ábra). Ennek a modellnek a használatával azonosítható a tesztelendő állapotátmenetek sorozata és meghatározhatók az ezen átmenetek eléréséhez szükséges eseménysorozatok. Elméletileg az összes állapotátmenet-sorozatot tesztelni kellene, bár a gyakorlatban ez túl költséges lehet. Például a meteorológiai állomás esetében a következő állapotsorozatokat szükséges tesztelni:

Leállítás → Várakozás → Leállítás

Várakozás → Kalibrálás → Tesztelés → Továbbítás → Várakozás

Várakozás → Gyűjtés → Várakozás → Összesítés → Továbbítás → Várakozás

Ha öröklődést használunk, akkor nehezebbé válik az objektumosztályra vonatkozó tesztek megtervezése. Olyan alosztályok esetén, amelyek egy szuperosztály által nyújtott műveleteket örökölnek, az összes örökölt műveletet tesztelni kell ezeknél az alosztályoknál. Ennek az oka, hogy az örökölt művelet feltételezésekkel élhet más műveletekkel és attribútumokkal kapcsolatban, amelyek viszont megváltozhatnak az öröklődés során. Ehhez hasonlóan, ha egy szuperosztály műveletét felülírjuk, ekkor az új műveletet tesztelni kell.

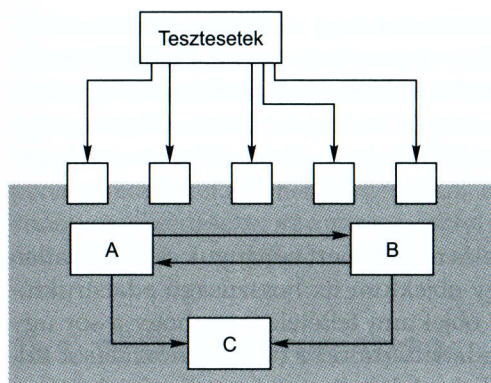
A 23.3.2. alfejezetben tárgyalt ekvivalenciaosztályok gondolata szintén alkalmazható az objektumosztályokra. Az azonos ekvivalenciaosztályba eső tesztek valószínűleg azok lesznek, amelyek az objektumok ugyanazon attribútumait használják. Ezért az ekvivalenciaosztályokat úgy kell azonosítani, hogy melyek inicializálják, melyek kérdezik le és melyek frissítik az összes objektumosztály attribútumait.

23.2.1. Interfésztesztelés

Egy rendszer komponensei sok esetben nem egyszerű függvények vagy objektumok, hanem együttműködő objektumokból álló összetett komponensek. A komponensalapú szoftvertervezésről szóló 19. fejezetben leírtak szerint ezeknek a komponenseknek a funkcionalitását interfészükön keresztül érhetjük el. Az összetett komponensek tesztelése főként annak ellenőrzését jelenti, hogy a komponens interfésze megfelel-e specifikációjának.

A 23.7. ábra bemutatja az interfésztesztelés folyamatát. Tételezzük fel, hogy az A, B és C komponenseket egy nagyobb komponensbe vagy alrendszerbe integráltuk. A teszteseteket nem az egyes komponensekre, hanem a komponensek kombinációjával előállított összetett komponens interfészére alkalmazzuk.

A tesztelés ezen formája különösképpen fontos az objektumorientált és komponensalapú fejlesztés esetén. Az objektumok és az interfészükkel megadott komponensek különböző rendszerek más komponenseivel együtt újrafelhasználhatók. Az összetett komponensben lévő interfészhibákat nem lehet az önálló objektumok tesztelésével kideríteni. A hibák inkább az objektumok közötti interakciók eredményei, mint egy egyedi objektum elszigetelt viselkedései.



23.7. ábra. Interfész tesztelése

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

1. *Paraméter-interfészek.* Ezek olyan interfészek, ahol az adat- vagy néha a függvényreferenciák továbbítódnak az egyik komponenstől a másikhoz.
2. *Osztott memóriájú interfészek.* Ezek olyan interfészek, ahol egy memóriablokk van megosztva az alrendszerek között. Az adatokat az egyik alrendszer a memóriába írja, ahonnan egy másik alrendszer kiolvassa.
3. *Procedurális interfészek.* Ezek olyan interfészek, ahol az egyik alrendszer a más alrendszerek által hívható eljárások egy halmazát bezárja. Objektumok és újrafelhasználható komponensek rendelkeznek ilyen típusú interfészekkel.
4. *Üzenettovábbító interfészek.* Ezek olyan interfészek, ahol egy alrendszer valamilyen szolgáltatást kér egy másik alrendszertől úgy, hogy üzenetet továbbít hozzá. A szolgáltatás lefuttatásával kapott eredményeket pedig egy válasz-üzenet tartalmazza. Néhány objektumorientált rendszer és a kliens-szerver-rendszerek használnak ilyen típusú interfészeket.

Komplex rendszereknél az interfészhibák a leggyakoribb hibák (Lutz, 1993), amelyeket három osztályba sorolhatunk:

1. *Interfész téves alkalmazása.* Egy hívó komponens meghív néhány másik komponent, és hibát követ el az interfészeik használatában. Ez a hiba különösen gyakori a paraméter-interfészeknél, ahol a paraméterek rossz típusúak, rossz sorrendűek lehetnek, illetve akkor, ha nem megfelelő számú paraméter lett átadva.
2. *Interfész félreértelmezése.* Egy hívó komponens félreértelmezi a hívott komponens interfész-specifikációját, és így von le következtetéseket a hívott komponens viselkedésmódjáról. A hívott komponens viszont nem a várakozásainak megfelelően viselkedik, és ez a hívó komponensnél nem várt viselkedést okoz. Például a bináris keresést végző rutint meg lehet hívni egy rendezetlen tömbbel. A keresés ezután hibás lesz.

3. *Időzítési hibák.* Ez valós idejű rendszereknél fordul elő, amelyek osztott memóriájú vagy üzenettovábbító interfészt használnak. Az adat előállítója és a feldolgozója eltérő sebességgel üzemelhet. Ha nem körültekintően figyelünk az interfésztervezésnél, előfordulhat, hogy a feldolgozó idejétmúlt információhoz fér hozzá, mert az információ előállítója nem frissítette az osztott interfész információit.

Az interfészhibákra vonatkozó tesztelés nehéz, mert néhányuk csak szokatlan feltételek esetén jelentkezik. Például egy objektum fix hosszúságú adatstruktúraként implementál egy sort. Egy hívó objektum feltételezheti, hogy a sor úgy lett implementálva, mint egy végtelen adatstruktúra, és nem ellenőrzi a sor túlcsoordulását, amikor egy elem érkezik. Ezt a feltételt csak úgy ellenőrizhetjük a tesztelés során, hogy olyan teszteseteket készítünk, amelyek kierőltetik a sor túlcsoordulását, és ez az objektum helytelen viselkedését okozza valamilyen detektálható módon.

Egy további probléma is felmerülhet a különböző modulokban vagy objektumokban található hibák egymásra hatása miatt. Egy objektumban a hiba talán csak akkor detektálható, ha néhány másik objektum nem várt módon viselkedik. Például egy objektum meghívhat valamilyen másik objektumot bizonyos szolgáltatás igénybevételére, és feltételezi, hogy a válasz helyes. Ha a kiszámított érték kapcsán valamilyen félreértés történt, akkor a visszaadott érték érvényes lehet, de helytelen. Erre csak akkor derül fény, amikor a későbbi számítások során valamilyen hiba történik.

Megfontolandó az interfésztesztelés néhány általános irányelve:

1. Vizsgálni kell a tesztelendő kódot, és minden egyes külső komponenshívást egyértelműen rögzíteni kell. A tesztek egy részét úgy kell összeállítani, hogy a külső komponensek paramétereinek az értékei a tartományuk legtávolabbi végére essenek. Ezek a szélsőséges értékek nagy valószínűséggel felfedik az interfész-inkonzisztenciákat.
2. Amikor mutatókat adunk át egy interfészen keresztül, mindig tesztelni kell az interfészt null értékű paraméterekkel.
3. Amikor egy komponens procedurális interfészen keresztül kerül meghívásra, olyan tesztek kellene, melyek esetleg a komponens hibázását okozzák. A hiba működésre vonatkozó feltételezés egyike a leggyakoribb specifikációs félreértelmezéseknek.
4. Üzenettovábbító rendszereknél használható az előző alfejezetben tárgyalt stressztesztelés. Tervezni kell olyan tesztek, amelyek sokkal több üzenetet generálnak, mint amennyi a gyakorlatban elő fog fordulni. Ilyen módon az időzítési problémákra derülhet fény.
5. Amikor számos komponens együttműködése osztott memórián keresztül történik, akkor szükségesek olyan tesztek, amelyek megváltoztatják a komponensek aktiválódási sorrendjét. Ezek a tesztek felfedhetnek implicit feltételezéseket a sorrendre vonatkozóan, amelyeket a programozók beépítettek az osztott adatok kezelésébe.

A statikus technikák gyakran költséghatékonyabbak, mint az interfészhibák felderítésére vonatkozó tesztelések. Egy erősen típusos nyelv, mint például a Java, lehetővé teszi, hogy a fordító számos interfészhibát felderíthessen. Míg egy gyengébb nyelv, például a C, statikus elemzőt (például a LINT, 22. fejezet) használ az interfészhibák detektálásához. A program átvizsgálása során a komponensinterfészekre lehet koncentrálni, és az interfész feltételezett viselkedésére vonatkozó kérdéseket az átvizsgálási folyamat során fel lehet tenni.

23.3. TESZTESETERVEZÉS

A tesztesettervezés a rendszer- és komponensvizsgálás azon része, amikor a rendszer tesztelését végző tesztesetek (az inputok és az előre jelzett outputok) tervezése történik. A folyamat célja a program hiányosságainak hatékony felderítésére alkalmas tesztesetek kialakítása.

Egy teszteset kialakításához ki kell választanunk a rendszernek vagy a komponensnek azt a jellemzőjét, amelyet tesztelni szeretnénk. Meg kell határozni a jellemző futtatásához szükséges inputokat, dokumentálni kell az előre jelzett outputokat vagy outputtartományokat, és ahol csak lehetőség van rá, meg kell tervezni az aktuális és az elvárt kimenetek egyezőségét vizsgáló automatikus ellenőrzéseket. A tesztesetek tervezésekor különböző lehetőségek közül választhatunk:

1. *Követelményalapú tesztelésről* akkor beszélünk, ha a teszteseteket a rendszer követelményeinek tesztelése céljából készítjük. Erre leginkább a rendszer-tesztelési lépésben van szükség, mert a követelményeket általában több komponens együttesen valósítja meg. Minden követelményhez azonosítanunk kell azokat a teszteseteket, amelyekkel megmutatható, hogy a rendszer teljesíti azt.
2. *Partíciós tesztelés* esetén azonosítjuk az input és output partíciókat, és úgy tervezzük meg a teszteseteket, hogy a rendszer minden partícióból lefuttatja az inputokat, és minden partícióba generál outputot. A partíciók hasonló jellemzőkkel rendelkező adatok csoportjai, például a negatív számok, a 30 karakternél rövidebb nevek, egy menüben kiválasztott elemek hatására bekövetkező események stb.
3. A *strukturális tesztelés* során a program részeit vizsgáló teszteseteket a program szerkezetének ismeretében készítjük el. Lényeges, hogy a tesztelés során meg kell próbálni minden utasítást legalább egyszer lefuttatni. A strukturális tesztelés az ezt lehetővé tevő tesztesetek meghatározásában nyújt segítséget.

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

23.3.1. Követelményalapú tesztelés

A követelménytervezés 6. fejezetben leírt egyik fő alapelve, hogy a követelményeknek tesztelhetőeknek kell lenniük, azaz oly módon kell azokat leírni, hogy olyan tesztet lehessen készíteni, amely alapján egy külső szemlélő is ellenőrizni tudja, hogy az adott követelmény teljesült-e, vagy sem. Épp ezért a követelményalapú tesztelés olyan szisztematikus tesztesettervezést jelent, amikor az egyes követelményekből teszt sorozatokat származtat. Ez a tesztelési mód inkább validáció, mintsem hiányosságtesztelés – azt próbáljuk bemutatni, hogy a rendszer megfelelően megvalósítja a követelményekben leírtakat.

Példaként tekintsük a 6. fejezetben bemutatott LIBSYS könyvtári rendszer követelményeit:

1. A felhasználónak képesnek kell lennie a teljes kiindulási adatbázisban vagy annak egy részhalmazában történő keresésre.
2. A rendszernek biztosítania kell a megfelelő megjelenítő programokat, hogy a felhasználó el tudja olvasni a dokumentumtárban lévő dokumentumokat.
3. Minden rendeléshez tartoznia kell egy egyedi azonosítónak (ORDER_ID), és lehetővé kell tenni, hogy a felhasználó átmásolhassa azt egy permanens tárolóterületre.

Feltételezve, hogy a keresési funkciót már tesztelték, az első követelmény számára kialakított lehetséges tesztek az alábbiak:

- Végezzünk kereséseket olyan elemekre, amelyekről tudjuk, hogy benne vannak, vagy nincsenek benne az adatbázisunkban, amikor az adatbázisaink egyetlen adatbázisból állnak.
- Végezzünk kereséseket olyan elemekre, amelyekről tudjuk, hogy benne vannak, vagy nincsenek benne az adatbázisunkban, amikor az adatbázisaink két adatbázisból állnak.
- Végezzünk kereséseket olyan elemekre, amelyekről tudjuk, hogy benne vannak, vagy nincsenek benne az adatbázisunkban, amikor az adatbázisaink több mint két adatbázisból állnak.
- Válasszunk ki adatbázisaink közül egyet, és végezzünk kereséseket olyan elemekre, amelyekről tudjuk, hogy benne vannak, illetve nincsenek benne.
- Válasszunk ki adatbázisaink közül egynél többet, és végezzünk kereséseket olyan elemekre, amelyekről tudjuk, hogy benne vannak, illetve nincsenek benne.

A fentiekből látható, hogy egy követelmény tesztelése nemcsak egyetlen teszt írását jelenti, a követelmények lefedését általában csak több teszt segítségével tudjuk biztosítani.

A LIBSYS-rendszer további követelményei számára hasonlóan kell a tesztek kialakítani. A második követelmény számára olyan tesztek kell írunk, ame-

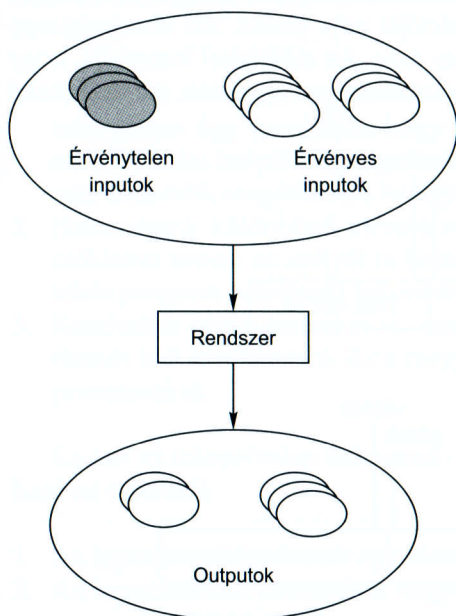
lyek a rendszer által kezelt összes dokumentumfajta esetén ellenőrzi, hogy azok megfelelően megjelennek-e. A harmadik követelmény ennél egyszerűbb. Ennek teszteléséhez mindössze arra van szükség, hogy szimuláljuk a rendelések elküldését, és ellenőrizzük, hogy a rendelésazonosító minden esetben jelen van, és egyedi.

23.3.2. Partíciós tesztelés

Egy program inputadatai többnyire számos különböző osztályba esnek. Ezek rendelkeznek valamilyen közös jellegzetességgel, mint például pozitív számok, negatív számok, menüválasztások stb. A programok rendes körülmények között egy osztály minden tagjára hasonló módon viselkednek, azaz ha egy két pozitív számot igénylő számítást végző programot tesztelünk, akkor elvárjuk, hogy a program bármely két pozitív számra ugyanúgy működjön.

Az azonos viselkedés miatt ezeket az osztályokat néha *ekvivalenciaosztályok*-nak vagy *-tartományoknak* nevezik (Bezier, 1990). A tesztesettervezés szisztematikus megközelítése ezeknek a rendszer vagy a komponens ekvivalenciaosztályainak azonosításán alapszik. A teszteseteket ezek után úgy tervezik, hogy a bemenetek és a kimenetek ezekbe az osztályokba esnek. A partíciós tesztelést mind rendszerek, mind komponensek esetén alkalmazhatjuk a tesztesetek kialakítása érdekében.

A 23.8. ábrán az egyes ekvivalenciaosztályokat ellipszisszel ábrázoljuk. Az input ekvivalenciaosztályok az adatoknak olyan halmazai, ahol minden halmaz-



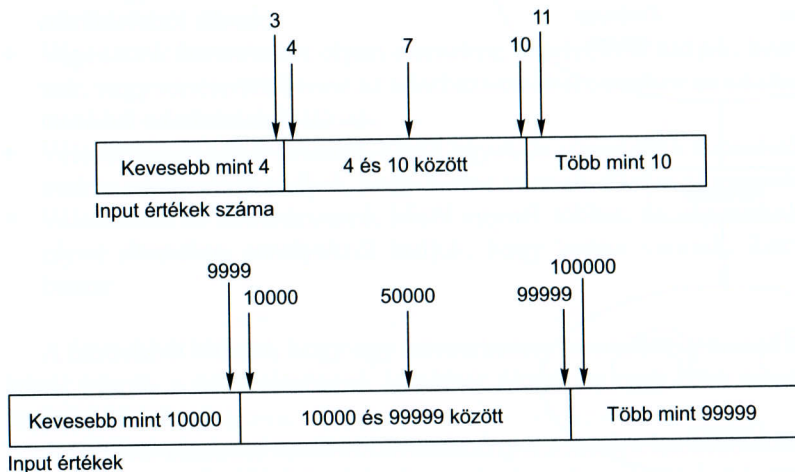
23.8. ábra. Ekvivalenciaosztályozás

elemet hasonló módon kell feldolgozni. Az output ekvivalenciaosztályok pedig olyan programkimenetek, amelyeknek valamilyen közös jellegzetességük van, ezért külön osztálynak tekinthetők. Továbbá meghatározhatók olyan osztályok is, ahol az inputok kívül esnek a választott többi osztályon. Ezek segítségével ellenőrizhető, hogy a program helyesen kezeli-e az érvénytelen inputot. Az érvényes és az érvénytelen inputok szintén ekvivalenciaosztályokat alkotnak.

Először meg kell határozni az osztályok halmazát, majd minden osztályból teszteseteket kell választani. A tesztesetek kiválasztásánál kövessük azt az irányelvet, hogy az osztály határáról és közepéről is választunk teszteseteket. Ennek az irányelvnek az ésszerű magyarázata az, hogy a tervezők és a programozók a rendszer fejlesztésénél hajlamosak csak az inputok tipikus értékeit figyelembe venni. A határértékek többnyire nem tipikusak, ezért kerülhetik el a fejlesztők figyelmét (például a nulla a többi nemnegatív számtól eltérően viselkedhet). A programhibák akkor fordulnak elő gyakran, amikor ilyen nemtipikus értékeket dolgozunk fel.

Ezek az ekvivalenciaosztályok a programspecifikáció vagy a felhasználói dokumentáció felhasználásával azonosíthatók, illetve a tesztelő azon tapasztalatára támaszkodva, hogy az input értékosztályok közül melyek jöhetnek leginkább számításba a hibák felderítésében. Tekintsünk például egy programspecifikációt, amely rögzíti, hogy a program 4 és 8 közötti számban vár olyan inputokat, melyek 10 000-nél nagyobb ötjegyű egészek. A 23.9. ábra az ehhez az esethez kapcsolódó ekvivalenciaosztályokat és a lehetséges bemeneti tesztadatok értékeit mutatja.

A tesztesetek levezetését a 23.10. ábrán látható kereső komponens specifikációjával illusztrálom. Ez a komponens egy elemsorozatban keres egy adott elemet (a kulcsot). Az adott elem sorozatbeli pozícióját adja vissza. Ezt absztrakt módon, elő- és utófeltételek segítségével adtam meg. Az előfeltétel kimondja, hogy a keresőeljárást nem üres sorozaton történő működésre tervezték. Az utófeltétel



23.9. ábra. Ekvivalenciaosztályok

```
procedure Search ( Key : ELEM ; T : SEQ of ELEM ;
  Found : in out BOOLEAN; L : in out ELEM_INDEX );
```

Pre-condition

- a sorozat legalább egyelemű
- T'FIRST <= T'LAST

Post-condition

- az elemet megtaláltuk, és az L hivatkozik rá
- (Found **and** T (L) = Key)

or

- az elem nincs a sorozatban
- (**not** Found **and**
- not** (**exist** i, T'FIRST >= i <= T'LAST, T (i) = Key))

23.10. ábra. Keresőeljárás specifikációja

pedig kimondja, hogy a Found változó akkor lesz beállítva, ha a kulcselem a sorozatban van. A kulcselem pozícióját az L index jelzi. Az index definiálatlan, ha az elem nincs a sorozatban.

Ebből a specifikációból két nyilvánvaló ekvivalenciaosztály azonosítható:

1. inputok, ahol a kulcselem a sorozat tagja (Found = true);
2. inputok, ahol a kulcselem nem tagja a sorozatnak (Found = false).

Ha olyan programokat tesztelünk, amelyek sorozatokat, tömböket vagy listákat tartalmaznak, a tesztervek készítésekor a következőket érdemes figyelembe venni:

1. A szoftver tesztelését egyelemű sorozatokkal végezzük. A programozók természetesen úgy gondolják, hogy a sorozatok több értékből állnak, és néha ezt a feltevést beépítik a programjukba. Következésképpen a program esetleg nem működik megfelelően, ha egyértékű sorozatokkal dolgozunk.
2. Használjunk különböző méretű sorozatokat a különböző tesztek során. Ez csökkenti annak az esélyét (a bemenet véletlenszerű jellege miatt), hogy egy hibás program véletlenül helyes eredményt adjon.
3. Készítsünk olyan tesztek is, amelyeknél a sorozat első, középső és utolsó elemét kell kiválasztani. Ez a megközelítés feltárja az osztályhatárokon lévő problémákat.

Ezeket az irányelveket felhasználva két további ekvivalenciaosztály definiálható az inputnál:

1. Az input sorozatnak csak egy eleme van.
2. Az input sorozat elemszáma nagyobb mint 1.

Ezt követően ezen osztályok kombinálásával további osztályokhoz juthatunk, például az egynél nagyobb elemszámú, de a keresett elemet nem tartalmazó osztályhoz. A 23.11. ábra a keresést végző komponens teszteléséhez általam azonosított osztályok láthatók.

Tömb	Elem	
Egy érték	A sorozatban van	
Egy érték	Nincs a sorozatban	
Több mint 1 érték	Első elem a sorozatban	
Több mint 1 érték	Utolsó elem a sorozatban	
Több mint 1 érték	Középső elem a sorozatban	
Több mint 1 érték	Nincs a sorozatban	
Input sorozat (T)	Kulcs (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

23.11. ábra. A keresőrutin ekvivalenciaosztályai

Ezek az ekvivalenciaosztályokon alapuló lehetséges tesztesetek egy halmazát szintén láthatjuk a 23.11. ábrán. Ha a kulcselem nincs a sorozatban, az L értéke definiálatlan ('??'). A teszteseteknél alkalmaztuk azt az irányelvet, hogy a különböző tesztesetek eltérő méretűek legyenek.

A keresőeljárás teszteléséhez használt inputértékek halmaza nem teljes körű. Az eljárás esetleg hibázhat, ha a bemenő sorozat az 1, 2, 3 és 4 elemeket tartalmazza. Bár ennek ellenére ésszerű feltételezni, hogy ha az osztály egy elemének feldolgozása nem eredményez hibát, akkor az osztály többi tagja sem fog. Természetesen hibák még létezhetnek. Nem azonosítottunk minden ekvivalenciaosztályt, hibázhattunk az ekvivalenciaosztályok felderítésében, esetleg a tesztadatokat rosszul készíthettük elő.

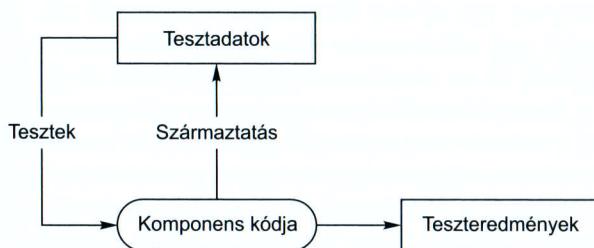
23.3.3. Struktúrateszt

A struktúrateszt (23.12. ábra) esetében a teszteket a szoftver struktúrájának és implementációjának ismeretében készítjük. Ezt a megközelítést néha hívják *fehér doboz* tesztelésnek, *üvegdoboz* tesztelésnek vagy *tiszta doboz* tesztelésnek, megkülönböztetve a *fekete doboz* teszteléstől.

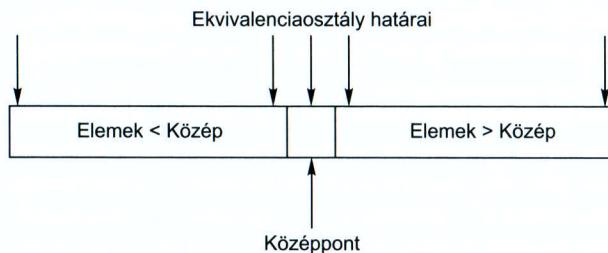
A komponensben lévő algoritmus megértése segítségünkre lehet további ekvivalenciaosztályok és tesztesetek azonosításában. Ennek illusztrálásához a keresőeljárás specifikációjában leírtakat (23.10. ábra) egy bináris keresési algoritmus-sal (23.14. ábra) implementáltam. Természetesen ennek szigorúbb előfeltételei vannak. A sorozatnak tömbnek kell lennie, rendezettnek kell lennie, és az alsó határnál lévő értéknek kisebbnek kell lennie, mint a felső határnál lévőnek.

A keresőeljárás kódjának vizsgálatával láthatjuk, hogy a bináris keresés a keresési teret három részre osztja. E részek mindegyike meghatároz egy ekvivalenciaosztályt (23.13. ábra). A kód vizsgálatához olyan teszteseteket kell választani ezekből az osztályokból, ahol a kulcs az osztály határán helyezkedik el.

Ez a tesztesetsorozat módosításához vezet, ahogyan az a 23.15. ábrán látható. Megfigyelhető, hogy a bemeneti tömböt növekvő sorrendbe rendeztem, és olyan további tesztekkel bővítettem, amelyekben a kulcs a tömb középpontjával határos.



23.12. ábra. Struktúrateszt



23.13. ábra. A bináris keresés ekvivalenciaosztályai


```

class BinSearch {
// Ez egy bináris keresőfüggvénynek olyan megvalósítása, amely objektumok egy rendezett
// tömbjét és egy kulcsot kap bemenő paraméterként, eredményül pedig egy 2 attribútummal
// rendelkező objektumot ad vissza, nevezetesen:
// index – a tömbindex értéke,
// found – boolean (logikai) érték, ami jelzi, hogy a kulcs benne van-e a tömbben vagy sem.
// Azért egy objektumot ad vissza, mert Javában lehetetlen egy
// függvénynek cím szerint alaptípusokat átadni és visszaadni két értéket.

    public static void search ( int key, int [] elemArray, Result r )
    {
1.      int bottom = 0 ;
2.      int top = elemArray.length - 1 ;
        int mid;
3.      r.found = false;
4.      r.index = -1 ;
5.      while (bottom <= top)
        {
6.          mid = (top + bottom)/2 ;
7.          if (elemArray [mid] == key)
            {
8.              r.index = mid ;
9.              r.found = true ;
10.             return ;
            } // if rész
        else
        {
11.             if (elemArray [mid] < key)
12.                 bottom = mid + 1 ;
            else
13.                 top = mid -1 ;
        }
    } // while ciklus
14.    } // search
    } // BinSearch

```

23.14. ábra. Egy bináris keresőrutin Java-implementációja

Input sorozat (T)	Kulcs (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29,	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

23.15. ábra. Tesztesetek a bináris kereséshez

23.3.4. Útvonaltesztelés

Az útvonaltesztelés strukturális tesztelési stratégia. Célja, hogy minden független végrehajtási útvonalat kipróbáljunk egy komponens vagy program esetén. Ha minden független útvonalon végigmentünk, akkor a komponens minden utasítása legalább egyszer végrehajtott. Továbbá minden feltételes utasítást tesztelt mind igaz, mind hamis esetekre. Egy objektumorientált fejlesztési folyamatban az útvonaltesztelést általában akkor alkalmazzák, amikor a módszerek objektumokkal való kapcsolatát tesztelik.

Egy programban található utak száma általában arányos a program méretével. Amint a modulokat a rendszerbe integráljuk, a strukturális tesztelési technikák használata kivitelezhetetlenné válik. Ezért az útvonal-tesztelési technikákat leginkább a tesztelési folyamatnak az egység-, illetve modultesztelési fázisaiban használják.

Az útvonaltesztelés nem teszteli a programon keresztül vezető összes útnak minden egyes lehetséges kombinációját. Minden egyes komponensre, eltekintve a nagyon triviális ciklusmentes esetektől, ez lehetetlen feladat. Ciklusokat tartalmazó programok esetén pedig végtelen számú útvonal-kombináció létezik. Hibák, amelyek csak szokatlan útvonal-kombinációk esetén merülnek fel, továbbra is jelen lehetnek, annak ellenére, hogy minden programutasítás legalább egyszer végrehajtott.

Az útvonaltesztelés kezdő pontja egy programfolyamat-gráf. Ez a programon keresztül vezető utak vázmodellje. Egy folyamatgráf csomópontokból áll, amelyek döntéseket reprezentálnak, és az élek mutatják a vezérlés irányát. A folyamatgráfot a program vezérlőutasításainak a megfelelő diagramra történő átírásával állítjuk össze. Ha nincs goto utasítás a programban, akkor a folyamatgráf előállítása egyszerű. A szekvenciális utasítások (értékadások, eljáráshívások és I/O-utasítások) pedig kihagyhatók a folyamatgráf felépítéséből. A feltételes utasítások (if-then-else vagy case) mindegyik ága különálló útként jelenik meg a gráfban, a ciklusokat pedig a ciklusfeltételt reprezentáló csomópontba visszamutató nyíllal jelöljük. Ciklusok és feltételes ágak illusztrálása látható a bináris keresés rutinjának folyamatgráfján (23.16. ábra). A 23.14. ábrán látható program és az itt látható folyamatgráf közötti minél nyilvánvalóbb megfeleltetés érdekében a különböző csomópontok száma megegyezik a program megfelelő sorának számával.

Az útvonaltesztelés célja biztosítani, hogy minden egyes független programútvonalon legalább egyszer végigmenjünk. Egy független programútvonal olyan út, amely a folyamatgráfban legalább egy új élen átmegy. A program szemszögéből ez azt jelenti, hogy egy vagy több új feltételt kezelünk. A feltételeknek mind az igaz, mind a hamis ágát le kell futtatni.

A bináris keresési eljárás folyamatgráfja a 23.16. ábrán látható, ahol minden csomópont a program egy olyan sorát reprezentálja, ami egy végrehajtható utasítást tartalmaz. A folyamatot követve láthatjuk, hogy bináris keresés folyamatgráfjában a következő utak vannak:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14

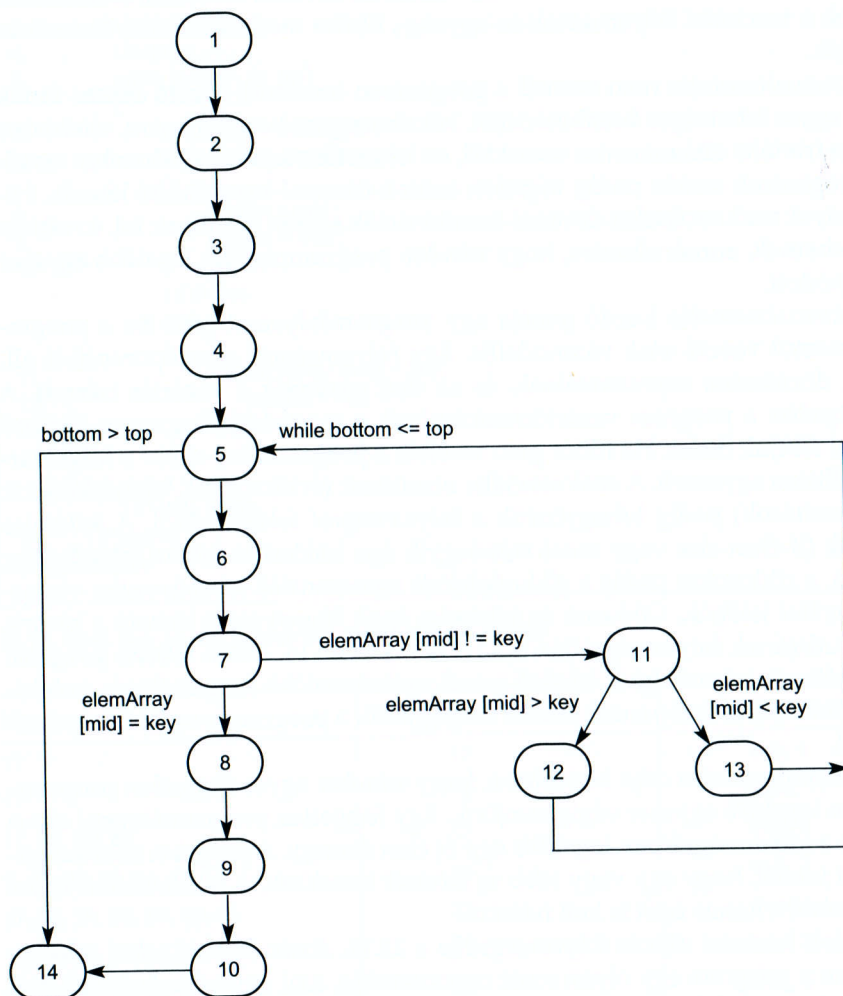
1, 2, 3, 4, 5, 14

1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...

1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ...

Ha ezek közül mind lefutott, akkor biztosak lehetünk abban, hogy a módszer minden utasítása legalább egyszer végrehajtódott, és minden ágat kipróbáltunk igaz és hamis értékekre is.

Egy programban a független utak száma meghatározható a program folyamatgráfja *ciklomatikus komplexitásának* (McCabe, 1976) a számításával. Azoknál a programoknál, amelyek nem tartalmaznak goto utasítást, a ciklomatikus komp-



23.16. ábra. A bináris keresés rutinjának folyamatgráfja

lexitás eggyel több, mint a programban található feltételek száma. Az egyszerű feltétel olyan logikai kifejezés, amely nem tartalmaz logikai operátorokat. Ha a program „and” és „or” operátorokat tartalmazó összetett feltételeket tartalmaz, akkor a ciklomatikus komplexitás kiszámításához az összetett feltételben szereplő egyszerű feltételek darabszámát kell használni.

Ezért, ha van hat if utasításunk és egy while ciklusunk csupa egyszerű feltételes kifejezéssel, akkor a ciklomatikus komplexitás 8. Ha egy feltételes kifejezés összetett (például if A and B or C), akkor ez több (jelen példában 3) egyszerű feltételként számít be, a ciklomatikus komplexitás ezért 10. A bináris keresőrutin (23.14. ábra) ciklomatikus komplexitása 4, mert csupán három egyszerű feltételt tartalmaz, az 5., a 7. és a 11. sorban.

Miután a ciklomatikus komplexitás kiszámításával a kódban felfedeztük a független utakat, a következő lépés a tesztesetek megtervezése úgy, hogy ezen utak mindegyikének végrehajtására sor kerüljön. A valamennyi programútvonal teszteléséhez szükséges tesztesetek minimális száma megegyezik a ciklomatikus komplexitás értékével.

A bináris keresőrutin esetében a tesztesetek megtervezése egyértelmű. Bár amikor a programok komplex elágazó struktúrával rendelkeznek, nehéz megmondani, hogy egy bizonyos teszteset hogyan kerül feldolgozásra. Ilyen esetekben dinamikus programelemzőt használhatunk a program futási profiljának felderítésére.

A dinamikus programelemzők olyan tesztelőeszközök, amelyek a fordítókcal működnek együtt. A fordítás során kiegészítő tárgy kódú utasítások adódnak a generált kódhoz. Ezek számolják, hogy az egyes programutasítások hányszor kerültek futtatásra. Miután a program lefutott, futási profil készíthető, ami megmutatja, hogy a program mely részei futottak, illetve nem futottak a különböző tesztesetek során. Ez a futási profil így a nem tesztelt programrészek felderítésére használható.

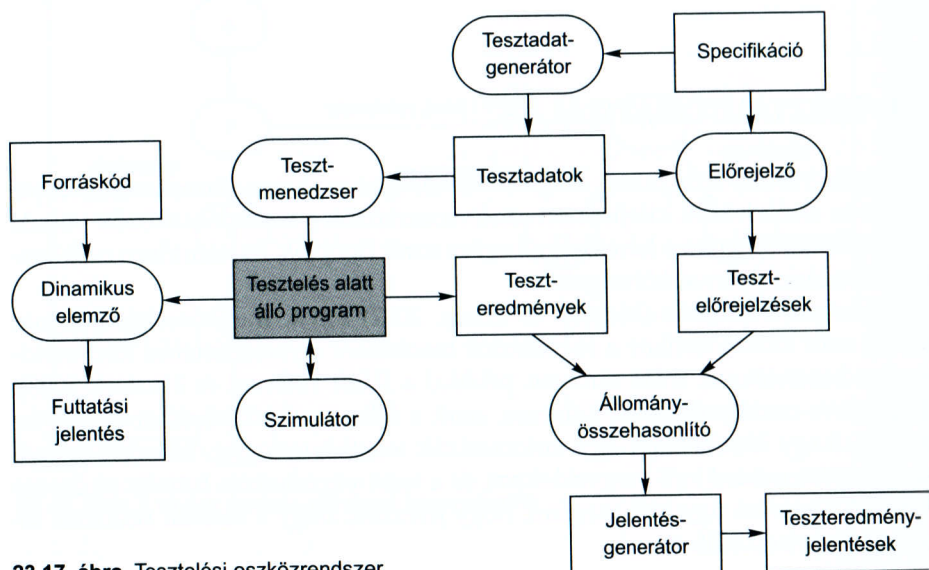
23.4. TESZTAUTOMATIZÁLÁS

A tesztelés a szoftverfolyamat drága és fáradságos szakasza. Ennek eredményeképpen az elsők között kifejlesztett szoftvereszközök a tesztelőeszközök voltak. Ezek az eszközök mára a lehetőségek egész sorát nyújtják, és jelentősen csökkentik a tesztelési folyamat költségeit.

A tesztautomatizálás (Mosley és Posey, 2002) egyik megközelítéséről már ejtettem szót akkor, amikor a regressziós tesztelésre olyan tesztelési keretrendszereket használtunk, mint amilyen például a JUnit (Massol és Husted, 2003). A JUnit Java-osztályok olyan halmaza, amit a felhasználó kibővíthet annak érdekében, hogy létrehozzon egy automatizált tesztkörnyezetet. Minden egyedi tesztet objektumként kell megvalósítani, és a teszt végrehajtója futtatja az összes tesztet. A teszteteket úgy kell megírni, hogy jelezzék, hogy a tesztelt rendszer elvárt módon viselkedik-e.

A 23.17. ábra néhány eszközt és a köztük található interakciókat mutatja, amelyek megtalálhatók egy tesztelési eszközrendszerben. A beépített eszközök:

1. *Tesztmenedzser.* A programtesztek futtatását menedzseli. A tesztmenedzser nyomon követi a tesztadatokat, a várt eredményeket és a tesztelt program-eszközöket. A JUnithoz hasonló tesztautomatizálási keretrendszerek példát szolgáltatnak rájuk.
2. *Tesztadat-generátor.* Tesztadatokat állít elő a tesztelt program számára. Ez megvalósítható egy adatbázisból történő adatleválogatással vagy minták használatával megfelelő formájú véletlen tesztadatok generálásával.
3. *Előrejelző.* A várt teszteredményekre vonatkozó előrejelzéseket állít elő. Az előrejelzők lehetnek korábbi programverziók vagy prototípusrendszerek. A teszteléssel együtt jár, hogy az előrejelző és a tesztelt program párhuzamosan fut. Az outputjaik közötti különbségeket pedig kiemeljük.
4. *Állomány-összehasonlító.* A programtesztek eredményeit hasonlítja össze a korábbi teszteredményekkel, és jelzi a köztük található különbségeket. Az összehasonlító elengedhetetlenek a regressziós teszteknel, ahol a régi és az új verziók futási eredményeit kell összehasonlítani. Az eredmények közötti különbségek mutatják az új verzióval kapcsolatos potenciális problémákat.
5. *Jelentésgenerátor.* Jelentésdefiníciókat és a teszteredmények generálásához szükséges eszközöket nyújtja.
6. *Dinamikus elemző.* A programot kiegészítő kóddal látja el, hogy az egyes utasítások futtatásának számát mérni lehessen. Miután a tesztek lefutottak, egy futási profil generálódik, amely megmutatja, hogy az egyes utasítások hány-szor futottak le.
7. *Szimulátor.* Különböző típusú szimulátorokat lehet biztosítani. A célszimulátorok azt a gépet szimulálják, amelyen a program futtatásra kerülne. A fel-



23.17. ábra. Tesztelési eszközrendszer

használói interfész-szimulátorok olyan szkriptvezérelt programok, melyek több egyidejű felhasználói interakciót szimulálnak. Szimulátorok használata az I/O-nál azt jelenti, hogy a tranzakciósorozatok időzítése ismételhető.

Nagy rendszerek esetén a tesztelési követelmények a fejlesztendő alkalmazástól függenek. Következésképpen a tesztelési segédeszközöket mindig az egyes rendszerek teszttervéhez kell igazítani. Példák az adaptációkra:

1. Sajátos alkalmazás jellegzetességek teszteléséhez új eszközök felvétele, illetve néhány meglévő teszteszköz szükségtelenné válása.
2. A felhasználói interfész szimulálásához szükség lehet szkriptek írására, és mintákat lehet definiálni a tesztadat-generátorok számára. Továbbá lehet, hogy jelentésmintákat is kell definiálni.
3. A várt teszteredmények egy csoportját nekünk kell előkészítenünk, ha nincs elérhető korábbi programverzió, amely előrejelzőként működhetne.
4. Lehet, hogy speciális rendeltetésű állomány-összehasonlítókat kell írunk, amelyek magukban foglalják az állományban található teszteredmények struktúrájára vonatkozó ismeretet.

Többnyire jelentős mennyiségű időt és erőfeszítést igényel egy átfogó tesztelési eszközrendszer létrehozása. A 23.17. ábrán látható teljes eszközrendszerek ezért csak nagy rendszerek fejlesztésénél használatosak. E rendszerek számára az átfogó tesztelés költsége a teljes fejlesztési költség 50 százaléka is lehet, ezért gazdaságos befektetni egy jó minőségű CASE-eszközbe a tesztelés támogatására. Azonban a különféle rendszerek különféle tesztátogatást igényelnek, és nem biztos, hogy rendelkezésre állnak kulcsrakész tesztelési eszközrendszerek. Rankin (Rankin, 2002) egy ilyen helyzetet vázol az IBM-nél, és leírja, hogyan tervezték meg az e-business szerverhez készült teszteléstámogató rendszert.

Kulcsfogalmak

- A tesztelés mindig csak a hibák programban való jelenlétét tudják megmutatni, sosem bizonyítják, hogy már nem maradtak hibák.
- A komponens tesztelés a komponens fejlesztőjének felelőssége. Egy különálló tesztelőcsapat általában a rendszer tesztelését végzi.
- Az integrációs tesztelés a rendszer tesztelésének kezdeti tevékenysége, amikor az integrált komponensekben keresünk hibákat. A kiadástesztetek a rendszer egy kiadható verzióját ellenőrzi abból a szempontból, hogy megfelel-e követelményeinek.
- A tesztelés során meg kell próbálnunk „elrontani” a rendszert. Ebben egyrészt tapasztalatainkra, másrészt különböző irányelvekre támaszkodhatunk, amelyek megkönnyítik a más rendszerekben hatékonyan bizonyult hiányosságfelderítő tesztesetek fajtái közül történő választást.

- Az interfésztesztelés az összetett komponensek interfészeinek hiányosságait hivatott feltérképezni. Ilyen hiányosságok bekövetkezhetnek a specifikáció félreolvasásából, félreértéséből is.
- Az ekvivalenciaosztályozás egy lehetőség tesztesetek előállítására. Ez az input és az output adathalmazokban lévő osztályok megtalálásán és a program ezen osztályokból származó értékekkel történő futtatásán alapszik. Az az érték, amelyik nagy valószínűséggel vezet sikeres teszthez, többnyire egy osztály határáról származó érték.
- A strukturális tesztelés a programon átvezető útvonalak elemzésén alapul, és ennek az elemzésnek a felhasználása segíthet a tesztesetek kiválasztásában.
- A tesztautomatizálás a tesztelés során alkalmazható szoftvereszközök biztosításával csökkenti a tesztelés költségeit.

További irodalom

How to Break Software: A Practical Guide to Testing. Korántsem elméleti, hanem sokkal inkább gyakorlati könyv a szoftvertesztelésről, amelyben a szerző bemutatja a tesztek tervezéséhez kidolgozott tapasztalati irányelveket. (Whittaker, J. A., 2002, Addison-Wesley.)

„Software Testing and Verification”. Az *IBM Systems Journal* különszáma, benne számos tesztelésről szóló cikkel, köztük egy kiváló áttekintéssel és a tesztmetrikákról és tesztautomatizálással szóló cikkekkel. (*IBM Systems Journal*, 41(1), January, 2002.)

Testing Object-oriented Systems: Models, Patterns and Tools. Roppan nagy könyv (több mint 1000 oldal), amely az objektumorientált tesztelés teljes áttekintését nyújtja. A teljességéből következik, hogy ne ez legyen az elsőként elolvasott könyv az objektumorientált teszteléssel kapcsolatban (a legtöbb objektumorientált fejlesztéssel foglalkozó könyvnek van egy tesztelési fejezete), viszont ez egyértelműen roppan fontos könyv az objektumorientált tesztelés témájában. (Binder, R. V., 1999, Addison-Wesley.)

„How to design practical test cases”. Egy „hogyan” cikk a tesztesetek tervezéséről egy, a kevés hibát tartalmazó szoftverek előállításában már jó nevet szerzett japán cégnél dolgozó szerzőtől. (Yamura, T., *IEEE Software*, 15(6), November, 1998.)

Feladatok

- 23.1. Magyarázza el, hogy miért tudjuk a teszteléssel a hibáknak csak a jelenlétét, nem pedig a hiányát felderíteni.
- 23.2. Hasonlítsa össze a fentről lefelé és a lentől felfelé történő integrációt és tesztelést, felsorolva az architektúrális tesztelésnek, a rendszer egy verziójának felhasználók számára történő bemutatásának és a tesztek gyakorlati megvalósításának előnyeit és hátrányait. Magyarázza el, hogy a legtöbb

nagy rendszer integrációja esetén miért a „fentről lefelé” és a „lentől felfelé” technikák keverékét használják a gyakorlatban.

- 23.3. Mi az a regressziótesztelés? Hogyan egyszerűsíti ezt az automatizált tesztek és a JUnithoz hasonló keretrendszerek használata?
- 23.4. Írjon forgatókönyvet, amely alapjául szolgálhat a 14. fejezet példájában használt meteorológiai állomás rendszere számára történő tesztek származtatásának.
- 23.5. A 8.14. ábra szekvenciadiagramját forgatókönyvként tekintve, tegyen javaslatot a LIBSYS-rendszer elektronikus elemei kibocsátásának tesztelésére.
- 23.6. Milyen problémák merülnek fel az elosztott adatbázisrendszer számára kialakítandó teljesítménytesztek kapcsán, mint amilyen a LIBSYS?
- 23.7. Magyarázza el, miért szükséges az interfésztesztelés, amikor az egységeket komponens tesztelés és programátvizsgálás segítségével alaposan validálták.
- 23.8. Az objektumtesztelés itt megfogalmazott elvei szerint tervezze meg a 8.5. ábrán látható állapotmodellel rendelkező mikrohullámú sütő állapotait tesztelő teszteseteket.
- 23.9. Megbízták azzal, hogy egy Paragraph objektumban tesztelje a catWhiteSpace műveletet, amely az adott bekezdésben az üres karakterek sorozatát egyetlen üres karakterre cseréli. Azonosítsa az ekvivalenciaosztályokat, és származtassa ebből a catWhiteSpace művelet teszt sorozatát.
- 23.10. Soroljon fel három olyan helyzetet, amikor egy program összes független útvonalának tesztelése nem derít ki programhibát.