ÖTÖDIK RÉSZ

VERIFIKÁCIÓ ÉS VALIDÁCIÓ

A program tesztelése a leggyakoribb mód annak ellenőrzésére, hogy az megfelel-e specifikációjának, és azt csinálja-e, amit az ügyfél szeretne. Azonban a tesztelés a verifikációs és validációs technikák széles tárházának csupán egy eleme. Ezen technikák némelyikét, mint például a programátvizsgálásokat, már csaknem harminc éve használják, de még mindig nem tartoznak a szoftverrendszerek fejlesztésének fő irányai közé.

Ebben a részben azokról a megoldásokról lesz szó, amelyek annak ellenőrzésére szolgálnak, hogy a szoftverek megfelelnek-e a specifikációnak, illetve az ügyfél igényeinek. A rész három fejezetből áll, amelyek a verifikációnak és a validációnak különböző aspektusait vizsgálják:

1. A 22. fejezet általános áttekintést nyújt a programok verifikációjáról és validációjáról. Különbséget teszek a verifikáció és validáció között, és szót ejtünk a V & V tervezési folyamatról. Ezt követően a rendszer-verifikáció statikus technikáit mutatom be. Ezekkel a technikákkal a program forráskódját ellenőrizzük, és nem tesztelést végzünk. Szó lesz a programátvizsgálásokról, az automatizált statikus elemzés hasznáról, végül pedig a formális módszerek verifikációs folyamatban betöltött szerepéről.
2. A programtesztelés a 23. fejezet témája. Írok arról, hogy hogyan végzik általában a többszintű tesztelést, és elmagyarázom a komponenstesztelés és a rendszertesztelés közötti különbséget. Egyszerű példák segítségével bemutatok számos, a programok teszteseteinek tervezéséhez használható technikát, majd a tesztelés automatizálásáról lesz szó. Az automatizált tesztelés szoftvereszközök használatával igyekszik csökkenteni a tesztelési folyamatra fordított időt és energiát.
3. A 24. fejezet egy speciálisabb témáról, a kritikus rendszerek validálásáról szól. A kritikus rendszerek esetén előfordulhat, hogy be kell bizonyítanunk az ügyfélnek vagy egy külső félnek, hogy a rendszer megfelel a specifikációjának és a vele szemben támasztott megbízhatósági követelményeknek. Bemutatom a megbízhatósági, biztonsági és védettségi felmérés különböző megközelítéseit, valamint hogy hogyan lehet a rendszer V & V folyamatainak során kapott bizonyítékokat a rendszer megbízhatóságának fejlesztésében használni.

22 VERIFIKÁCIÓ ÉS VALIDÁCIÓ

TÉMAKÖRÖK

A fejezet célja a szoftverek verifikációjának és validációjának bemutatása, különös tekintettel a statikus verifikációs technikákra. Mire a fejezet végére érünk:

- megértjük, hogy mi a különbség a szoftverek verifikációja és validációja között;
- megismerkedünk a programátvizsgálással mint a programban előforduló hibák felderítésének módszerével;
- megismerkedünk az automatizált statikus elemzés fogalmával és a verifikációban és validációban betöltött szerepével;
- megértjük, hogyan alkalmazhatjuk a statikus elemzést a Cleanroom fejlesztési folyamat során.

TARTALOM

22.1. Verifikáció- és validációtervezés
22.2. Szoftverek átvizsgálása
22.3. Automatizált statikus elemzés
22.4. Verifikáció és formális módszerek

Az implementációs folyamat közben és után az éppen fejlesztett programot ellenőrizni kell, hogy megfelel-e a specifikációjának és szolgáltatja a várt funkcionalitásokat azok felé, akik fizetnek érte. Az ellenőrző és elemző folyamatok neve *verifikáció és validáció* (V & V). A verifikációs és validációs tevékenységek a szoftverkészítés folyamatának minden lépésében jelen vannak. Követelményvizsgálattal indul, és a tervezés áttekintésén, valamint a kódvizsgálaton keresztül egészen a termék teszteléséig tart.

A verifikáció és a validáció két különböző dolog, habár gyakran összetévesztik őket. A köztük lévő különbséget tömören Boehm (Boehm, 1979) fogalmazta meg:

- „Validáció: A megfelelő terméket készítjük el?"
- „Verifikáció: A terméket jól készítjük el?"

Ezekből a definíciókból az látszik, hogy a verifikáció magában foglalja annak ellenőrzését, hogy a szoftver megfelel-e specifikációjának. Ellenőrizni kell, hogy a rendszer eleget tesz-e a megadott funkcionális és nemfunkcionális követelményeknek. Azonban a validáció ennél általánosabb folyamat. Azt kell biztosítania, hogy a szoftver megfelel a vásárló elvárásainak. Ez túlmutat annak ellenőrzésén, hogy a rendszer megfelel-e specifikációjának, egészen annak bemutatásáig, hogy a szoftver azt csinálja, amit a vásárló elvár tőle. Ahogyan arról a 2. részben már volt szó, egy szoftverrendszer specifikációja nem mindig tükrözi a rendszer felhasználóinak és tulajdonosainak valós kívánalmait és igényeit.

A verifikációs és validációs folyamat végcélja megbizonyosodni arról, hogy a szoftverrendszer „a célnak megfelel". Ez azt jelenti, hogy a rendszernek elég jónak kell lennie arra a célra, amire használni akarják. A szükséges elfogadási szint függ a rendszer céljától, a rendszer felhasználóinak elvárásaitól, valamint a rendszer aktuális piaci környezetétől:

1. *Szoftverfunkció.* A szükséges elfogadási szint attól függ, hogy a szoftver menynyire kritikus a szervezet számára. Például egy biztonságosságkritikus rendszer elfogadási szintje sokkal nagyobb, mint egy új ötletek bemutatására kifejlesztett szoftverrendszer prototípusáé.
2. *Felhasználói elvárások.* A szoftveripar egyik szomorú észrevétele, hogy sok felhasználó alacsony elvárásokat támaszt a szoftverekkel szemben, és nem lepődik meg, ha azok használat közben meghibásodnak. Hajlandók elfogadni ezeket a hibákat, ha a használattal járó haszon ellensúlyozza a hátrányokat. A felhasználó rendszerhibákkal szembeni türelme azonban az 1990-es évek óta folyamatosan csökken. Manapság kevésbé elfogadható dolog megbízhatatlan rendszereket szállítani, így a szoftvercégeknek nagyobb erőfeszítéseket kell tenniük a verifikáció és a validáció területén.
3. *Piaci környezet.* Egy rendszer piacra dobásakor eladóinak figyelembe kell venniük a versenytársakat, azt az árat, amit a vásárlók hajlandók kifizetni a rendszerért, illetve a rendszer leszállításának szükséges ütemezését. Ha egy cégnek kevés versenytársa van, akkor (mivel első szeretne lenni a piacon) elképzelhető, hogy még azelőtt piacra dobja a programot, mielőtt azt teljesen letesztelnék és „be lenne lőve". Ahol a vásárlók nem hajlandók magas árat fizetni a szoftverért, ott toleránsabbak lehetnek a szoftverhibákkal szemben. Ezeket a tényezőket mind figyelembe kell venni annak eldöntésekor, hogy mennyi erőfeszítést érdemes a V & V folyamatnak szentelni.

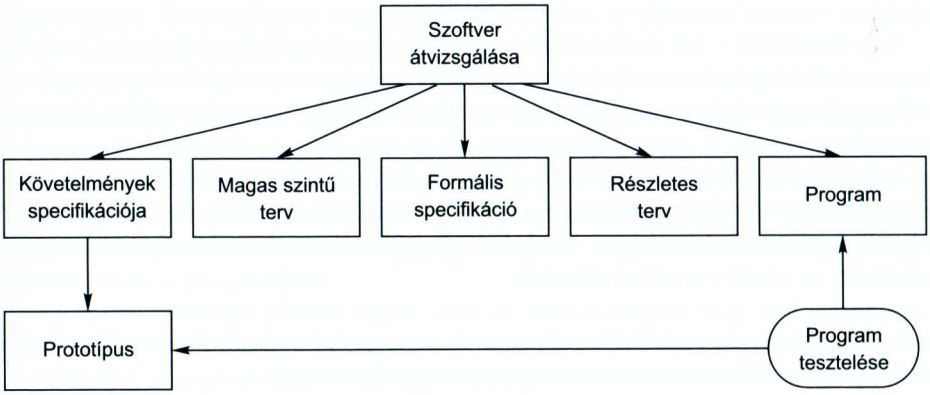
A V & V folyamaton belül a rendszer ellenőrzésére és elemzésére két, egymást kiegészítő technika használható:

1. A *szoftverátvizsgálások vagy áttekintések* a rendszer reprezentációját elemzik és ellenőrzik, például a követelményt leíró dokumentumot, a tervről készült ábrákat és a program forráskódját. Ezek a folyamat minden szintjén alkalmazhatók. A vizsgálatok kiegészíthetők a rendszer forrásszövegének vagy a hozzá kapcsolódó dokumentumoknak az automatikus elemzésével. A szoftver-

átvizsgálások és automatizált elemzések statikus V & V technikák, mert nem igénylik a program lefuttatását.

2. A *szoftvertesztelés* a szoftver implementációjának tesztadatokkal történő lefuttatásával, valamint a szoftver kimeneteinek és futás közbeni viselkedésének vizsgálatával ellenőrzi, hogy az megfelelő teljesítményt nyújt-e. A tesztelés dinamikus verifikációs és validációs technika, mivel a rendszer futtatható reprezentációjával dolgozik.

A 22.1. ábra a szoftverátvizsgálásoknak és a tesztelésnek a szoftverfolyamatban betöltött egymást kiegészítő szerepét mutatja be. A nyilak mutatják a folyamat azon szakaszait, amelyekben ezek a technikák alkalmazhatók. Mint látható, az átvizsgálások a szoftverfolyamat bármely lépésében használhatók. A követelményekkel kezdődően a szoftver minden olvasható reprezentációja átvizsgálható. Ahogyan azt már korábban említettem, a specifikáció és a terv hibáinak felderítésében a fő módszer a követelmények és terv áttekintése.



22.1. ábra. Statikus és dinamikus verifikáció és validáció

Egy rendszert csak akkor tesztelhetünk, ha már elkészült a program egy végrehajtható változatának egy prototípusa. Az inkrementális fejlesztés egyik előnye, hogy a rendszer egy tesztelhető változata már a fejlesztés folyamatának viszonylag korai szakaszában elkészül. Így az egyes funkcionalitásokat már a rendszerhez történő hozzáadáskor tesztelhetjük, ezért a tesztelés már a teljes implementáció elkészülte előtt megkezdhető.

Az átvizsgálási technikák közé tartoznak a programátvizsgálások, az automatizált forráskódelemzés és a formális verifikáció. A statikus technikák azonban csak a program és annak specifikációja közötti megfelelést (verifikáció) tudják ellenőrizni, azt nem mutatják ki, hogy a program hasznosan is működik. Nem tudják ellenőrizni a szoftver olyan fontos jellemzőit sem, mint például a teljesítmény vagy a megbízhatóság.

Habár a szoftverátvizsgálásokat manapság már széles körben használják, a fő szoftververifikációs és -validációs technika mindig is a programtesztelés marad.

A tesztelés a programot az általa feldolgozott valós adatokhoz hasonló adatokkal teszi próbára. A program kimeneteinek vizsgálatával és azokban anomáliák keresésével következtethetünk arra, hogy a programban vannak-e hibák, illetve hiányosságok. A tesztelésnek két, a szoftverfolyamat különböző szintjein használható fajtája ismeretes:

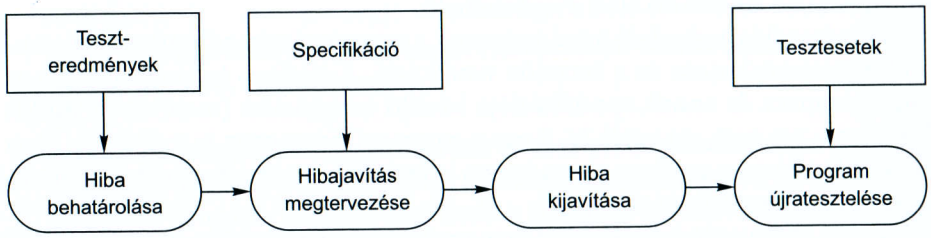
1. A *validációs tesztelés* célja megmutatni, hogy a szoftver megfelel a követelményeinek, vagyis ez az, amit a vásárló szeretne. A validációs tesztelés részeként statisztikai tesztelést is végezhetünk, amellyel a program teljesítményét és megbízhatóságát vizsgálhatjuk és ellenőrizhetjük, hogyan működik az üzemeltetési feltételek közepette. A statisztikai tesztelésről és a megbízhatóság becsléséről a 24. fejezetben lesz szó.
2. A *hiányosságtesztelés* az üzemszerű használat szimulációja helyett a rendszer hiányosságainak felfedésére szolgál, célja a program és annak specifikációja között meglévő ellentmondások felderítése. A hiányosságtesztelésről bővebben a 23. fejezetben írok.

Természetesen a két tesztelésfajta nem különül el egymástól élesen. A validációs tesztelés feltárja a rendszer bizonyos hiányosságait is, míg a hiányosságtesztelés során a tesztek némelyike megmutatja, hogy a program megfelel követelményeinek.

A V & V folyamatokat gyakran a belövési folyamattal ötvözik. Mihelyt a tesztelt programban hibákat találunk, azok kijavításának érdekében változtatnunk kell a programon. Azonban a tesztelés (vagy általánosabban a verifikáció és a validáció) és a belövés céljai eltérőek:

1. A verifikációs és validációs folyamatok arra szolgálnak, hogy megállapítsák, hogy egy szoftverrendszerben vannak-e hiányosságok.
2. A belövés az a folyamat (22.2. ábra), amely behatárolja és kijavítja ezeket a hiányosságokat.

A programok belövésére nem létezik egyszerű módszer. A jó hibakeresők mintákat keresnek a teszt kimenetében, és a hiányosság típusának, a kimeneti mintának, a programozási nyelvnek és a programozási folyamatnak az ismeretében behatárolják a hiányosságot. A belövés során felhasználhatjuk a szokásos



22.2. ábra. A belövés folyamata

programozói hibákról (például egy számláló megnövelésének elmulasztása) szóló tudásunkat, és azokat illesztjük a megfigyelt mintákra. A programozási nyelvre jellemző hibákat – például rossz mutatóhivatkozás C-ben – is figyelembe kell vennünk.

A hibák behatárolása egy programban nem mindig egyszerű folyamat, mivel a hiba nem szükségszerűen annak a pontnak a közelében van, ahol a sikertelenség bekövetkezett. A programhiba behatárolásához a belövésért felelős programozónak esetleg újabb teszteket kell megterveznie, amelyek megismétlik az eredeti hibát, és segítik a hiba forrásának felderítését. Szükség lehet a program végrehajtását szimuláló kézi nyomkövetésre is. Bizonyos esetekben a program futásáról információt gyűjtő belövőeszközök is igen hasznosak lehetnek.

Az interaktív belövőeszközök általában a fordítórendszerrel integrált nyelvi segédeszközökből álló programcsomagok részét képezik. Ezek speciális végrehajtási környezetet biztosítanak a programnak, amely lehetővé teszi a hozzáférést a fordítóprogram szimbólumtáblájához, ezáltal a program változóinak értékeihez. A felhasználók gyakran utasításról utasításra „lépegetve" vezérelhetik a végrehajtást. Az utasítások végrehajtását követően, a változók értékeit megvizsgálva, a lehetséges hibák felderíthetők.

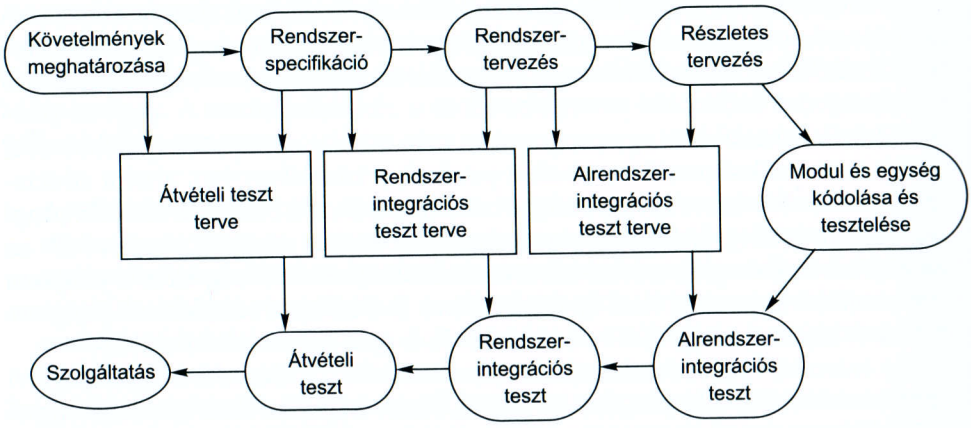
Miután sikerült hiányosságot feltárni a programban, ki kell javítani, majd a rendszert újra validálni kell. Ez magával vonhatja a program újravizsgálását, vagy az előző tesztek megismétlését (regressziós tesztelés). A regressziós tesztelést annak ellenőrzésére használják, hogy a programban bekövetkezett változtatások nem okoznak-e újabb hibákat a rendszerben. A tapasztalatok azt mutatják, hogy a „hibajavítások" viszonylag nagy hányada vagy nem teljes, vagy pedig új hibákat okoz a programban.

Elviekben minden javítási lépés után az összes tesztet meg kell ismételni, a gyakorlatban azonban ez túlságosan költséges. A teszterv részeként meg kell határozni a komponensek és az egyes komponensekhez rendelt tesztek közötti függőségeket, azaz a komponenseknek nyomon követhetőnek kell lenniük a tesztesetekből. Ha ez a nyomon követhetőség dokumentálva van, akkor a módosított komponens és a tőle függő komponensek ellenőrzéséhez elegendő a futtatást csupán a teljes tesztadatok egy részhalmazára elvégezni.

22.1. VERIFIKÁCIÓ- ÉS VALIDÁCIÓTERVEZÉS

A verifikáció és validáció költséges folyamat. Néhány rendszer, például a bonyolult, nemfunkcionális megszorításokkal rendelkező valós idejű rendszerek fejlesztési költségének több mint felét is kitehetik a V & V költségek. Ahhoz, hogy az átvizsgálásokból és a tesztelésből a lehető legtöbbet lehessen kihozni, és hogy felügyelni lehessen a verifikációs és validációs folyamat költségeit, gondos tervezésre van szükség.

Egy szoftverrendszer verifikációjának és validációjának tervezését még a fejlesztési folyamat elején el kell kezdeni. A 22.3. ábrán látható szoftverfejlesztési



22.3. ábra. Teszttervek mint fejlesztés és tesztelés közötti kapcsolatok

modellt szokás V-modellnek is nevezni (a 22.3. ábra jobb oldalára állítva egy V betűt formál). Ez az általános vízesésmodell (4. fejezet) egy példányosítása, és azt mutatja be, hogy a tesztek tervei hogyan származtathatók a rendszer specifikációjából és tervéből. A modell a verifikációs és validációs tevékenységet több szakaszra bontja, amelyek mindegyikét a program saját tervével és specifikációjával való megfelelőségének ellenőrzésére megadott tesztek vezérelnek.

A V & V tervezési folyamatnak egyensúlyt kell kialakítania a verifikáció és validáció statikus és dinamikus technikái között, a szoftverek átvizsgálásához és teszteléséhez szabványokat és eljárásokat kell megfogalmaznia, a programvizsgálatok vezérlésére ellenőrző listákat kell létrehoznia (22.3. alfejezet), és definiálnia kell a szoftver tesztterét.

Az átvizsgálásokra és a tesztelésre szánt relatív erőfeszítés a kifejlesztendő rendszer típusától és a szervezeti szaktudástól függ. Az általános szabály szerint minél kritikusabb egy rendszer, annál több erőfeszítést kell szánni a statikus verifikációs technikákra.

A teszttervezés nemcsak a terméktesztek leírására szolgál, hanem szabványok megalkotására is a tesztelési folyamat számára. Amellett, hogy segítenek a menedzsereknek az erőforrások kiosztásában és a tesztelési ütemek becslésében, a teszttervek a tervezésben és a rendszer tesztelésében részt vevő szoftverfejlesztőknek is szólnak. Lehetővé teszik a technikai személyzetnek, hogy áttekintő képet kapjanak a rendszertesztekről, és el tudják helyezni saját munkájukat ebben a környezetben. A teszttervekről és azok általánosabb minőségtervekhez fűződő kapcsolatukról Frewin és Hatton (Frewin és Hatton, 1986) ad jó leírást. Humphrey (Humphrey, 1989) és Kit (Kit, 1995) szintén ír teszttervezésről.

A nagy és komplex rendszer teszttervének fő komponenseit a 22.4. ábra mutatja be. A tesztelések ütemezésén és tevékenységein túlmenően a tesztterv definiálja a szükséges hardver- és szoftvererőforrásokat. Ez azoknak a menedzsereknek hasznos, akik felelősek azért, hogy a tesztelést végző csapat hozzáférjen

A tesztelési folyamat
A tesztelési folyamat fő fázisainak leírása. Ez az ebben a fejezetben korábban leírtak szerint végezhető el.

A követelmények nyomon követhetősége
A felhasználókat az olyan rendszerek érdeklik, amelyek megfelelnek követelményeiknek, a tesztelést pedig úgy kell megtervezni, hogy minden követelmény különállóan legyen tesztelve.

Tesztelt elemek
Meg kell adni a szoftverfolyamat azon termékeit, amelyeket tesztelni kell.

A tesztelés ütemezése
Szükség van átfogó tervezési ütemtervre és az ütemtervnek megfelelő erőforrás-foglalásra. Ez – nyilvánvalóan – kapcsolódik az általánosabb projektfejlesztési ütemezéshez.

Tesztdokumentáló eljárások
Nem elég a teszteket csupán lefuttatni. A tesztek eredményeit szisztematikusan fel kell jegyezni. Lehetővé kell tenni a tesztelési folyamat átvizsgálását annak ellenőrzésére, hogy helyesen hajtották-e végre.

Hardver- és szoftverkövetelmények
Ez a szakasz felsorolja a szükséges szoftvereszközöket és a becsült hardverhasználatot.

Megszorítások
A tesztelési folyamatot befolyásoló megszorításokat (például munkaerő-leépítés) kell ebben a szakaszban figyelembe venni.

22.4. ábra. Egy szoftver tesztterének szerkezete

ezekhez az erőforrásokhoz. A teszterveknek számos eshetőséget kell figyelembe vennie azért, hogy kalkulálni lehessen a tervezésben és az implementációban előforduló csúszásokat, és a tesztelésre kiválasztott személyeket be lehessen vetni más tevékenységekre.

Kisebb rendszerek esetében kevésbé formális tesztterv is használhatunk, de a tesztelési folyamat tervezéséhez ekkor is szükség van formális dokumentumra. Bizonyos agilis folyamatok esetében, mint például az extrém programozás, a tesztelés elválaszthatatlan a fejlesztéstől. Az egyéb tervezési tevékenységekhez hasonlóan a tesztterv is inkrementális. Az XP-ben az ügyfél felelőssége eldönteni, hogy mennyi erőfeszítést kell a rendszer tesztelésére szánni.

A tesztterv nem statikus dokumentumok, hanem a fejlesztési folyamat során fejlődnek, a fejlesztési folyamat más lépéseiben bekövetkező késések miatt változnak. Ha egy rendszer egy része még nincs teljesen készen, egészében nem tesztelhető. A tervet éppen ezért módosítani kell, hogy a tesztelőket valamilyen más tevékenységre lehessen bevetni, míg a szoftver újra tesztelhetővé nem válik.

22.2. SZOFTVEREK ÁTVIZSGÁLÁSA

A szoftverek átvizsgálása statikus V & V folyamat, amelynek során áttekintjük a szoftverrendszert hibák, kimaradt részek és anomáliák után kutatva. Általában az átvizsgálások középpontjában a forráskód áll, de a szoftver bármilyen olvasható reprezentációja (például követelmények, terv) átvizsgálható. A rendszer átvizsgálása során a hibák felderítését a rendszerről, a szakterületről, a programozási nyelvről és a tervről szóló ismereteink segítségével végezzük.

A teszteléssel szemben az átvizsgálásoknak három nagy előnyük van:

1. Tesztelés során bizonyos hibák elfedhetnek más hibákat. Ha egy hibát felfedezünk, sosem lehetünk biztosak abban, hogy a további kimeneti rendellenességek egy új hibának, vagy az eredeti hiba mellékhatásainak köszönhetők. Mivel az átvizsgálás statikus folyamat, nem kell foglalkoznunk a hibák közötti kölcsönhatásokkal, tehát egy átvizsgálási menet során sok hibát is felderíthetünk.
2. A hiányos verziók is pluszköltségek nélkül átvizsgálhatók. Ha a program még nincs teljesen készen, a rendelkezésre álló részek tesztelése érdekében speciális tesztkörnyezetet kell kifejleszteni, ami nyilvánvalóan növeli a rendszerfejlesztés költségeit.
3. A programhibák utáni kutakodás mellett az átvizsgálás olyan szélesebb körű minőségi jellemzőket is figyelembe vehet, mint a szabványoknak történő megfelelés, a hordozhatóság és a karbantarthatóság. Kereshetünk a rendszer karbantartását és frissítését megnehezítő, nem hatékony kódrészleteket, nem megfelelő algoritmusokat és rossz programozási stílusjegyeket is.

Az átvizsgálások ötlete igen régi. Számos tanulmány és kísérlet igazolja, hogy a hibák felderítésére az átvizsgálások hatékonyabbak a programtesztelésnél. Fagan (Fagan, 1986) azt állította, hogy egy program hibáinak több mint 60 százaléka felderíthető informális programátvizsgálásokkal. Mills és mások (Mills *és mások*, 1987) szerint a helyességi állításokon alapuló átvizsgálások egy formálisabb megközelítésével egy programban a hibák több mint 90 százaléka felderíthető. Ezt a technikát alkalmazza a 22.4. alfejezetben bemutatott Cleanroom-folyamat. Selby és Basili (Selby *és mások*, 1987) tapasztalati úton összehasonlították az átvizsgálások és a tesztelés hatékonyságát. Arra jutottak, hogy a statikus kódátvizsgálás a hiányosságtesztelésnél hatékonyabb és kevésbé költséges programhiba-felderítő módszer. Gilb és Graham (Gilb és Graham, 1993) szintén erre az eredményre jutottak.

Az átvizsgálás és a tesztelés nem egymás versenytársai. Mindkettőnek megvannak az előnyei és a hátrányai, és a verifikációs és validációs folyamatban együtt kell alkalmazni. Gilb és Graham azt mondják, hogy az átvizsgálás egyik leghatékonyabb módja a rendszer teszteseteinek átvizsgálása. Az átvizsgálás képes felderíteni a tesztekkel kapcsolatos problémákat, és segítséget tud nyújtani a rendszertesztelés sokkal hatékonyabb módszereinek megtervezésében. A rendszer verifikációját és validációját a fejlesztési folyamat elején kezdhetjük átvizs-

gálással, de a rendszer integrálása után teszteléssel kell ellenőrizni a tulajdonságokat, és megbizonyosodni arról, hogy a rendszer funkcionalitása megegyezik azzal, amit a tulajdonosa akart.

Az átvizsgálás eredményességének dacára bebizonyosodott, hogy sok szoftverfejlesztő szervezetnél nehéz a formális átvizsgálást bevezetni. A programtesztelésben jártas szoftvertervezők vonakodva fogadják el, hogy ezek a technikák a tesztelésnél hatékonyabb hibafelismerést tesznek lehetővé. A menedzserek gyanakvóan tekintenek ezekre a módszerekre, mivel a tervezés és a fejlesztés során további költségeket igényelnek, és nem szeretnék vállalni annak kockázatát, hogy a programtesztelés során nem derül fény megfelelő megtakarításokra.

Az átvizsgálások kétségkívül „előretolják" a szoftver V & V költségeit, és csak a fejlesztőcsapatok tapasztalttá válásával eredményeznek költségmegtakarítást. Ezen túlmenően, az átvizsgálások bevezetése gyakorlati problémákat is okoz, hiszen elvégzésük időbe telik, és ezért a fejlesztési folyamat lelassul. Egy nyomás alatt lévő menedzsert pedig igen nehéz meggyőzni arról, hogy ez az idő később, a program belövésekor visszatérül.

22.2.1. A programátvizsgálási folyamat

A programátvizsgálások célja a program hiányosságainak felderítése. A formalizált átvizsgáló folyamat elképzeléseit elsőként az IBM dolgozta ki az 1970-es években (Fagan, 1976; 1986). Mára ez a programverifikáció széles körben elterjedt módszerévé vált, különösen a kritikus rendszerek esetében. Fagan eredeti módszeréből kiindulva számos alternatív átvizsgálási megközelítést dolgoztak ki (Gilb és Graham, 1993), azonban ezek mindegyike Fagan eredeti elképzelésein alapul, amelyek szerint egy különböző háttérrel rendelkező tagokból álló csoportnak a program forráskódját gondosan, sorról sorra át kell néznie.

A programátvizsgálás és a minőségvizsgálat egyéb fajtái közötti fő különbség az, hogy az átvizsgálás alapvető célja a hiányosság felderítése, nem pedig átfogóbb tervezési kérdések figyelembevétele. A hiányosságok lehetnek akár logikai hibák, a kódban található anomáliák, amelyek egy hibás feltételt jelentenek, vagy akár a meg nem felelés a szervezeti és projektszabványoknak. Ezzel szemben a vizsgálatok egyéb fajtái az ütemezéssel, a költségekkel, a meghatározott mérföldköveknek megfelelő haladással, illetve annak megállapításával foglalkozhatnak, hogy a szoftver valószínűleg megfelel-e a szervezeti céloknak.

Az átvizsgálás legalább négy emberből álló kis csapat által végzett formális folyamat. A csapat tagjai szisztematikusan elemzik a kódot, és rámutatnak az esetleges hiányosságokra. Fagan eredetileg a szerző, olvasó, tesztelő és moderátor szerepkörök bevezetését javasolta. Az olvasó hangosan felolvassa a kódot az átvizsgálást végző csapatnak; a tesztelő a kódot tesztelési szemszögből vizsgálja; míg a moderátor a folyamat szervezését végzi.

A szervezetek átvizsgálással kapcsolatos tapasztalatainak felhalmozásával együtt újabb szerepkörök bukkantak fel. Az átvizsgálásnak a Hewlett–Packard fejlesztési folyamatába való sikeres bevezetését taglaló leírásában Grady és Van

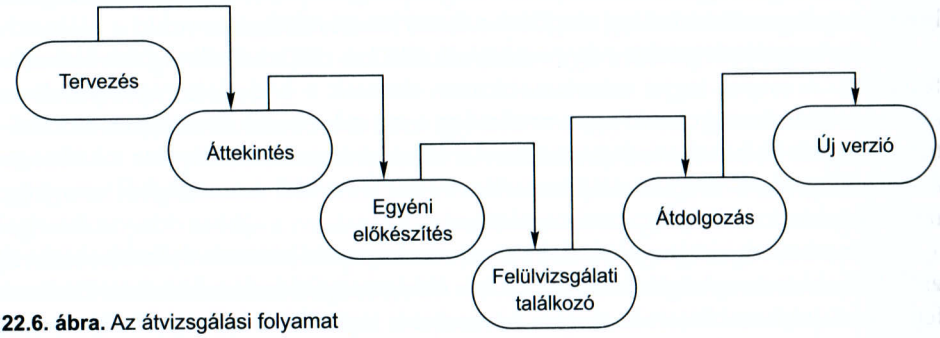
Slack (Grady és Van Slack, 1994) a 22.5. ábrán látható hat szerepkört javasolják. Szerintük a program hangos felolvasása szükségtelen. A különböző szerepkörök ugyanazon személyhez is rendelhetők, így a csapat mérete átvizsgálásról átvizsgálásra változhat. Gilb és Graham szerint az átvizsgálókat úgy kell kiválasztani, hogy különféle nézőpontokat (például tesztelő, végfelhasználó, minőségkezelő stb.) tükrözzenek.

Szerepkör	Leírás
Szerző vagy tulajdonos	A program vagy dokumentum elkészítéséért felelős programozó vagy tervező. Ő felelős az átvizsgálási folyamat során felfedezett hiányosságok kijavításáért.
Átvizsgáló	Megtalálja a programok, illetve a dokumentumok hibáit, mulasztásait és inkonzisztenciáit. Az átvizsgálást végző csapat hatáskörén kívül elhelyezkedő átfogóbb kérdéseket is felfedhet.
Olvasó	Egy átvizsgálási találkozón elmagyarázza a kódot, illetve dokumentumot.
Írnok	Feljegyzi az átvizsgálási találkozók eredményeit.
Elnök vagy moderátor	Kezeli a folyamatot, és megkönnyíti az átvizsgálást. A folyamat eredményeit jelenti a legfőbb moderátornak.
Legfőbb moderátor	Az átvizsgálási folyamat tökéletesítéséért, az ellenőrző lista frissítéséért, a szabványok fejlődéséért stb. felelős.

22.5. ábra. Szerepkörök az átvizsgálási folyamatban

Az átvizsgálási folyamat tevékenységei a 22.6. ábrán láthatók. Mielőtt a program átvizsgálása kezdetét venné, lényeges a következők megléte:

1. Az átvizsgálandó kódnak legyen pontos specifikációja. Teljes specifikáció nélkül lehetetlen egy komponenst olyan részletezettségi szinten átvizsgálni, amely a hiányosságok felderítéséhez szükséges.
2. Az átvizsgáló csapat tagjai jártasak legyenek a szervezeti szabványokban.
3. Hozzá lehessen férni a kódnak egy aktuális, szintaktikusan helyes változatához. „Majdnem teljes" kód átvizsgálásának nincs értelme, még akkor sem, ha a késlekedés felborítja az ütemtervet.



22.6. ábra. Az átvizsgálási folyamat

A moderátor az átvizsgálás megtervezéséért felelős. Ez magában foglalja a csapat kiválasztását, a találkozó helyének megszervezését, és annak biztosítását, hogy az átvizsgálandó anyag és specifikációi teljesek. Az átvizsgálandó programot az áttekintő szakaszban mutatják meg az átvizsgálást végző csapatnak, amikor is a program írója elmagyarázza, hogy az mit is csinál. Ezt egyéni előkészület követi. Minden csapattag tanulmányozza a specifikációt és a programot, és hiányosságokat keres a kódban.

Maga az átvizsgálás viszonylag rövid ideig tart (két óránál nem tovább), és a hiányosságok, anomáliák és szabványoknak való meg nem felelések felfedezésére összpontosít. Az átvizsgálást végző csapat nem tesz javaslatot arra, hogyan kellene ezeket a hiányosságokat pótolni, és az egyéb komponensekben sem javasol változtatásokat.

Az átvizsgálást követően szerzője módosítja a programot, a feltárt problémák kijavítása céljából. Az ezt követő szakaszban a moderátornak döntenie kell, hogy szükséges-e a kód újravizsgálása. Dönthet úgy is, hogy nincs szükség teljes újravizsgálásra, azaz a hiányosságokat sikeresen kiküszöbölték. A moderátor ezek után kibocsátásra jóváhagyja a programot.

Az átvizsgálás során gyakran felhasználásra kerül a szokásos programozási hibák ellenőrző listája. Ez az ellenőrző lista könyvek hasonló példái és az adott szakterület szokásos hibáinak ismerete alapján építhető fel. Mivel minden nyelvnek megvannak a saját jellemző hibái, különböző programozási nyelvekhez különböző ellenőrző listát kell készíteni. Az átvizsgálásokról szóló átfogó fejtegetéseiben Humphrey (Humphrey, 1989) számos példát ad ellenőrző listákra.

Ezek az ellenőrző listák azért a programozási nyelvtől függenek, mert a különböző nyelvek fordítóprogramjai különböző szintű ellenőrzéseket biztosítanak. Például egy Ada-fordító ellenőrzi, hogy egy függvényt megfelelő számú paraméterrel hívunk-e meg, míg egy C-fordító nem. A 22.7. ábrán néhány, az átvizsgálási folyamat során végrehajtható lehetséges ellenőrzés látható. Gilb és Graham (Gilb és Graham, 1993) hangsúlyozzák, hogy minden szervezetnek ki kell dolgoznia a helyi szabványokon és gyakorlaton alapuló saját ellenőrző listáját, amelyet új hiányosságfajták felfedezése esetén frissítenie kell.

Egy átvizsgáláshoz szükséges idő és az ez idő alatt átvizsgálható kód mennyisége függ az átvizsgálást végző csapat tapasztaltságától, a programozási nyelvtől és az alkalmazás szakterületétől. Fagan az IBM-nél és Barnard és Price (Barnard és Price, 1994) egyaránt hasonló következtetésre jutott, amikor telekommunikációs szoftverek átvizsgálási folyamatát tanulmányozták:

1. Az áttekintő szakasz során óránként körülbelül 500 forráskódú utasítás vizsgálható át.
2. Az egyéni előkészület során körülbelül 125 forráskódú utasítást lehet megvizsgálni óránként.
3. A felülvizsgálati találkozó során óránként 90–125 utasítás vizsgálható át.

Egy négy tagból álló átvizsgáló csapat esetén 100 kódsor átvizsgálásának költsége nagyjából egy embernapnyi ráfordítással egyezik meg. Ez azt feltételezi,

A hiba típusa	Átvizsgálási ellenőrzés
Adathibák	Minden változó kapott-e értéket, mielőtt értéke felhasználásra kerül? Minden konstans meg van-e nevezve? A tömbök felső határa a tömb méretével, vagy (méret – 1)-gyel egyezik-e meg? Sztringek használata esetén, explicit módon ki van-e téve az elhatárolójel? Elképzelhető-e valamilyen puffertúlcsordulás?
Vezérlési hibák	Minden feltételes utasításban helyes-e a feltétel? Minden ciklus biztosan véget ér-e? Az összetett utasítások megfelelően vannak-e zárójelezve? Elágaztató utasítás esetén minden lehetséges esetet figyelembe vettek-e? Ha az elágaztató utasításban az egyes esetek után breakre van szükség, akkor az nem hiányzik-e valahonnan?
Input/output hibák	Minden inputváltozót felhasználtak-e? Minden outputváltozóhoz rendeltek-e értéket? A váratlan inputok okozhatnak-e adatmeghibásodást?
Interfészhibák	Minden függvény- és módszerhívás megfelelő paraméterszámmal rendelkezik? A formális és aktuális paraméterek típusai megegyeznek-e? Megfelelő-e a paraméterek sorrendje? Ha a komponensek egy megosztott memóriaterülethez férnek hozzá, ugyanazzal az osztott memóriaszerkezet-modellel dolgoznak-e?
Tárkezelési hibák	Ha egy láncszerkezet módosult, minden mutató helyesen van-e beállítva? Dinamikus memóriakezelés esetén megtörtént-e a megfelelő mennyiségű helyfoglalás? Explicit módon felszabadították-e a már nem szükséges tárrészeket?
Kivételkezelési hibák	Minden lehetséges hibafeltételt figyelembe vettek-e?

22.7. ábra. Átvizsgálási ellenőrzések

hogy maga az átvizsgálás körülbelül egy órát vesz igénybe, és minden csapattag 1-2 órát tölt az átvizsgálás előkészületeivel. A tesztelési költségek igencsak változók, és a programban található hibák számától függnek. A programátvizsgáláshoz szükséges erőfeszítések azonban talán kevesebb mint a felét teszik ki annak, amennyire egy azzal ekvivalens hiányosságtesztéléshez szükség volna.

Néhány szervezet (Gilb és Graham, 1993) felhagyott a komponenstesztéléssel, és helyette átvizsgálást alkalmaz. Úgy találták, hogy a programátvizsgálás olyan hatékony a hibák megtalálásában, hogy a komponenstesztelés költségei nem indokolhatók. Ezen szervezetek szerint a komponensátvizsgálások rendszertesztéléssel kombinálva biztosítják a leginkább költséghatékony V & V stratégiát. Ahogyan azt a fejezetben később tárgyaljuk, a Cleanroom szoftverfejlesztési folyamat is ezt az elvet vallja.

Az átvizsgálások bevezetése következményekkel jár a projektmenedzsmentre nézve, és nagyon fontos, hogy a szoftverfejlesztő csapatok elfogadják az átvizsgálást. A programátvizsgálások – szemben a bizalmasabb komponenstesztelési folyamattal – nyilvános hibakeresési folyamatok, ahol egy ember hibája a teljes programozócsapat szeme elé kerül. Az átvizsgálást végző csapatok vezetőit gon-

dosan fel kell készíteni a folyamat menedzselésére, és olyan viselkedési kultúra kialakítására, amelyben a segítség a hibák felderítésekor érkezik, és ezekhez a hibákhoz semmiféle szemrehányás nem kapcsolódik.

Ahogy egy szervezet egyre több tapasztalatot gyűjt az átvizsgálási folyamatról, annak eredményei a folyamat tökéletesítésének eszközeként használhatók fel. Az átvizsgálások a bekövetkező hibák fajtáira vonatkozó adatgyűjtés számára ideális megoldást nyújtanak. Az átvizsgálást végző csapat és az átvizsgált kód szerzői okokat kereshetnek, hogy miért következhettek be ezek a hibák. Ahol lehetséges, ott módosítani kell a folyamatot, hogy a hibák okait kiküszöböljék, és azok a jövőben ne ismétlődhessenek meg.

22.3. AUTOMATIZÁLT STATIKUS ELEMZÉS

Az átvizsgálás a statikus elemzés egyik formája, amikor a programot anélkül vizsgáljuk, hogy végrehajtanánk. Ahogyan korábban már említettem, az átvizsgálás gyakran ellenőrző listákon és a különböző programozási nyelvekben elkövetett tipikus hibák azonosítására szolgáló heurisztikákon alapszik. Bizonyos hibák és heurisztikák esetén a programok lista alapján történő ellenőrzése automatizálható. Ennek eredményeként a különböző programozási nyelvek számára automatikus statikus elemzőket fejlesztettek ki.

A statikus elemzők olyan szoftvereszközök, amelyek a program forrásszövegének vizsgálatával derítik fel a lehetséges hibákat és anomáliákat. A programszöveg elemzésével ismerik fel a programban található különféle utasításokat. Észlelhetik, hogy az utasítások formailag helyesek-e, következtetéseket tehetnek a vezérlés menetére vonatkozóan, és sok esetben kiszámíthatják a program adatainak összes lehetséges értékét, valamint kiegészítik a nyelv fordítóprogramja által biztosított hibafelismerő lehetőségeket. Ezen eszközöket az átvizsgálási folyamat részeként és különálló V & V tevékenységként egyaránt alkalmazni lehet.

Az automatikus statikus elemzés célja, hogy felhívja az átvizsgáló figyelmét a programban található anomáliákra, például az inicializálás nélkül használt változókra, az egyáltalán nem használt változókra, olyan adatokra, amelyek értéke kifuthat tartományából stb. A 22.8. ábrán néhány statikus elemzéssel felismerhető ellenőrzés látható. Az anomáliák gyakran programozási hibák vagy mulasztások eredményei, vagyis olyan dolgokat hangsúlyozhatnak, amelyek a program végrehajtása során problémát okozhatnak Ezek az anomáliák azonban nem szükségképpen programhibák, lehetnek szándékosak vagy következmény nélküliek is.

A statikus elemzés szakaszai a következők:

1. *A vezérlés folyamatának elemzése.* Ez a szakasz a többszörös be-, illetve kilépési ponttal rendelkező ciklusokat és elérhetetlen kódrészleteket ismer fel és hangsúlyoz ki. Az elérhetetlen kódrészlet olyan kód, amely feltétel nélküli ugró utasításokkal van körülvéve, vagy egy feltételes utasítás azon ágában van, amelynek őrfeltétele soha nem lehet igaz.

2. *Az adathasználat elemzése.* Ez a szakasz azt emeli ki, hogy a programban hogyan használják a változókat. Megtalálja az inicializálás nélkül használt változókat, azokat, amelyek kétszer kerülnek outputra anélkül, hogy közben új értéket kapnának, valamint a deklarált, de sehol fel nem használt változókat. Az adathasználat elemzése felismeri azokat a nem hatékony feltételvizsgálatokat is, ahol a tesztfeltétel redundáns. A redundáns feltételek értéke sosem változik meg – vagy mindig igazak, vagy mindig hamisak.
3. *Interfészelemzés.* Ez az elemzés a rutinok és eljárások deklarációjának konzisztenciáját és azok használatát ellenőrzi. Ez szükségtelen abban az esetben, ha az implementációt egy erősen típusos nyelven (például Javában) végezzük, ahol a fordítóprogram elvégzi ezeket az ellenőrzéseket. Az interfészelemzés a gyengén típusos nyelvekben fedezhet fel típusokkal kapcsolatos hibákat, mint például a FORTRAN vagy a C. Az interfészelemzés fényt deríthet a deklarált, ám meg nem hívott függvényekre és eljárásokra, valamint függvények fel nem használt visszatérési értékére is.
4. *Az információáramlás elemzése.* Az elemzés e fázisa a bemenő és kimenő változók közötti függőségeket ismeri fel. Noha nem ismeri fel az anomáliákat, megmutatja, hogy a program változóinak értéke hogyan származnak más változók értékeiből. Ennek az információnak a birtokában a kódátvizsgálás képes megtalálni a rosszul kiszámított értékeket. Az információáramlás elemzése képes megmutatni az egy változó értékét befolyásoló feltételeket is.
5. *Útvonalelemzés.* A szemantikus elemzés e szakasza azonosítja a program összes lehetséges végrehajtási útvonalát, és felsorolja az ezeken az útvonalakon végrehajtott utasításokat. Lényegében felgöngyölíti a program végrehajtását, és lehetővé teszi az összes lehetséges predikátum önálló módon történő elemzését.

A hiba típusa	Statikus elemzési ellenőrzés
Adathibák	Inicializálás előtt használt változók. Deklarált, de nem használt változók. Változók, amelyek kétszer kaptak értéket, de a két értékadás között értékükre nem történt hivatkozás. Lehetséges tömbhatártúllépések. Deklarálatlan változók.
Vezérlési hibák	Elérhetetlen kód. Feltétel nélküli ugrás ciklusokba.
Input/output hibák	Változók, amelyek kétszer kerülnek outputra, közbülső értékadás nélkül.
Interfészhibák	Paraméterek típusütközése. Paraméterszám-eltérés. Függvények visszatérési értékeinek fel nem használása. Meghívatlan függvények és eljárások.
Tárkezelési hibák	Érték nélküli mutatók. Mutatóaritmetika.

22.8. ábra. Automatizált statikus elemzési ellenőrzések

A statikus elemzők különösen értékesek akkor, ha egy C-hez hasonló programozási nyelvet használunk. A C-nek nincsenek a típusokra vonatkozó szigorú szabályai, és a C-fordító által végezhető ellenőrzés is korlátozott. Éppen ezért jó sok olyan programozói hibalehetőség van, amely az elemzőeszköz segítségével automatikusan felderíthető. Ez különösen fontos akkor, amikor a C-t (vagy kisebb mértékig a C++-t) kritikus rendszerek fejlesztésére használjuk. Ebben az esetben a statikus elemzés nagyszámú potenciális hiba felderítésére alkalmas, és ezzel lényegesen csökkentheti a tesztelés költségeit.

Kétség sem férhet ahhoz, hogy a C-hez hasonló nyelvek esetében a statikus elemzés a programhibák felderítésének hatékony eszköze. Ez kompenzálja a programozási nyelv tervezési gyengeségeit. A modern programozási nyelvekből azonban, mint amilyen a Java is, a nyelv tervezői számos hibaforrást rejtő nyelvi tulajdonságot eltávolítottak. Minden változót inicializálni kell, nincs goto utasítás, így kevésbé valószínű, hogy véletlenül elérhetetlen kódot írunk, a tárkezelés pedig automatikus. Ez a hibaelkerülő megközelítés a hibafelismerésnél is hatékonyabb a program megbízhatóságának fejlesztésében. Habár léteznek Javához készült statikus elemzők, ezek nem terjedtek el széles körben. Nem teljesen világos, hogy a felderített hibák száma arányban áll-e a kimenetük elemzéséhez szükséges idővel.

Éppen ezért, a statikus elemzés bemutatására inkább egy kis C-programot választottam, mintsem egy Java-programot. A Unix- és Linux-rendszerek tartalmaznak egy C-programok számára készült LINT nevű statikus elemzőt. A LINT olyan statikus ellenőrzéseket végez, amely megfelel az erősen típusos nyelvek fordítóprogramja által nyújtottaknak, mint amilyen például a Java. A 22.9. ábrán

```
138% more lint_ex.c

#include <stdio.h>
printarray (Anarray)
   int Anarray;
{
   printf("%d",Anarray);
}
main ()
{
   int Anarray[5]; int i; char c;
   printarray (Anarray, i, c);
   printarray (Anarray);
}
139% cc lint_ex.c
140% lint lint_ex.c
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

22.9. ábra. Statikus elemzés LINT segítségével

a LINT egy kimenetére láthatunk példát. Az első parancs (138-as sor) a programot listázza ki, amely egy printarray nevű egyparaméteres függvényt definiál, majd meghívja ezt a függvényt három paraméterrel. Az i és c változók deklarálva vannak ugyan, de soha nem kapnak értéket. A függvény által visszaadott értéket sem használjuk fel.

A 139-es sorszámú sor azt mutatja meg, hogy a programot a C-fordító hibamentesen lefordítja. Ezt a LINT statikus elemző lefuttatása követi, amely észleli és jelenti a program hibáit.

A statikus elemző azt mutatja ki, hogy a c és i skalárváltozókat használja, de nincsenek inicializálva, valamint, hogy a printarray deklarációjától eltérő számú paraméterrel lett meghíva. Ezenfelül észreveszi a printarray első argumentumának inkonzisztens használatát és azt is, hogy a függvény értéke nem kerül felhasználásra.

Az eszközalapú elemzés nem helyettesítheti az átvizsgálásokat, mivel az elemzők bizonyosfajta hibákat nem képesek felismerni. Például képesek felismerni az inicializálatlan változókat, de nem képesek felismerni a helytelen kezdeti értékadásokat. A gyengén típusos nyelvekben, mint amilyen a C is, a statikus elemzők felismerik a rossz argumentumszámmal és -típussal rendelkező függvényeket, de nem tudják kezelni azt a helyzetet, amikor egy helyes típusú, de rossz argumentumot adunk át egy függvénynek.

A fenti problémák megoldása érdekében az olyan statikus elemzők, mint amilyen például az LCLint (Orcero, 2000; Evans és Larochelle, 2002), támogatják az annotációk használatát, amikor is a felhasználók stilizált megjegyzések formájában különböző megszorításokat definiálhatnak. Ezek a megszorítások lehetővé teszik a programozó számára, hogy például egy függvény változói nem változtathatók meg, mely globális változók használhatók stb. A statikus elemző ellenőrizni tudja, hogy a program eleget tesz-e a megszorításoknak, és kiemeli a hibásnak tűnő kódrészleteket.

22.4. VERIFIKÁCIÓ ÉS FORMÁLIS MÓDSZEREK

A szoftverfejlesztés formális módszerei a szoftver matematikai reprezentációján (ez általában formális specifikáció) alapulnak. Ezek a formális módszerek elsősorban a specifikáció matematikai elemzésével foglalkoznak, a specifikációt egy részletesebb, de ekvivalens szemantikájú reprezentációba transzformálja, és formális eszközökkel ellenőrzi, hogy a rendszer egyik reprezentációjának szemantikája megegyezik-e egy másik reprezentációéval.

A formális módszerek használatára mint alapvető statikus verifikációs technikára tekinthetünk. Használatukhoz a rendszer-specifikációnak és a programnak nagyon részletes elemzése szükséges, ami gyakran időrabló és költséges. Éppen ezért formális módszereket leggyakrabban csak biztonságosság- és védettségkritikus szoftverfejlesztési folyamatokban használnak. Az Egyesült Ki-

rálység védelmi szabványa előírja a formális matematikai specifikáció és a hozzá kapcsolódó verifikáció használatát biztonságosságkritikus szoftver esetén (MOD, 1995).

A formális módszerek a V & V folyamat különböző lépéseiben használhatók:

1. Kifejleszthetjük a rendszer formális specifikációját, és matematikailag elemezhetjük azt, inkonzisztenciát keresve. Ez a technika hatékonyan támogatja a specifikációs hibák és hiányosságok felderítését, amint azt a 10. fejezetben tárgyaltam.
2. Matematikai állítások segítségével formálisan ellenőrizhetjük, hogy a szoftverrendszer kódja összhangban van-e specifikációjával. Ehhez egy formális specifikációra van szükség, és a programozási és bizonyos tervezési hibák felderítésében hatékony. A formális verifikációs folyamat támogatására egyaránt használhatunk egy transzformációs fejlesztési folyamatot, amikor egy formális specifikációnak egy részletesebb reprezentációkat tartalmazó transzformáció-sorozatát tekintjük, és egy Cleanroom-folyamatot.

A formális specifikáció és a hozzá kapcsolódó programverifikáció használata mellett az szól, hogy a formális specifikáció a specifikáció részletes elemzését kényszeríti ki. Feltárhat esetlegesen előforduló ellentmondásokat vagy hiányosságokat, amelyeket egyébként valószínűleg nem fedeznénk fel a rendszer működésbe lépése előtt. A formális verifikáció megmutatja, hogy a fejlesztett program megfelel a specifikációnak, és implementációs hibák nem veszélyeztetik az üzembiztonságot.

A formális specifikáció használata ellen az szól, hogy speciális jelöléseket igényel. Ezeket csak speciálisan képzett dolgozók tudják használni, és a szakterületi szakértők sem képesek megérteni azokat. Ezáltal a rendszerkövetelményekkel kapcsolatos problémákat elrejtheti a formalitás. A szoftvertervezők nem tudják felismerni a követelményekben lévő esetleges hibákat, mivel nem értik a szakterületet; a szakterületi szakértők szintén nem találják meg ezeket a problémákat, mivel nem értik a specifikációt. Bár a specifikáció matematikailag konzisztens lehet, előfordulhat, hogy nem ír le olyan rendszertulajdonságokat, amik valóban szükségesek lennének.

Egy nemtriviális szoftverrendszer verifikálása rengeteg időt vesz igénybe, és olyan speciális eszközöket igényel, mint a tételbizonyítók. Ezért ez különösen költséges folyamat és a költségek nemlineárisak. A rendszer méretének növekedésekor a formális verifikálás költsége aránytalanul megnő. Sokan ezért azt gondolják, hogy a formális verifikáció nem költséghatékony. Ugyanolyan szintű biztonságosság vagy védettség érhető el alacsonyabb költségen, más validálási technikák használatával, például az átvizsgálással vagy a rendszer tesztelésével.

Néha azt állítják, hogy a rendszerfejlesztésben a formális módszerek használata megbízhatóbb és biztonságosabb rendszerekhez vezet. Nincs kétség afelől, hogy egy formális rendszer-specifikációban kisebb valószínűséggel szerepelnek olyan anomáliák, amiket a rendszertervezőnek kell feloldania. Ugyanakkor, a

formális specifikáció és bizonyítás nem garantálja, hogy a szoftver a gyakorlati használat során megbízhatóan fog működni. Ennek okai a következők:

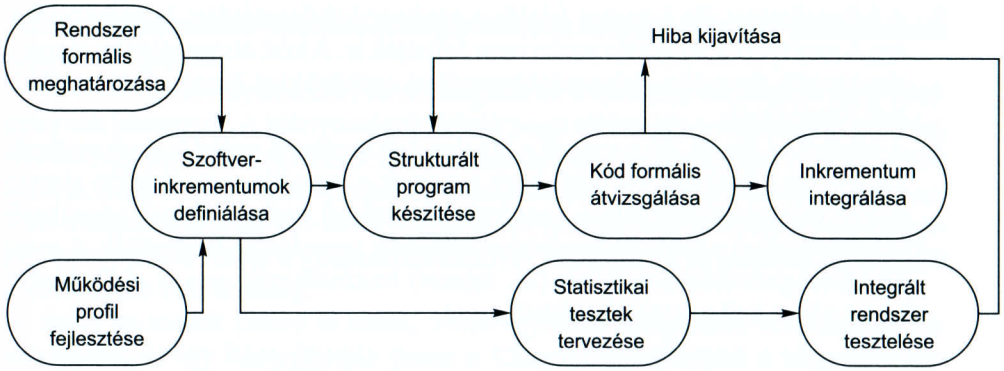
1. *Nem biztos, hogy a specifikáció a rendszer felhasználóinak valós követelményeit tükrözi.* Lutz (Lutz, 1993) felfedezte, hogy a felhasználók által tapasztalt számos hiba olyan specifikációbeli hiba vagy hiányosság okán állt elő, amelyet nem lehetett felfedezni formális rendszer-specifikációval. Továbbá a felhasználók ritkán értik meg a formális jelöléseket, így nem tudják közvetlenül elolvasni a formális specifikációt, hogy megtalálják a hibákat és hiányosságokat.
2. *A bizonyítás tartalmazhat hibákat.* A programbizonyítások terjedelmesek és összetettek, így akár a nagy és összetett programok rendszerint hibákat is tartalmaznak.
3. *A bizonyítás megengedhet helytelen használati módot.* Ha a rendszert nem a várt módon használják, a bizonyítás érvényét vesztheti.

Hátrányai ellenére, véleményem szerint, a formális módszereknek fontos szerepet kell játszaniuk a biztonságosságkritikus és védettségkritikus rendszerek fejlesztésében. A formális specifikációk nagyon hatékonyak a rendszerhibák legáltalánosabb okait képező specifikációs problémák felderítésében. A formális verifikáció növeli a bizalmat e rendszerek legkritikusabb komponensei iránt. A formális módszerek használata egyre jobban terjed, ahogy a megrendelők igényt tartanak rájuk, és ahogy egyre több tervező ismeri meg alaposabban ezeket a technikákat.

22.4.1. Cleanroom-szoftverfejlesztés

A formális módszereket számos szoftverfejlesztési folyamattal integrálták. A B módszertanban (Wordsworth, 1996) egy formális specifikációt helyességmegőrző transzformációk sorozatán keresztül kell programmá alakítani. Az SDL-t (Mitschele-Thiel, 2001) telekommunikációs rendszerek fejlesztéséhez használják, a VDM-et (Jones, 1986) és a Z-t (Spivey, 1992) pedig vízesés-típusú folyamatok esetén. Formális módszereket használó, szintén jól dokumentált megoldás a Cleanroom szoftverfejlesztési folyamat. A Cleanroom-szoftverfejlesztés (Mills *és mások*, 1987; Cobb és Mills, 1990; Linger, 1994; Prowel *és mások*, 1999) olyan elképzelés, amely a szoftverhibák szigorú átvizsgálási folyamat használatával történő elkerülésén alapszik.

A Cleanroom-folyamat modellje a 22.10. ábrán látható. A szoftverfejlesztés ezen megközelítésének célja a hibamentes szoftver. A „Cleanroom" név a félvezetőgyártás egységeinek analógiájából származik. Ezekben az egységekben (cleanroomokban) a hibákat úgy kerülik el, hogy a gyártást ultratiszta légkörben végzik. Azért ebben a fejezetben írok erről, mert a rendszerkomponensek modultesztelését helyettesíti átvizsgálásokkal, amelyek e komponensek specifikációjukkal való konzisztenciájának ellenőrzését végzik.



22.10. ábra. A Cleanroom fejlesztési folyamat

A szoftverfejlesztés Cleanroom-féle megközelítése öt fő jellemzőn alapszik:

1. *Formális specifikáció.* A kifejlesztendő szoftver formális megadása. A specifikáció a rendszer ingerekre adott válaszait bemutató állapotátmenet-modellel adható meg.
2. *Inkrementális fejlesztés.* A szoftvert a Cleanroom-folyamat használatával különállóan fejleszthető és validálható inkrementumokra osztjuk. Ezeket az inkrementumokat, a felhasználói inputokkal együtt, a folyamat egyik korai szakaszában kell megadni.
3. *Strukturált programozás.* Csak korlátozott számú vezérlési és adatabsztrakciós konstrukciót használunk. A programfejlesztés folyamata a specifikáció lépésenkénti finomításának folyamata. Csak korlátozott számú konstrukciót használunk, és a cél az, hogy a specifikáció szisztematikus transzformálásával álljon elő a programkód.
4. *Statikus verifikáció.* A kifejlesztett szoftvert szigorú szoftverátvizsgálások használatával statikus verifikációnak vetjük alá. A kódkomponenseken nem végzünk modultesztelést.
5. *A rendszer statisztikai tesztelése.* Az integrált szoftverinkrementumot a 24. fejezetben leírtak szerint, megbízhatóságának meghatározása céljából statisztikailag teszteljük. Ezek a statisztikai tesztek, mint ahogyan az a 22.10. ábrán is látható, a rendszer specifikációjával párhuzamosan kifejlesztett működési profilon alapszanak.

A nagy rendszerek fejlesztésekor a Cleanroom-folyamatban három csapat vesz részt:

1. *A specifikációs csapat.* Ez a csapat felelős a rendszer specifikációjának kifejlesztéséért és karbantartásáért. Ők hozzák létre a vásárlóorientált specifikációkat (követelmények definíciója) és a verifikáció számára létrejövő matematikai specifikációkat is. Bizonyos esetekben – ha a specifikáció már teljes – a specifikációs csapat felelősséget vállalhat a fejlesztésért is.

2. *A fejlesztőcsapat.* Ez a csapat felelős a szoftver kifejlesztéséért és verifikálásáért. A szoftvert a fejlesztés során nem futtatják le. A kód átvizsgálásán alapuló strukturált, formális, helyességbizonyítási eszközökkel kiegészített verifikációt használnak.
3. *A hitelesítési csapat.* Ez a csapat a kifejlesztett szoftvert próbára tevő statisztikai tesztek kialakításáért felel. Ezek a tesztek a formális specifikáción alapulnak. A tesztesetek kitalálása a szoftverfejlesztéssel párhuzamosan végezhető. A teszteseteket a szoftver megbízhatóságának igazolására használják. A megbízhatóságnövelési modellek (24. fejezet) használhatók annak eldöntésére, hogy mikor kell a tesztelést abbahagyni.

A Cleanroom-módszer használata nagyon kevés hibát tartalmazó szoftvereket eredményezett. Cobb és Mills számos sikeres Cleanroom-alapú fejlesztési projektről számolnak be, amelyek egységesen alacsony hibaarányt mutattak a leszállított rendszerekben (Cobb és Mills, 1990). Ezen projektek költségei összemérhetők voltak a hagyományos fejlesztési technikákat használó projektekéivel.

A Cleanroom-folyamat inkrementális fejlesztési szemlélete az, hogy a kritikus vásárlói funkciókat a korai inkrementumokban kell leszállítani. A kevésbé lényeges rendszerfunkciók a későbbi inkrementumokba kerülnek. A vásárló éppen ezért még a teljes rendszer leszállítása előtt kipróbálhatja ezeket az inkrementumokat. Ha követelménybeli probléma kerül a felszínre, akkor a vásárló ezt az információt visszacsatolja a fejlesztőcsapatnak, és új kibocsátást igényel az inkrementumból.

Az extrém programozáshoz hasonlóan ez azt jelenti, hogy a legfontosabb vásárlói funkciókat validálják a legtöbbet. Az új inkrementumokat kombináljuk a már létezőkkel, és az integrált rendszert teszteljük. Ezért a már létező inkremensek az új rendszerinkremensek hozzáadásakor új tesztesetekkel újratesztelésre kerülnek.

A szigorú programátvizsgálás a Cleanroom-folyamat kulcsfontosságú része. A rendszer specifikációjaként létrejön a rendszer egy állapotmodellje, és különböző, egyre részletesebb rendszermodellek sorozatán keresztül ezt finomítják futtatható programmá. A fejlesztéshez használt szemlélet olyan jól definiált transzformációkon alapszik, amelyek megpróbálják minden részletesebb reprezentációba való transzformációnál megőrizni a helyességet. Az új reprezentációt minden lépésben át kell vizsgálni, és a transzformáció kimenetének és bemenetének konzisztenciáját bemutató szigorú matematikai állításokkal kell kiegészíteni.

A Cleanroom-folyamatban használt matematikai eszközök gyengébbek a formális matematikai bizonyításoknál. Azok a formális matematikai bizonyítások, amelyek azt igazolják, hogy a program megfelel a specifikációjának, csak nagyon drágán fejleszthetők ki. Ezek azon programozási nyelv formális szemantikájának ismeretétől függnek, amelyen megírjuk a programot, és olyan matematikai tételeket kell konstruálni, amelyek összekapcsolják a programot annak formális specifikációjával. Ezután ezeket a tételeket kell matematikailag bebizonyítani, gyakran nagy és bonyolult tételbizonyító programok segítségével. A magas költségek és az igényelt speciális szaktudás miatt az ilyen bizonyításokat általában

csak a nagymértékben biztonságosságkritikus és védettségkritikus alkalmazások számára végzik el.

A Cleanroom-folyamatban az átvizsgálás és a formális verifikáció igen hatékonynak bizonyult. A hiányosságok, hibák nagy többségét a végrehajtás előtt felfedezik, így azok nem kerülnek be a kifejlesztett szoftverbe. Linger (Linger, 1994) azt írja, hogy a Cleanroom-projektek tesztelése során a forráskódban ezersoronként átlagosan csupán 2,3 hibát fedeztek fel. A fejlesztési költségek összességében nem nőttek, mivel a kifejlesztett szoftver tesztelésére és javítására kevesebb erőfeszítésre volt szükség.

Selby és mások (Selby *és mások*, 1987) egyetemi hallgatókat alkalmaztak fejlesztőként, és így hasonlították össze a Cleanroom-fejlesztést a hagyományos technikákkal. Azt vették észre, hogy a legtöbb csapat sikeresen alkalmazta a Cleanroom-módszert. Az így létrejött programok jobb minőségűek voltak, mint a hagyományos technikákkal fejlesztettek – a forráskód több megjegyzést tartalmazott és egyszerűbb szerkezetű volt. A Cleanroomot használó csapatok többségének sikerült betartania a fejlesztési határidőt.

A Cleanroom-fejlesztés csak akkor működik, ha képzett és elkötelezett tervezők végzik. A Cleanroom-módszer iparbeli sikeréről szóló jelentések, ha nem is kizárólag, de azért leggyakrabban az irányába elkötelezett emberektől érkeztek. Nem tudjuk, hogy ez a folyamat hatékonyan átvihető-e másfajta szoftverfejlesztő szervezetekhez is. Ezek a szervezetek általában kevesebb, kevésbé elkötelezett és kevésbé képzett tervezőt alkalmaznak. A Cleanroom-módszer – vagy bármely más, szintén formális módszereket használó elképzelés – ilyen szervezetekhez történő átvitele továbbra is kihívás marad.

Kulcsfogalmak

- A verifikáció és a validáció nem ugyanaz a dolog. A verifikáció arra szolgál, hogy megmutassuk, hogy a program megfelel a specifikációjának. A validáció annak igazolására szolgál, hogy a program azt teszi, amit a felhasználó szeretne.
- A teszttervnek tartalmazniuk kell a tesztelendő elemek leírását, a tesztelés ütemezését, a tesztelési folyamat kezelésére szolgáló eljárásokat, a hardver- és szoftverkövetelményeket és a valószínűleg felmerülő tesztelési problémákat.
- A statikus verifikációs technikák magukban foglalják a program forráskódjának vizsgálatát és elemzését hibafelderítés céljából. Ezeket a technikákat a programteszteléssel együtt a V & V folyamat részeként célszerű használni.
- A programátvizsgálások hatékony módjai a programhibák megtalálásának. Egy átvizsgálás célja a hibák behatárolása. Az átvizsgálási folyamatot egy hibaellenőrző listának kell vezérelnie.
- A programkódot egy kis csapat szisztematikusan ellenőrzi. A csapattagok között van egy csapatvezető, azaz a moderátor, a kód szerzője, az olvasó, aki az átvizsgálás során bemutatja a kódot, és a tesztelő, aki tesztelési szempontból tekint a kódra.

- A statikus elemzők olyan szoftvereszközök, amelyek a program forráskódjának feldolgozásával az olyan anomáliáknak szentelnek figyelmet, mint a nem használt kódrészletek, vagy az inicializálatlan változók. Ezek az anomáliák a kódban ejtett hibák eredményei lehetnek.
- A Cleanroom-szoftverfejlesztés a statikus programverifikációs technikákon és a rendszer megbízhatóságának igazolására statisztikai teszteken alapuló szoftverfejlesztési megközelítés. Nagy megbízhatóságú rendszerek készítésében igen sikeres.

További irodalom

Software Quality Assurance: From Theory to Implementation. Ez a könyv jó általános áttekintést ad a verifikációról és a validációról, egy különösen jó, átvizsgálásról szóló fejezettel. (Galin, D., 2004, Addison-Wesley.)

„Software Inspection". Egy folyóirat különszáma, amely számos, programátvizsgálásról szóló cikket tartalmaz, azt is taglalva, hogy miként alkalmazható ez a technika az objektumorientált fejlesztésben. (*IEEE Software*, 20(4), July/August 2003.)

„Software debugging, testing and veification". A verifikációról és a validációról szóló általános cikk, egyike azon keveseknek, amelyek mind a tesztelésről, mind pedig a statikus verifikációs technikákról írnak. (Hailpern, P. és Santhanam, B., *IBM Systems Journal*, 41(1), January, 2002.)

Cleanroom Software Engineering: Technology and Process. A Cleanroom szemléletmódról szóló könyv, amely több szakaszt szentel a technika alapjainak, a folyamatnak és egy gyakorlati esettanulmánynak. (Powell, S. J. *és mások*, 1999, Addison-Wesley.)

Feladatok

22.1. Fejtse ki, hogy milyen különbségek vannak a verifikáció és validáció között, és magyarázza el, hogy a validáció miért különösen bonyolult folyamat.

22.2. Magyarázza el, hogy miért nem szükséges egy programnak teljesen hibamentesnek lennie, mielőtt leszállításra kerül. Milyen mértékig használható a tesztelés annak validálására, hogy a program megfelel céljának?

22.3. A 22.4. ábra tesztterv*et* egyedi rendszerek számára alakították ki, amelyek rendelkeznek különálló, követelményeket leíró dokumentummal. Tegyen javaslatot a teszterv szerkezetének változására annak érdekében, hogy hiányosan dokumentált szoftvertermékek tesztelésére is alkalmas legyen.

22.4. Magyarázza el, hogy a programátvizsgálási technikák miért jelentik hatékony módját a programban található hiányosságok felderítésének. Milyen hibafajták felderítése nem valószínű átvizsgálások segítségével?

22.5. Magyarázza el, hogy egy versenyképes, elitista kultúra miért tarthatja bonyolultnak a programátvizsgálások V & V technikaként történő bevezetését.

22.6. A Java, C++, C és egyéb programozási nyelvekre vonatkozó ismereteivel készítsen egy ellenőrző listát az olyan gyakori (nem szintaktikai) hibákról, amelyeket a fordítóprogram nem vesz észre, de programátvizsgálással felderíthetők.

22.7. Készítsen egy állapotlistát, amelyet egy Java, C++ vagy valamilyen, ön által használt programozási nyelv számára készült statikus elemző képes lehet felismerni. Hasonlítsa össze ezt a listát a 22.7. ábrán láthatóval.

22.8. Magyarázza el, hogy miért lehet a formális módszerek használata biztonságosságkritikus rendszerek fejlesztése esetén költséghatékony. Mit gondol, az ilyen rendszerek fejlesztőinek egy része miért ellenzi a formális módszerek használatát?

22.9. Egy menedzser úgy dönt, hogy a programátvizsgálások jelentéseit a dolgozók értékelési folyamatának bementeként fogja használni. Ezekben a jelentésekben az olvasható, hogy ki ejtette, és ki fedezte fel az egyes programhibákat. Etikus menedzseri viselkedés ez? Etikus volna akkor, ha a dolgozókat előre tájékoztatnák, hogy ez fog történni? Milyen eltérést jelenthet ez az átvizsgálási folyamatban?

22.10. A rendszertesztelésre alkalmazott egyik gyakori megközelítés a rendszer addig történő tesztelése, amíg a tesztelésre fordított költségvetés ki nem merül, majd a rendszert ezek után leszállítják a vásárlóknak. Fejtse ki ennek etikáját.