



Szoftvertchnológia

7. Tesztelés (2)

BSc kurzus

Dr. Balla Katalin



Tartalom

- Tesztelési technikák
- Statikus tesztelési technikák
- Dinamikus tesztelési technikák
- Tesztelés agilis környezetben



Tesztelési technikák

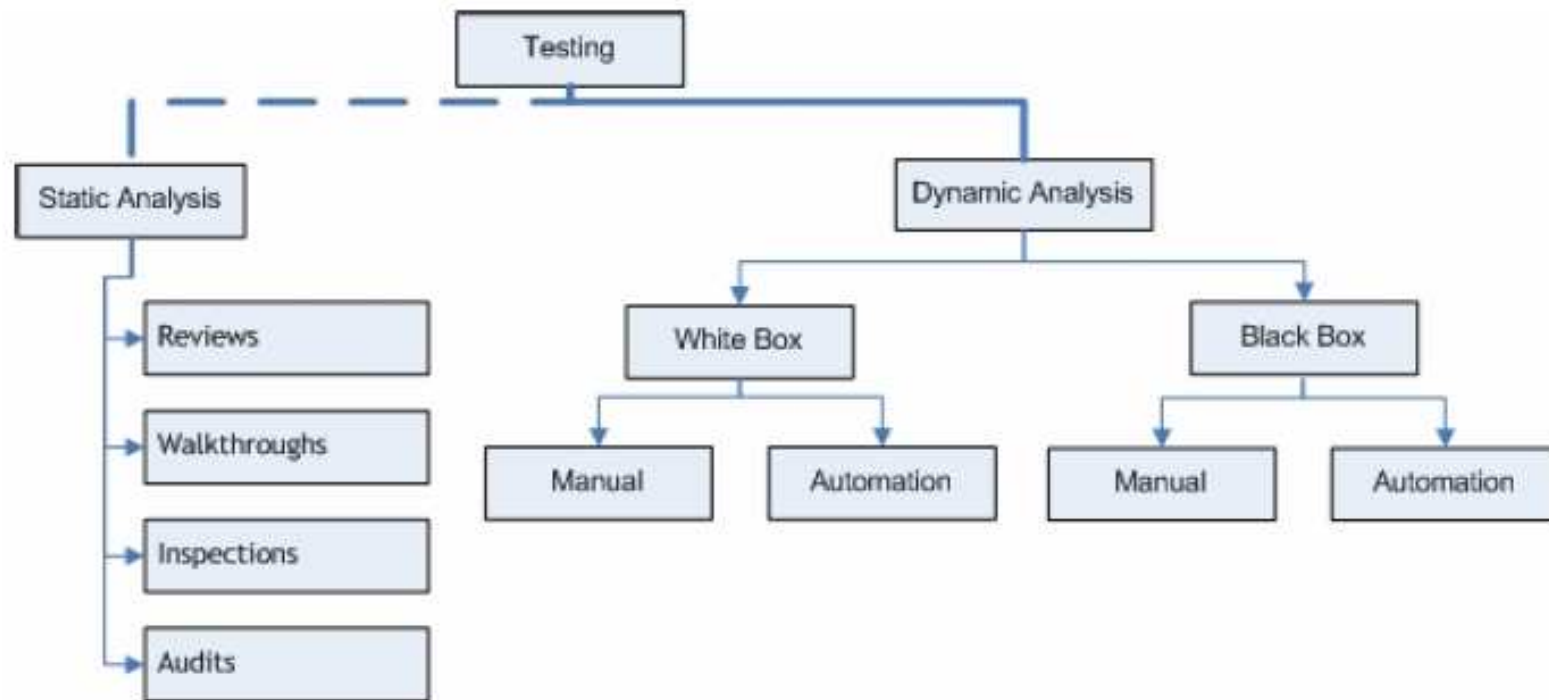
- Segítenek a megfelelő tesztelési módot / eseteket kiválasztani
- Alapvető megközelítések:
 - A szoftver belső szerkezetének / felépítésének vizsgálata (a program futtatása nélkül) : statikus tesztelés
 - A szoftver működés közbeni vizsgálata: dinamikus tesztelés



Tesztelési technikák

- Nincs „univerzálisan legjobb” tesztelési technika
 - De:
 - vannak általánosan érvényes elméletek
 - vannak a különböző tesztelési technikákra vonatkozó tapasztalatok
- Mindig a konkrét eset függvényében lehet az „abban az esetben legjobb” tesztelési technikát kiválasztani
 - Tesztelő, fejlesztő és felhasználó véleménye fontos!

Tesztelési technikák

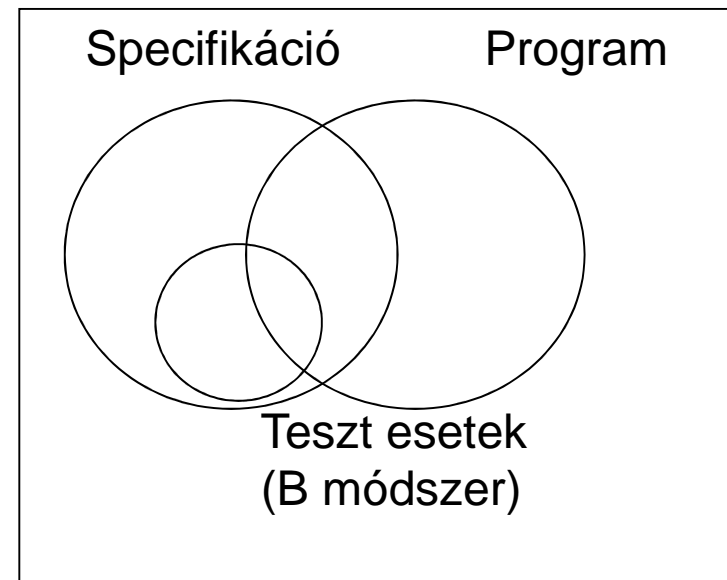
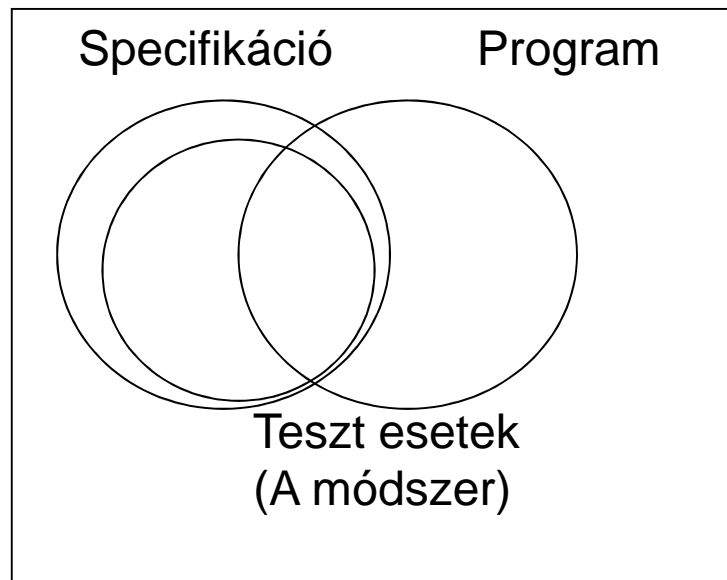




Tesztelési technikák

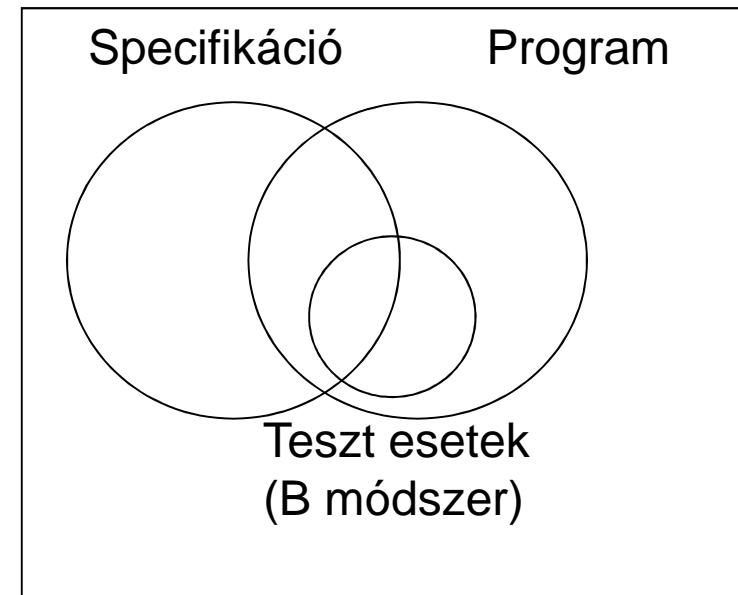
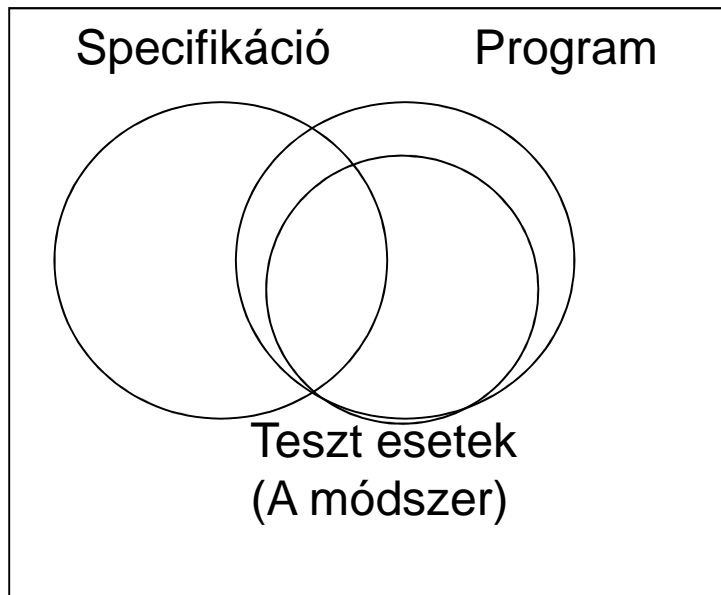
- A kiválasztott tesztelési technika segít a tesztek tervezésében, a teszt esetek azonosításában

Teszt esetek azonosítása funkcionális teszthez

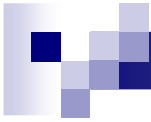


Kérdés: melyik módszer jobb?

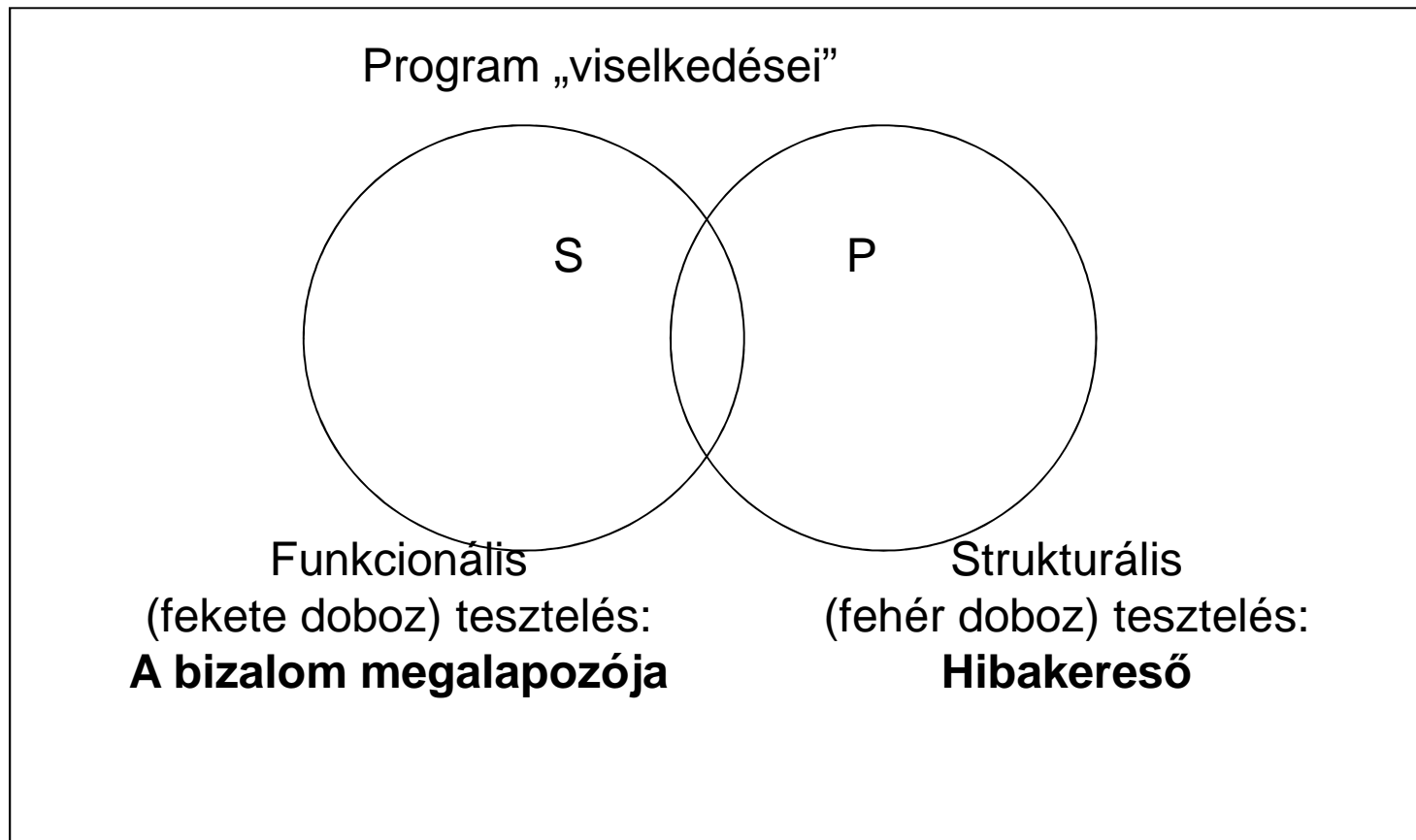
Teszt esetek azonosítása strukturális teszthez



Kérdés: melyik módszer jobb?



Teszt esetek azonosítása





Tesztelési technikák

- Általában a *statikus* és *dinamikus*, az utóbbin belül pedig a *strukturális* és *funkcionális* technikákat **megfelelő arányban kombinálva** alakul ki az adott esetben jó tesztelési módszer
- A jó tesztelési módszert „kikeverni” nem egyszerű feladat! Szaktudást igényel!



Statikus tesztelés

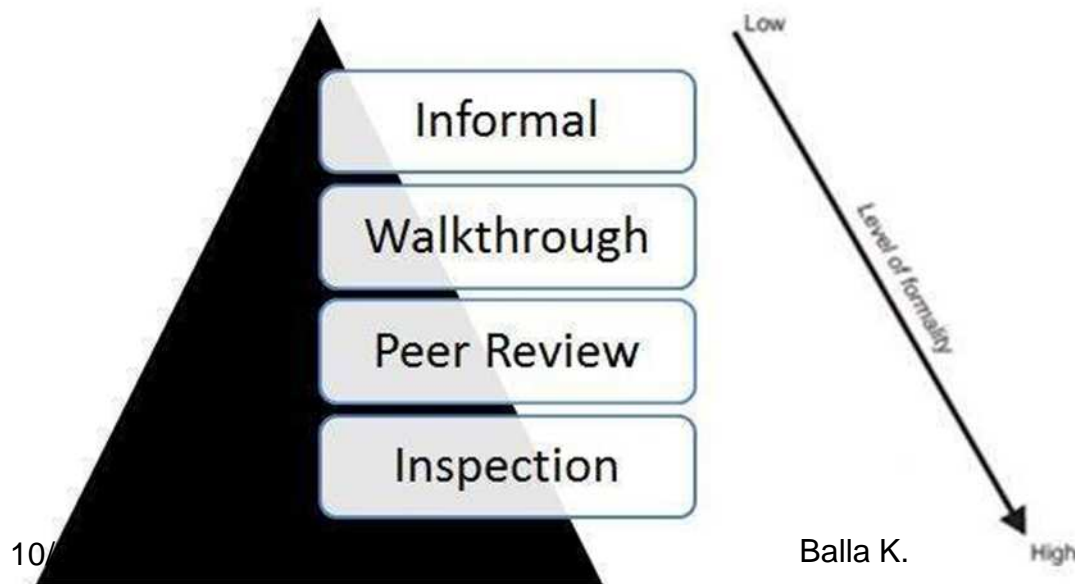
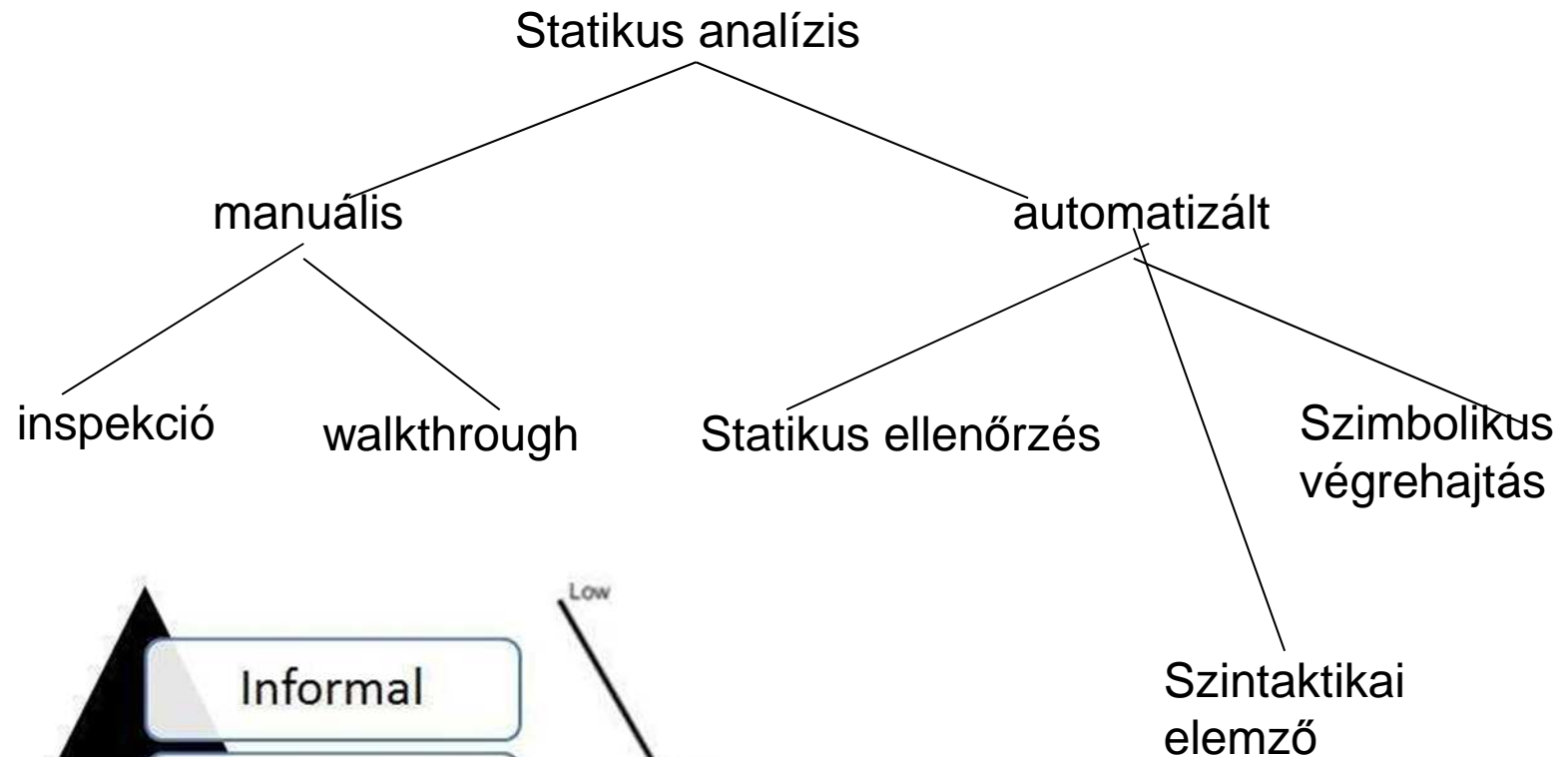
- A tesztelésnek az a formája, amikor magát a szoftvert nem használjuk (nem futtatjuk)
- Statikus tesztelési módszerek:
kódszemlézés, inspekció, „walkthrough”,
statikus kódelemzés
- Mindegyik emberi „megfigyelésen” alapul
- Bármilyen munkatermékre alkalmazható
- Hatásos



Statikus tesztelés

- Alapelv: futtatással nem tudunk mindent lefedni – viszont a teljes kódot át lehet nézni
- Cél nemcsak a hibák megtalálása, hanem okaik felderítése és további előfordulásuk meggátolása (pl. a folyamat javításával)
- Egy komponens vagy rendszer tesztelése specifikáció, vagy implementáció szinten **a szoftver futtatása nélkül**. Például felülvizsgálat vagy statikus forráskód elemzés (HTB).

Statikus tesztelés





Statikus tesztelés

■ Statikus analízis technikák

□ Alapvetően 3 módon:

- a program belső szerkezetének vizsgálata
- a program teljességének és konzisztenciájának vizsgálata előre meghatározott szabályok alapján
- a program összehasonlítása a specifikációjával vagy dokumentációjával.



Statikus kódelemzés

- A szoftver forráskódjának elemzése azzal a céllal, hogy megértsük, mit csinál a szoftver. Ugyanakkor, helyességi kritériumokat is felállítunk.
- Többféle elemzési technika van, egy részüket eszköz is támogatja
 - A „veszélyesnek látszó” elemeket keressük
 - Formális módszerek, amelyekkel a program szemantikája matematikailag leírható



Statikus analízis

- Formális módszer-család, amellyel egy szoftver viselkedéséről automatikusan információ nyerhető
- A statikus analízis egyik alkalmazása pl. egy debuggoló, amely a futási időben történő hibákat segít megtalálni
- A statikus analízis eredménye nem egyértelmű: tulajdonképpen semmilyen módszerrel nem lehet eldönteni, hogy futtatás során megjelennek-e hibák
- Technikák
 - Modell ellenőrzése



Ellenőrzés papíron

- **A** szoftver, vagy a specifikáció tesztelése a végrehajtás kézi szimulálása által. Lásd még: statikus analízis (*desk checking*)



Walkthrough / átvizsgálás

- Egy dokumentum szerzője által végzett lépésenkénti bemutató abból a célból, hogy információt gyűjtsön valamint közös álláspontot alakítson ki. (HTB, [Freedman és Weinberg, IEEE 1028])



Walkthrough

- Tippek:

- Osszuk fel a kódot részekre. Mindegyik rész egy „célhoz” kapcsolódjék.
 - „Részek” értelme, hossza: programtól függ. „Egy „rész” annyit valósít meg, hogy érdemes legyen célt találni hozzá.”
- Azonosítsuk / értsük meg minden változó jelentését
- Keressünk ismert hibákat
 - Pl. indexben a ciklusváltozó. $<$ vagy \leq ?
 - `for (index = 0; index <= MAX_COUNT; index++) { array[index]=j; }`
- Határozzuk meg a walkthrough inputjait
 - A kódhoz / kód-részekhez válasszunk inputokat
- „Haladjunk végig” a kódon
 - Úgy gondolkodjunk, ahogyan egy számítógép dolgozik!

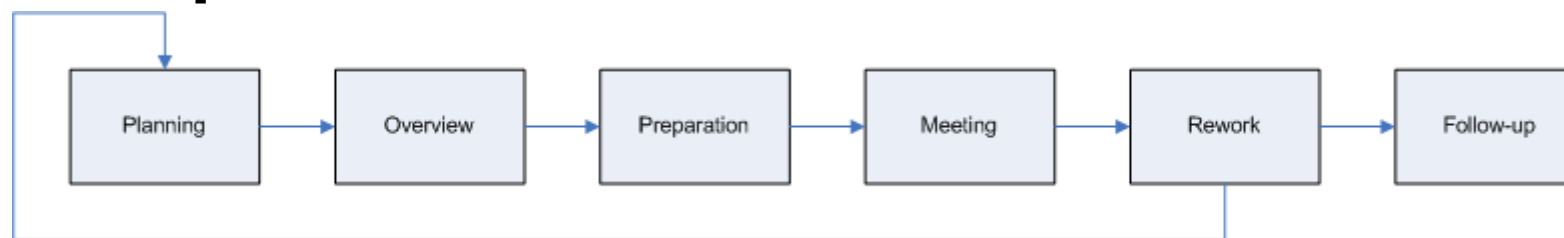
- Forrás: Find the Bug A Book of Incorrect Programs. By [Adam Barr](#). Publisher: Addison Wesley Professional Pub Date: October 06, 2004 ISBN: 0-321-22391-8



Inspekció / vizsgálat / felülvizsgálat

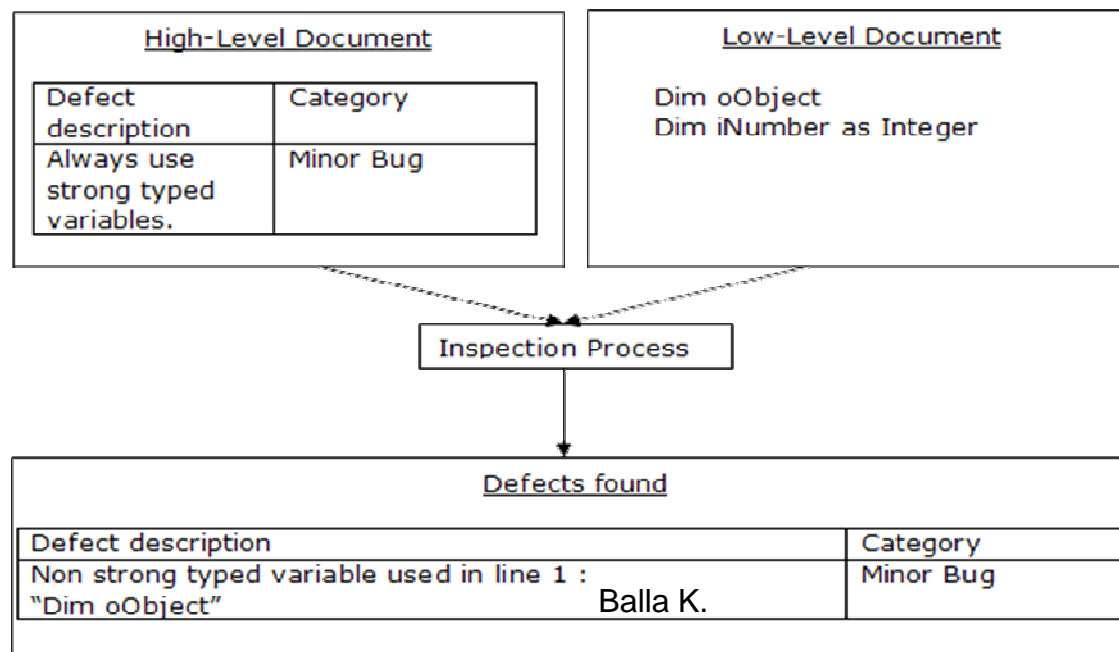
- Az egyenrangú felülvizsgálat egy típusa, amely a dokumentum vizuális vizsgálatán alapul, hogy megtaláljuk a hibákat, vagy pl. a szabványokhoz képest meglevő különbségeket, illetve a magasabb szintű dokumentációktól való eltéréseket. A leginkább formális felülvizsgálati módszer, amely emiatt mindig dokumentált eljáráson alapul. (HTB)
- Bármely munkatermék képzett szakértők által, jól meghatározott szabályok szerint végzett felülvizsgálata . Célja hibák megtalálása (saját hibáinkat nem vesszük észre)
- Inspekciót a fejlesztés bármelyik fázisában lehet alkalmazni: követelményekre, tervekre, kódra, teszt esetekre.
- Inspekció: elsősorban a kód felülvizsgálatát szokta jelenteni
- Az inspekció a leghatásosabb az összes szemle között – viszont költséges és nehéz bevezetni
- 2-3 szemlélő alkalmazása a leghatékonyabb.

Inspekció



Fagan alapmodellje az inspekcióhoz. Eredetileg az IBM számára dolgozta ki. Lényege a strukturáltság.

Sokan és sokféleképpen fejlesztették tovább, pontosították.






Inspekció

- A Fagan-féle inspekció lényeges eleme az inspekcióban részt vevő szerepkörök definiálása
 - Moderátor
 - Időzítés, az inspekció hosszának meghatározása, a megbeszélés követése / vezetése
 - Olvasó
 - „idegenvezető” az inspekció alatt. Soronként, hangosan felolvassa az inspektált dokumentumot
 - Szerző
 - A szemlézett elem áttekintése, válasz a kérdésekre, javítások
 - „Jegyző”
 - Dokumentálja az inspekció eredményeit
 - Inspektorok
 - Előkészülnek az inspekcióra, részt vesznek az inspekción
 - Gyakorta megesik, hogy bizonyos szakemberek csak bizonyos aspektusokkal foglalkoznak



Inspekció - tapasztalatok

- Csökkentheti a tesztelési költségeket, akár a fejlesztési költségek 20-30 %-ra.
- Elmaradhat az egységteszt és integrált teszt, csak a rendszerteresztet kell elvégezni
 - Pl: Rendszerterv inspekciója a követelményekhez viszonyítva, kód inspekciója a rendszertervvel összevetve stb.
- (Ezzel együtt, kevesen „merik” alkalmazni)



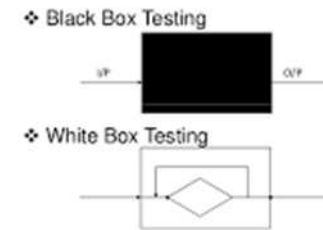
Peer review / egyenrangú szemle / egyenrangú felülvizsgálat

- A szoftverfejlesztés alatt végzett tevékenységek felülvizsgálata nem a tevékenységet elvégző által, melynek célja, hogy hibákat fedezzen fel illetve javító javaslatokat hozzon. Példák: vizsgálat, technikai felülvizsgálat, átvizsgálás. (HTB)

Tesztelési technikák

■ Dinamikus tesztelés

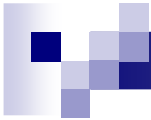
- A rendszer /modul futását feltételezi
- A rendszer futtatásának próbája, tesztkörnyezetben, tesztadatokkal
- Típusai:
 - Követelményeken alapuló: **feketedoboz teszt.**
 - A szoftver belső szerkezetén alapuló: **strukturális / fehérdoboz teszt.**





Feketedoboz tesztelés

- A program belső szerkezetére történő hivatkozás nélküli funkcionális, vagy nem-funkcionális teszt. → *black box testing*
- **Feketedoboz teszttervezési technika:** olyan módszer, amelynél a szoftver specifikáció alapján, a program belső szerkezetének ismerete nélkül tervezünk tesztek.



Feketedoboz tesztelés

- Tipikus feketedoboz teszt tervezési technikák:
 - ☐ Határérték tesztelés
 - ☐ Ekvivalencia partícionálás
 - ☐ Döntési táblák
 - ☐ Állapotátmenet tesztelés
 - ☐ Use case tesztelés



Határérték elemzés

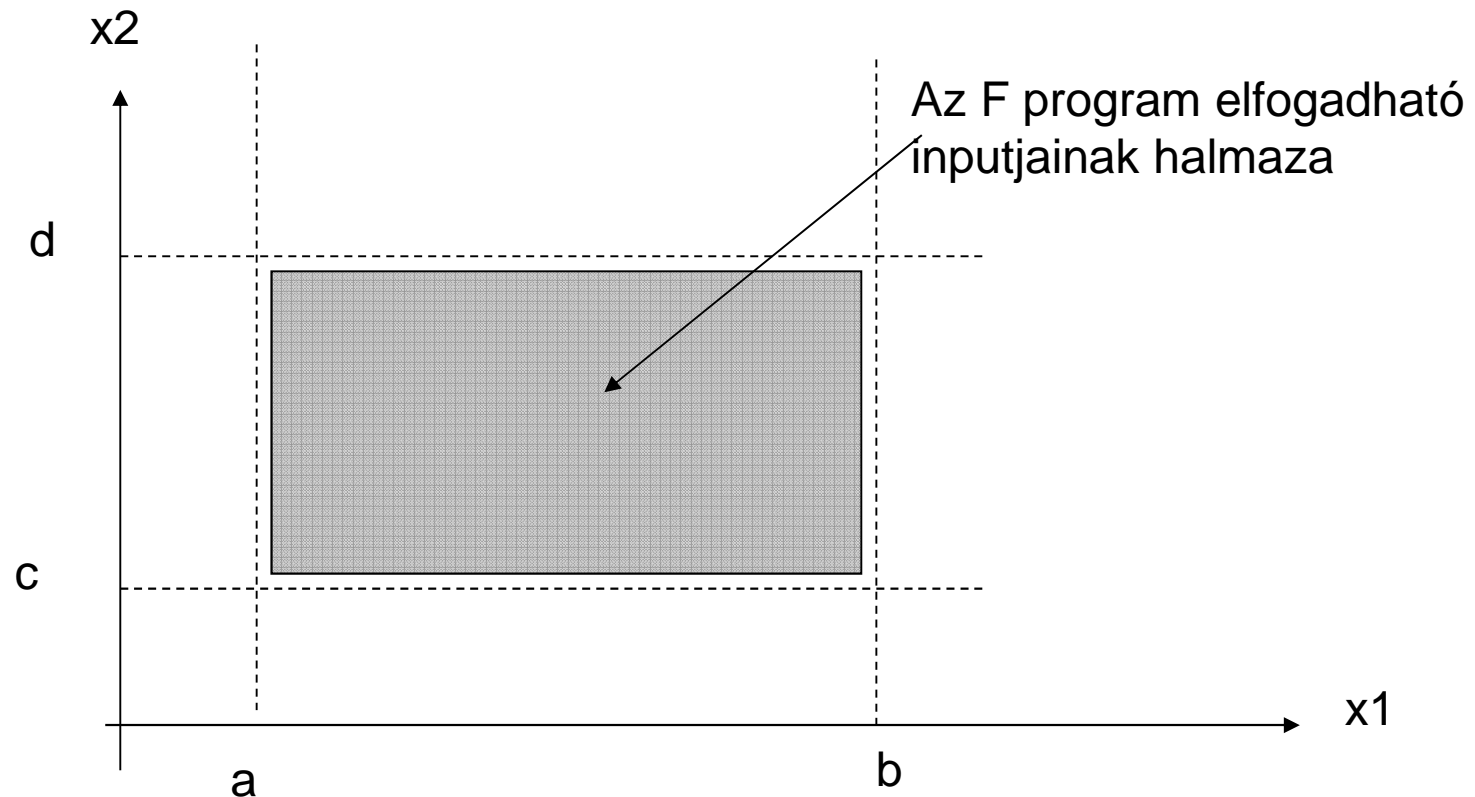
- A program változóinak, illetve paramétereinek szélsőérték- elemzésén alapuló feketedoboz teszttervezési technika. → *boundary value analysis*



Határérték analízis

- $Y = F(x_1, x_2)$
- Ha F egy programként kerül implementálásra, x_1 és x_2 bizonyos korlátok között mozognak (a korlátokat nem mindig adjuk meg explicit módon)
 - $a \leq x_1 \leq b$
 - $c \leq x_2 \leq d$

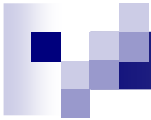
Kétváltozós függvény input tartománya





A határérték analízis alapelvei (1)

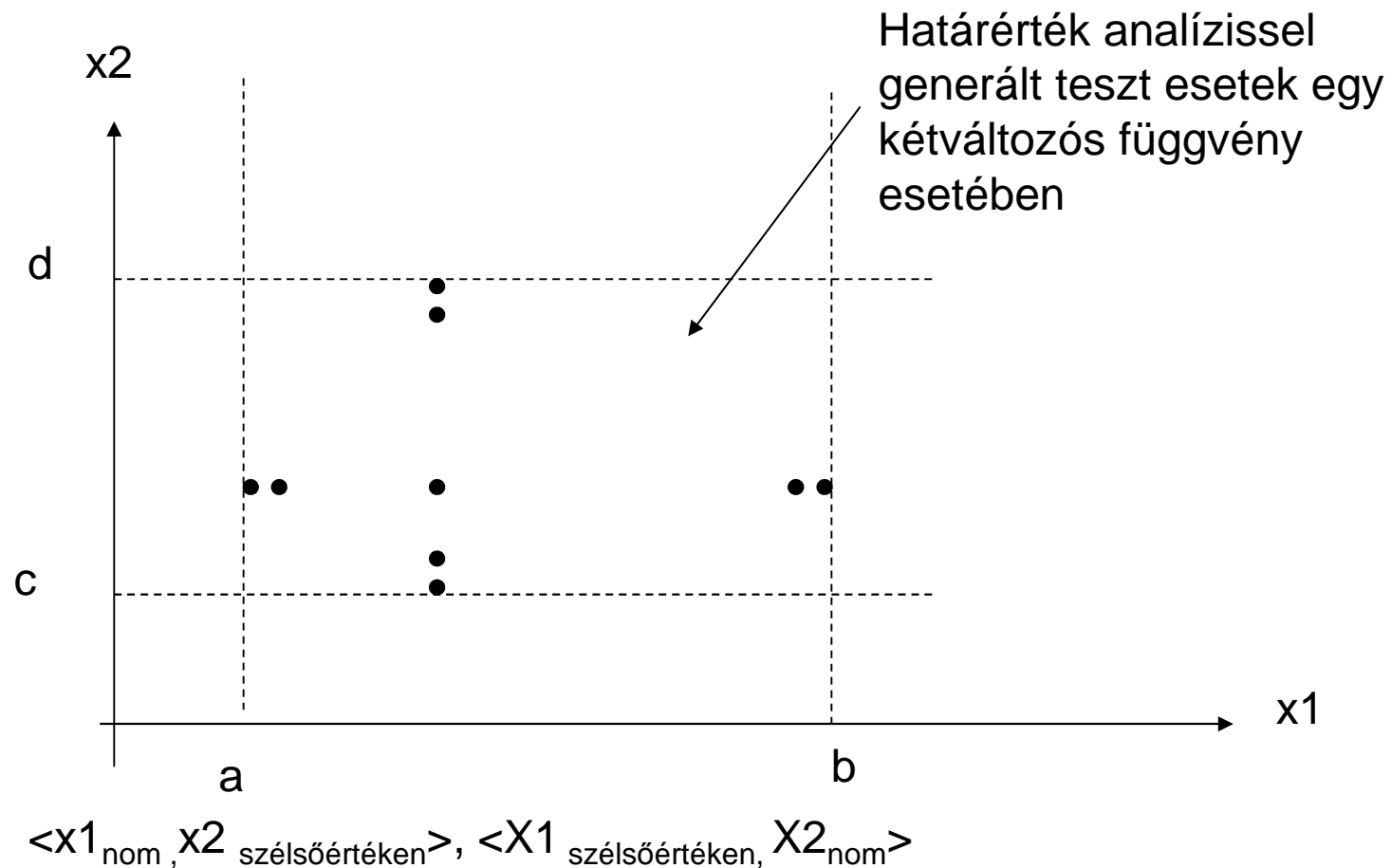
- Az ezzel a módszerrel generált tesztesetek *az input tartomány határán található értékeket használják*
 - Magyarázat: megfigyelték, hogy a hibák sokszor jelentkeznek az input értékek megengedett határain vagy azok közelében.
 - Pl: „loop condition” -ban $<$ tesztet használnak \leq helyett; szövegszerkesztőben új sor beszúrása Page Break előtt – „nyomtatott forma” megjelenítésben a beszúrt sor eltűnik...
- Tesztesetek kialakítása: a következő input adatokkal való tesztelés: minimum érték (min), „éppen csak” minimum feletti (min+), nominális érték (nom), „éppen csak” maximum alatti (max-), maximum érték (max)



A határérték analízis alapelvei (2)

- Feltételezés: egy megjelenő hiba (failure) csak nagyon ritka esetben egynél több belső hiba (fault) eredménye.
- Határérték analízissel kialakított teszt esetek generálása: az összes változót nominális értéken tartjuk, egy kivételével. Ez a kivétel veszi fel a szélső értékeket.

Teszt esetek generálása határérték analízissel





A határérték analízis általánosítása

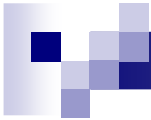
- Kétféleképpen:

- Változók számának növelése

- n változós függvények esetében határérték analízis eredményeképpen $4n + 1$ teszt eset alakul ki (esetenként $n-1$ változó marad nominális értéken, a megmaradt változó pedig sorra felveszi a min, min+, nom, max-, max értékeket)

- A korlátok kiterjesztése / általánosítása

- A változók típusától függően

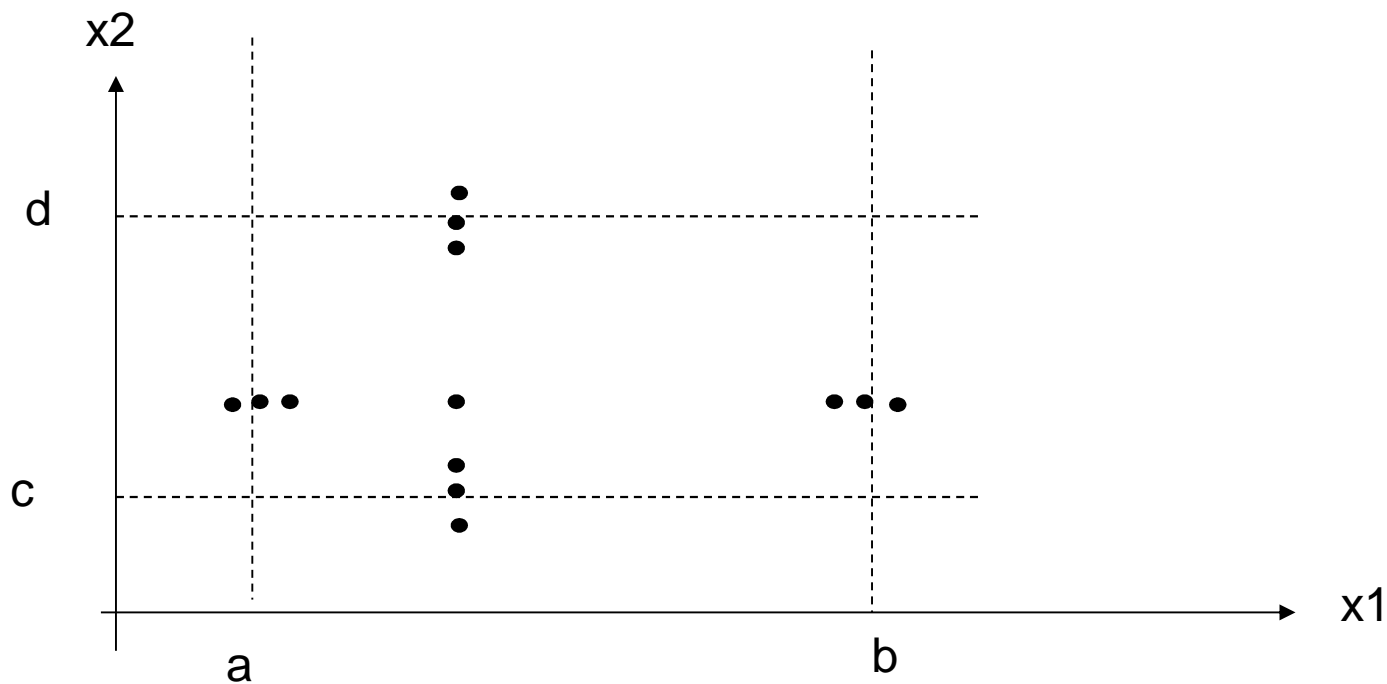


A határérték analízis általánosítása

- A korlátok kiterjesztése / általánosítása – pl:
 - Hónapok: [1,12]
 - Ha a változók halmaza diszkrét, jól meghatározott, nem okoz gondot a határértékek meghatározása
 - Ha nincsenek explicit korlátok, a tesztelőnek kell megalkotnia őket (pl. háromszög oldalainak hossza: [1,200] vagy [1,a legnagyobb, adott programnyelvben reprezentálható szám])
 - Nem értelmes a határérték analízis a Bool változók esetében (a szélsőértékek: True és False, de mi a többi???)– itt inkább döntési táblákkal tesztelünk

Robosztusság tesztelés

- A határérték analízis kiterjesztése: az előbbi 5 értéken kívül teszteljük az „éppen csak” maximum feletti (max+) és „éppen csak” minimum alatti (min-) értékeket is





„Legrosszabb eset” teszt

- Nem fogadjuk el a határérték analízis második alapelvét (*a megjelenő hibák (failure) csak nagyon ritka esetben egynél több belső hiba (fault) eredményei.*)
- Megvizsgáljuk, mi történik, ha egynél több változónak van szélső értéke.
 - Minden változó esetében az 5 értéket vesszük (min, min+, nom, max, max-), és ezen értékek Descartes szorzatát képezve generálunk teszt eseteket (5^n teszt esetet kapunk)



A határérték analízis korlátai

- Nem veszi figyelembe a függvény típusát, sem a változók jelentését
 - A határérték analízissel generált tesztek kezdetlegesek: kevés, a programra vonatkozó ismeretre és képzelőerőre alapoznak
- Akkor alkalmazható hatékonyan, ha program *független* változók függvénye, és ezek a változók „korlátos” *fizikai változókat* jelölnek.



Határérték tesztelés - összefoglalás

- A legkezdetlegesebb tesztelési forma
- Független változókat és fizikai mennyiségeket feltételez
- „Egyszerre egy hiba” elve
- A robosztus teszttel, valamint a „legrosszabb eset” teszttel kombinálva jó eredményeket ad
- A robosztussági teszt jól használható a belső változók tesztelésére
- Jól használható a hibaüzenetek figyelésére



Ekvivalencia partícionálás

- Feketedoboz tesztelési módszer, amely során olyan teszteseteket készítünk, amelyek az ekvivalencia partíciók egyes reprezentánsait tesztelik. Jellemzően minden egyes ekvivalencia partíciót érdemes legalább egyszer lefedni. → *equivalence partitioning*



Ekvivalencia partícionálás

■ Miért használjuk?

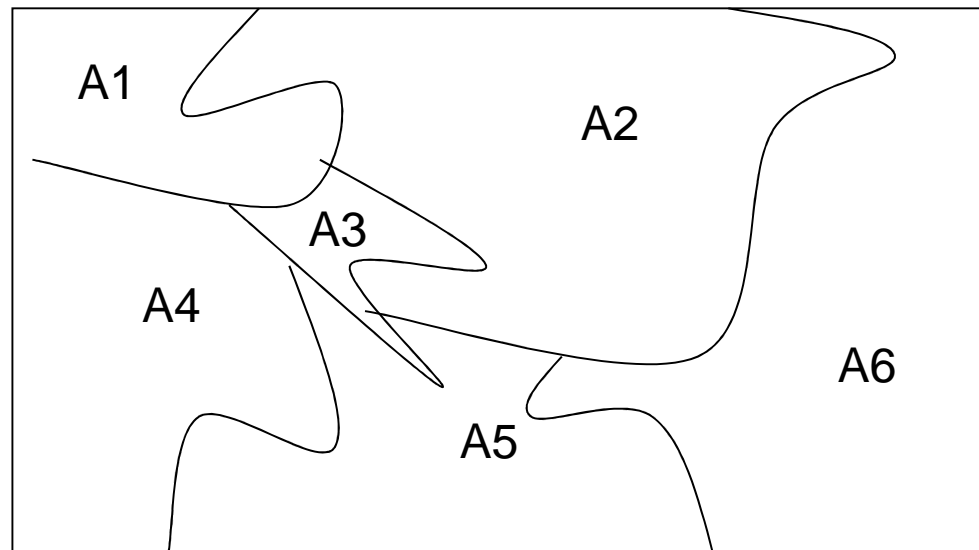
- Szeretnénk biztosítani, hogy tesztelésünk „teljes”
- Szeretnénk elkerülni a redundáns adatokkal való tesztelést


■ Ekvivalencia osztályok

- Egy halmaz partícióját alkotják. A partíció elemei diszjunktak, egyesítésük pedig a teljes halmaz.
 - A teszt adatokat úgy kiválasztva, hogy minden ekvivalencia osztályból legyen egy input teszt adat, biztosítható a tesztelés teljessége
 - A diszjunkság a redundancia mentességet biztosítja

Ekvivalencia partícionálás

- A teszt eseteket úgy határozzuk meg, hogy minden ekvivalencia osztályból egy elemet választunk





Ekvivalencia osztályok alkalmazása a tesztelésben

■ Kulcsfontosságú:

- Az ekvivalencia osztályokat (az ekvivalencia relációt) jól, okosan kell meghatározni!
 - Példa: a háromszög problémában az *output* alapján lehet ekvivalencia osztályokat meghatározni (és pl. nem érdemes 1,1,1 – 5,5,5 és 100,100,100 eseteket is kipróbálni)
 - Általában „kitaláljuk” az ekvivalencia osztályokat. Előfordul, hogy ebben a folyamatban arra is gondolunk, hogyan lehet implementálva a program.
- Ekvivalencia osztályok létrehozására esetenként saját módszereket, eljárásokat dolgoznak ki



Döntési táblákon alapuló tesztelés

- A funkcionális tesztelésben a legrigorózusabb technika – a logika szigorán alapszik
- Már az 1960-as években használtak döntési táblákat, komplex logikai összefüggések reprezentálására és elemzésére.
- A döntési táblák jól használhatók olyan esetek leírására, amikor különböző akciók kombinációira kerül sor bizonyos, változó feltételrendszerek mellett



Döntési tábla

Példa

	Szabály1	Szabály2	Szabály3	Szabály4	Szabály5	Szabály6
c1(cond)	T	T	T	F	F	F
C2	T	T	F	T	T	F
C3	T	F	-	T	F	-
A1(action)	X	X		X		
A2	X				X	
A3		X		X		
A4			X			X

C: feltétel

A: akció / tevékenység



Döntési táblák létrehozása

■ Ha

- Feltételek: input – (gyakran az input ekvivalencia osztályaira utal)
- Akciók: output – (gyakran a tesztelt elem főbb végrehajtandó részeire utal)

akkor a *szabályokat* teszt esetekként értelmezhetjük.

- Mivel a döntési tábla teljessége ellenőrizhető, biztosan tudhatjuk, hogy minden lehetőséget átfogó teszt eset-halmazt dolgoztunk ki.



Döntési táblák létrehozása

- Ha a redundáns bemenetekhez (szabályokhoz) ugyanaz az akció társul, nincs gond
- Előfordulhat, hogy redundáns bemenethez (ugyanahhoz a szabályhoz) többféle akció társul. Az ilyen döntési tábla inkonzisztens (nem lehet eldönteni, hogy adott feltételekkel melyik akció lép működésbe)
- Az ilyen eseteket meg kell szüntetnie a tesztelőnek.



A döntési táblák alkalmazása

- Akkor ajánlott, ha:
 - If-then-else logika van
 - Az inputok között logikai kapcsolatok vannak
 - Az inputok részhalmazait felhasználó számítások vannak
 - Az input és output között ok-okozati összefüggés van
 - Magas a ciklomatikus komplexitás
- Sok változó esetén a döntési tábla nagyon elbonyolódhat. Ezen különböző technikákkal segíthetünk (kiterjesztett táblák, algebrai egyszerűsítés...)
- A jó döntési tábla iterációk során alakul ki

Állapotátmenet teszt

- Olyan feketedoboz teszttervezési technika, amiben úgy tervezzük meg a teszteseteket, hogy érvényes és érvénytelen állapotátmeneteket generáljanak.

State	Entry action	Entry_Action1 Entry_Action2
	exit action	Exit_Action1 Exit_Action2
	Input_Action_Condition1	Input_Action1 Input_Action2
	Input_Action_Condition2	Input_Action3
Next_State1	Transition_Condition1	
Next_State2	Transition_Condition2	

Figure 3 State transition table

STATE	EVENT	ACTION	NEXT STATE
Wait For Dollar	Bill Detected	Load Bill	Verify Dollar
Verify Dollar	Verification Failed	Reject Bill	Wait For Dollar
Verify Dollar	Verification Passed	Dispense Coins	Dispensing Coins
Dispensing Coins	Sufficient Funds Remain	Accept Another Dollar	Wait For Dollar
Dispensing Coins	Insufficient Funds Remain	Turn On Out Of Money Light	Out Of Money
Out Of Money	Money Refill	Accept Another Dollar	Wait For Dollar

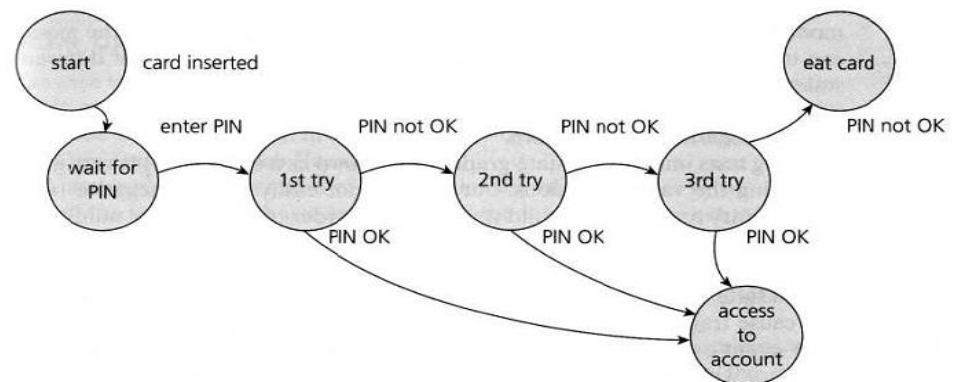
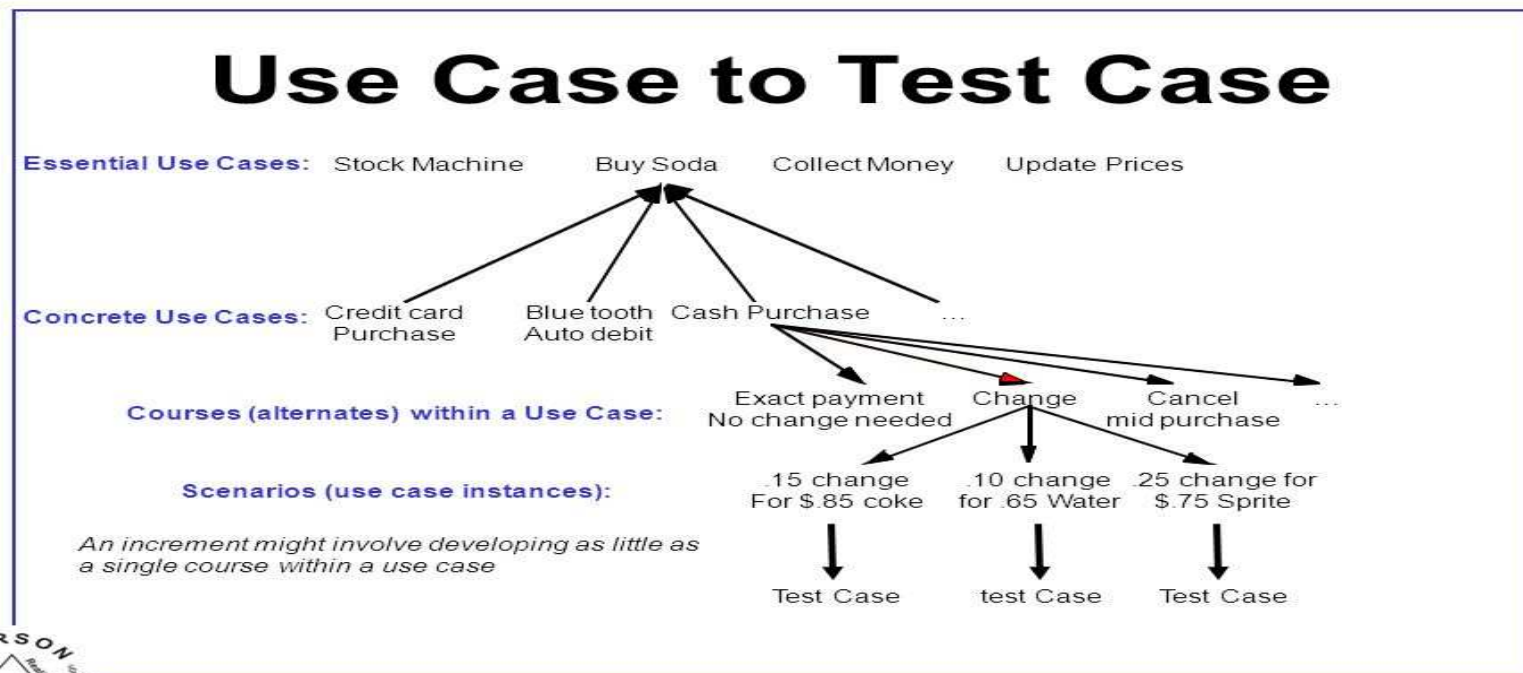


FIGURE 4.2 State diagram for PIN entry

Használati eset teszt

- Olyan feketedoboz teszttervezési technika, amelyekben a műszaki tesztterveket (test design) különböző használati eset forgatókönyvek futtatására készítették. → *use case testing*





Egyéb tesztelési technikák

- Alapvetően funkcionálisak
- Tapasztalat-alapúak
 - ☐ Sajátos érték alapú tesztelés
 - ☐ Random tesztelés
 - ☐ Felderítő teszt
 - ☐ Hibasejtés
 - ☐ Hibatámadás
 - ☐ Ellenőrző lista alapú teszt

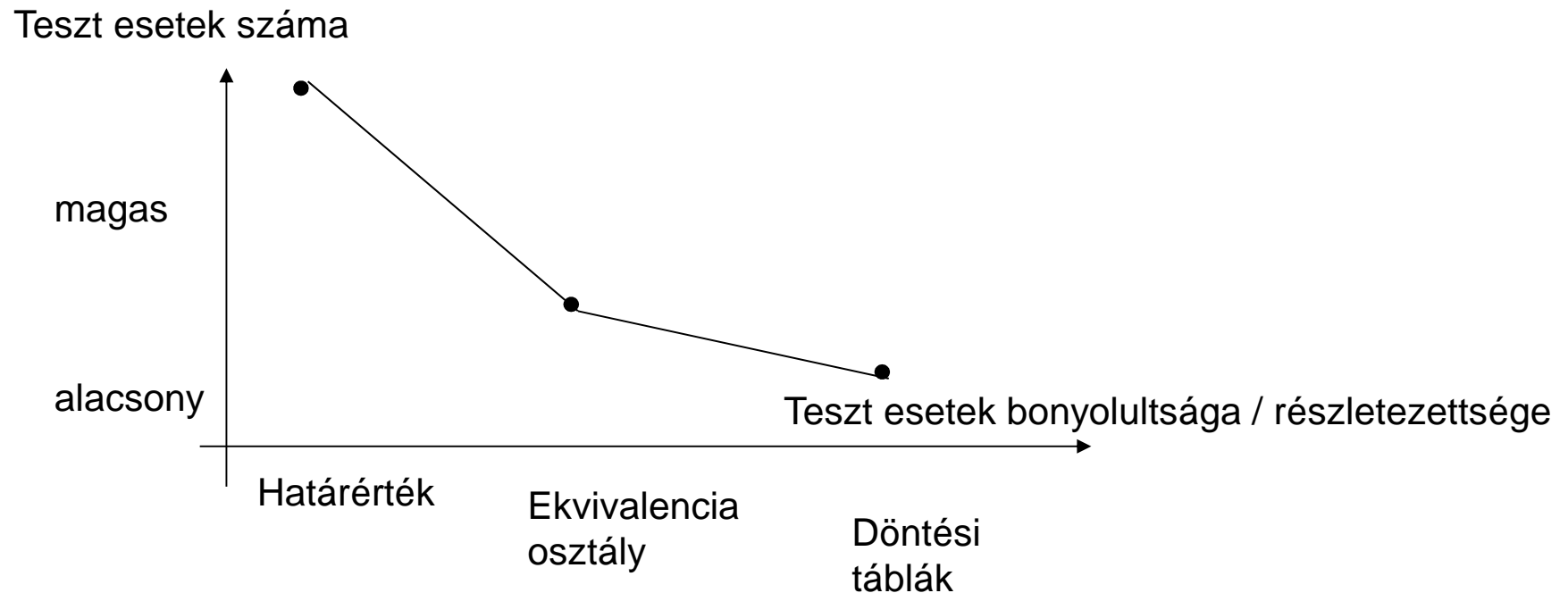


Funkcionális tesztelés - összefoglalás

- A program: egy matematikai függvény, amely az inputokat outputoknak felelteti meg
- A funkcionális tesztelés az input adatokból kiindulva vizsgálja az outputot
 - Határérték tesztelés: határérték analízis, robosztussági tesztelés, legrosszabb eset, robosztus legrosszabb eset
 - Ekvivalencia osztályok létrehozása, a teszt esetek számának csökkentésére: gyenge normál, gyenge robosztus, erős normál, erős robosztus
 - Döntési táblák: a változók közötti logikai összefüggéseket is figyelembe veszik

A funkcionális tesztelés ráfordítása

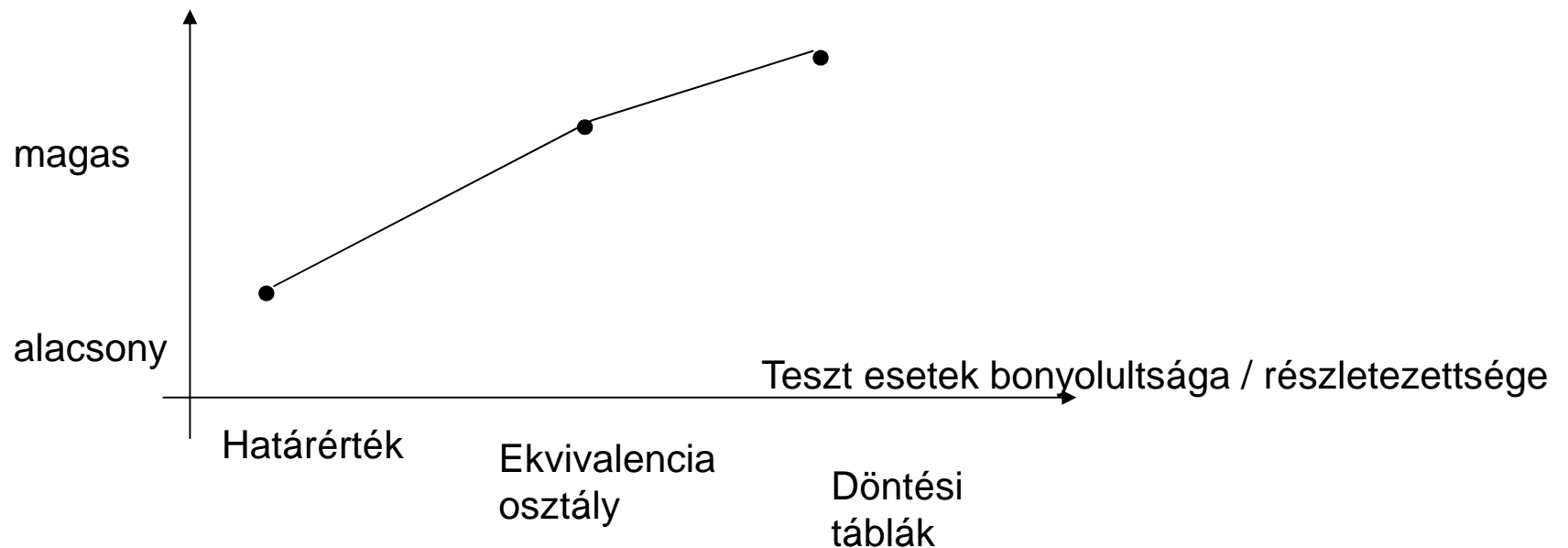
- Változó, mind a teszt esetek számát, mind az esetek kidolgozásához szükséges ráfordítást tekintve



A funkcionális tesztelés ráfordítása

■ Ráfordítás, technikánként

A teszt esetek kidolgozásának ráfordítása





Funkcionális és nem-funkcionális követelmények tesztelése

- Egy rendszer funkcionális követelményei azt írják le, amit a rendszernek csinálnia kell (a funkciókat).
 - Különböző formában dokumentálhatók. A „funkció” az, **amit** a rendszer csinál.
- **A funkcionális tesztek** a funkciókra alapoznak, és ezek megfelelő működését tesztelik.



Funkcionális és nem-funkcionális követelmények tesztelése

- **Nem-funkcionális követelmény:** olyan követelmény, amely a funkcionalitáshoz nem, de a megbízhatósághoz, hatékonysághoz, használhatósághoz, karbantarthatósághoz és hordozhatósághoz kapcsolódik. → *non-functional requirement*
 - A nem-funkcionális követelmények tesztelésében segít pl. az **ISO/IEC 25010** szabvány szoftvertermékre vonatkozó minőségi modellje.
- **Nem-funkcionális teszt:** egy komponens vagy rendszer funkcionalitáshoz nem kapcsolódó tulajdonságainak tesztelése, mint például megbízhatóság, hatékonyság, használhatóság, karbantarthatóság és hordozhatóság. → *non-functional testing*



Strukturális tesztelés

- Más néven: **fehértoboz teszt**
- A szoftver belső struktúrájának elemzésén alapuló tesztelés → *white-box testing, glass box testing, clear-box testing*
- A belső struktúrára vonatkozó információk a kódból, architektúrából, modellekből nyerhetők.
- A strukturális (fehértoboz) teszt minden tesztszinten végrehajtható. A strukturális technikák legjobban a specifikáció alapú technikák után használhatók annak érdekében, hogy egy adott típusú struktúra lefedettségének elemzésével támogassák a teszt lefedettségének mérését.
- A **lefedettség** azt mutatja meg, hogy egy tesztkészlet milyen mértékben hívott meg egy struktúrát, és ezt az értéket a lefedett elemek százalékában fejezik ki.
- Ha a lefedettség nem 100%, további tesztek tervezhetnek, hogy a kimaradt elemeket is teszteljék, ezzel növelve a lefedettséget.



Strukturális tesztelés

- A forráskódon alapszik
- „Abszolút” alapokon nyugszik
- Technikáiban pontos definíciókat, matematikai analízist alkalmaz
 - Az alkalmazott technikák már a '70-es években léteztek
- Egészen pontos méréseket tesz lehetővé



Strukturális tesztelés

- **Utasítás szintű teszt és lefedettség**
 - Az utasítás szintű lefedettség annak értékelése, hogy valamely teszteset készlet a futtatható utasítások hány százalékát érintette. Az utasítás szintű tesztnél a származtatott tesztesetek meghatározott utasításokat hajtanak végre, általában az utasítás lefedettség növelése érdekében.
- **Döntési teszt és lefedettség**
 - Annak értékelése, hogy valamely teszteset készlet a döntési eredmények (pl. egy If utasítás Igaz vagy Hamis lehetőségei) hány százalékát hajtotta végre. A döntési tesztnél a származtatott tesztesetek speciális döntési eredményeket hajtanak végre, általában a döntési lefedettség növelése érdekében.



Strukturális tesztelés

- A **döntési lefedettség**et a megtervezett, illetve végrehajtott döntési eredmények száma és a tesztelendő kódban található összes lehetséges döntési eredmény hányadosa határozza meg.
- A döntési teszt a vezérlési folyam tesztelés egy formája - a döntési pontokkal egy sajátos vezérlési folyamatot hoz létre.
- A döntési lefedettség magasabb rendű az utasítás-lefedettségénél: 100%-os döntési lefedettség esetén garantált a 100%-os utasítás-lefedettség, aminek fordítottja nem igaz.

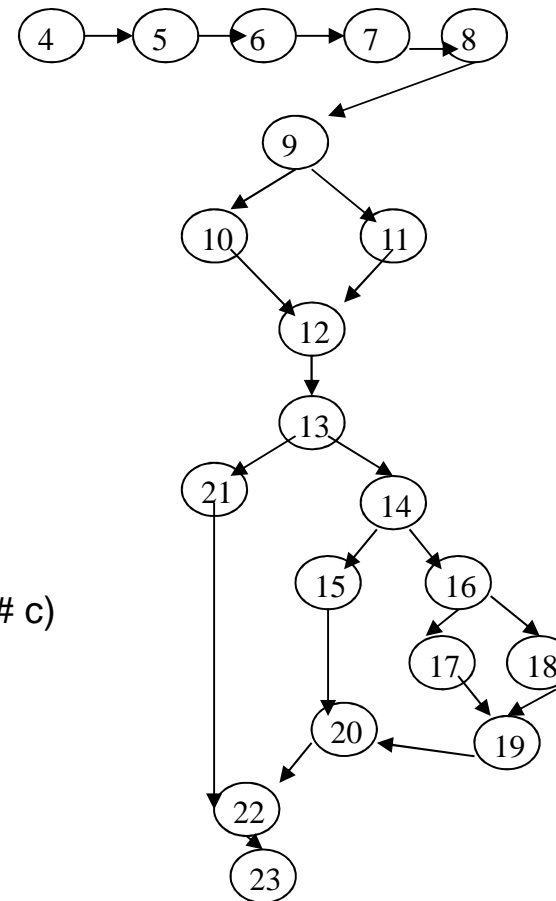


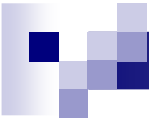
Útvonal alapú tesztelés

- Fehérdoboz teszttervezési technika, amely során a teszteseteket úgy tervezzük, hogy egy-egy végrehajtási utat járjanak be. → *path testing*
- A programnak megfeleltetünk egy programgráfot
 - Ha i és j csomópontok a gráfban, akkor i -ből j -be akkor és csak akkor vezet él, ha a j csomópontnak megfelelő utasítást vagy utasítás részletet (alapblokkot) közvetlenül az i csomópontnak megfelelő utasítás vagy utasítás részlet után lehet végrehajtani
 - Az ilyen gráfot CFG-nak (Control Flow Graph, vezérlésfüggés gráfja) nevezzük

Példa

```
1 Program triangle
2 Dim a,b,c, As Integer
3 Dim IsA Triangle As Boolean
  'step1: get input
4 Output („Enter 3 variables”
5 Input (a,b,c)
6 Output („side A:",a)
7 Output („side B", b)
8 Output („side C", c)
  'step2: Is A Traiangle?
9 If (a <b+c) And (b <a+c) And (c<a+b)
10 then IsATriangle = True
11 else IsATriangle = False
12 Endif
'Step 3: Determine triangle type
13 If IsATriangle
14   Then If (a=b) And (b=c)
15     Then Output („Equilateral”)
16     Else If (a # b) And (b # c) And (a # c)
17       Then Output („Scalene”)
18       Else Output („Isosceles”)
19     Endif
20   Endif
21 Else Output („Not a triangle”)
22 Endif
23 End triangle
```

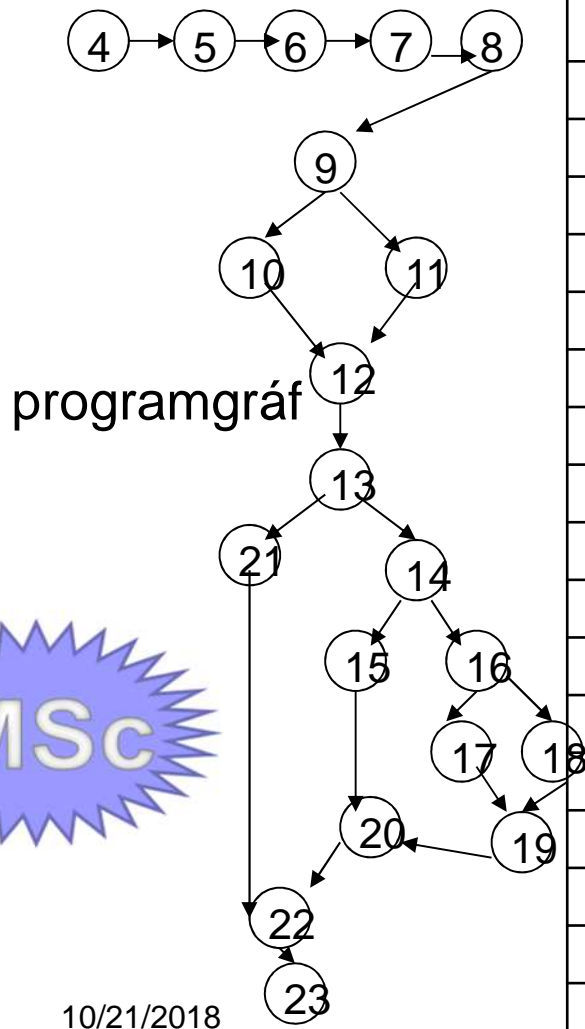




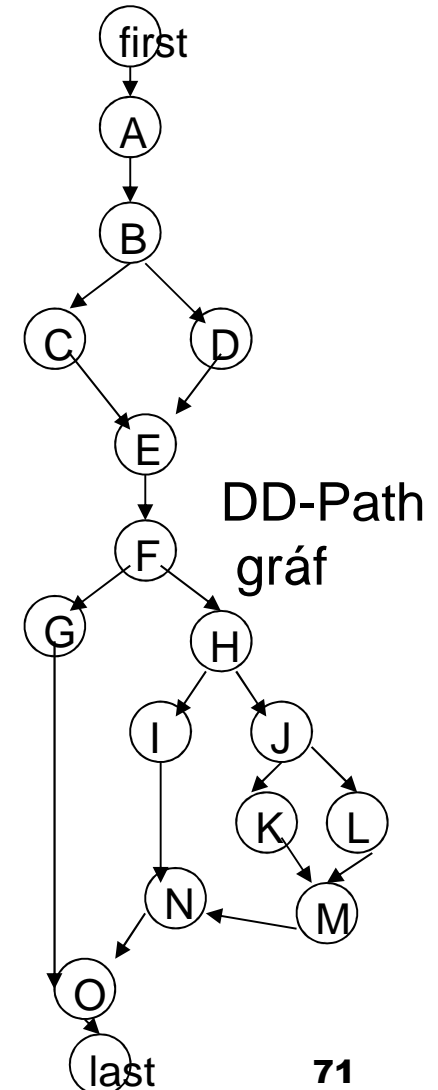
DD-Path (Decision to decision Path, döntés-döntés útvonal)

- Miller, 1977
- Szekvenciális utasítás-sorozat, nincs benne elágazás.
- Ha egyik utasítás hibás, az összes többi sem fog jól végrehajtódni
- DD-Path: olyan út / lánc, amelyben az első és az utolsó csomópont különböző, és minden belső csomópont esetében $\text{indeg} = 1$ és $\text{outdeg} = 1$
 - A kezdő csomópont 2 –szeresen összefüggő a lánc összes egyéb csomópontjával, és nincsenek 1-szeresen és 3-szorosan összefüggő csomópontok
 - A 0 hosszúságú lánc is DD- Path-nak tekinthető (egyetlen csomópont, élei nincsenek)

DD-Path (Decision to decision Path, döntés-döntés útvonal)



Program gráf csomópontja	DD-Path neve	Definíciós eset
4	first	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Balla K. last	2





Változtatáshoz kapcsolódó tesztelés

- Bármikor, ha változtatunk a rendszerben
- **Regressziós tesztelés:**
 - Egy korábban már letesztelt program, módosítást követő tesztelése, annak biztosítása érdekében, hogy a módosulás nem okozott hibát a szoftver nem módosított részeiben. A teszt végrehajtása a szoftver vagy a szoftverkörnyezet változtatásakor történik. (HTB)
- **Progressziós tesztelés:** feltételezzük, hogy az integrációs teszt rendben lefutott, és az új funkciókat tesztelhetjük



Karbantartási teszt

- Módosítások vagy megváltozott környezet miatt a működő rendszeren végrehajtott teszt.
→ *maintenance testing*
- A módosítások lehetnek
 - **tervezett** kiegészítő változtatások (pl. a kiadáshoz kapcsolódóan),
 - javító vagy **vészhelyzeti változtatások**, a környezet változása, mint pl. az operációs rendszer vagy adatbázis frissítése, az operációs rendszer újonnan kialakult vagy nemrég felfedezett sebezhető pontjainak védelme.
- A migrációra (pl. egy platformról egy másikra) vonatkozó karbantartási tesztnek tartalmaznia kell az új környezet, illetve a megváltoztatott szoftver működési tesztjeit.



Tesztmenedzsment

- A teszttevékenységek tervezése, becslése, monitorozása és irányítása, amelyet általában a tesztmenedzser végez.
 - Nagyon fontos a teszteléshez kapcsolódó adatok elemzése
 - Lásd még: Az alapvető tesztelési folyamat előző előadásban
 - Lásd még: Mérés és elemzés, 10. előadás



Mire jó a tesztmenedzsment?

- Például: a tesztelés hatékonyságának figyelésére



A tesztelés hatékonysága

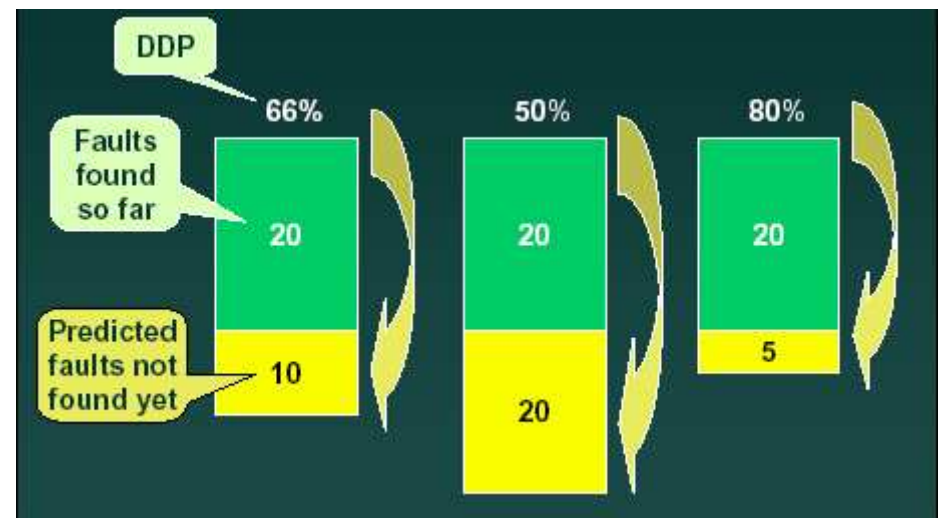
■ Hibamegtalálási százalék (DDP):

$$\frac{\text{Ebben a tesztelésben megtalált hibák száma}}{\text{Az összes hibák száma, beleértve azokat, amelyeket csak később találunk meg}} \times 100$$

- „Ez a tesztelés” lehet:
 - ☐ egy teszt-fázis (pl. egységteszt, integrációs teszt, átvételi teszt....)
 - ☐ egy funkcióra vagy alrendszerre vonatkozó összes teszt
 - ☐ egy rendszerre vonatkozó összes teszt

A tesztelés hatékonysága

- A még a rendszerben levő hibák számának előrejelzése
- Tudjuk, hogy ebben a fázisban pl. a DDP=66% szokott lenni
(ebben megtalált / (ebben megtalált + X)) x 100 = 66
 $X = (\text{ebben megtalált} \times 100) / 66 - \text{ebben megtalált}$
 $X1 = (20 \times 100) / 66 - 20 = 10$
 $X2 = (20 \times 100) / 50 - 20 = 20$
 $X3 = (20 \times 100) / 80 - 20 = 5$



(Source: Dorothy Graham: Measuring the value of testing. Escom, 2. April 2001. London



Tesztelés agilis környezetben

- Folyamatosan vannak rövid iterációk a tervezés, kódolás és tesztelés tevékenységekre
- A tesztelési tevékenységek is iteratív módon, folyamatosan kerülnek végrehajtásra



Agilis tesztelés

- Az agilis tesztelési gyakorlatok az agilis projektvégrehajtáshoz igazodnak: extreme programming (XP), a fejlesztést a tesztelés vevőjének tekintik, hangsúlyozzák a „test-first” tervezési paradigmát.



Tesztelés agilis környezetben

- Az agilis tesztelés **a lehető legkorábbi tesztelést** hangsúlyozza a szoftverfejlesztési életciklusban
- Megköveteli **a nagyon hangsúlyos vevői részvételt** a tesztelésben is, amint a kód hozzáférhetővé válik.
 - A kódnak elég stabilnak kell lennie ahhoz, hogy lehetővé tegye a rendszertesztelést
- **Fontos a regressziós** tesztelés.
- **A kommunikáció** a csapaton belül és a csapatok között kulcsfontosságú!

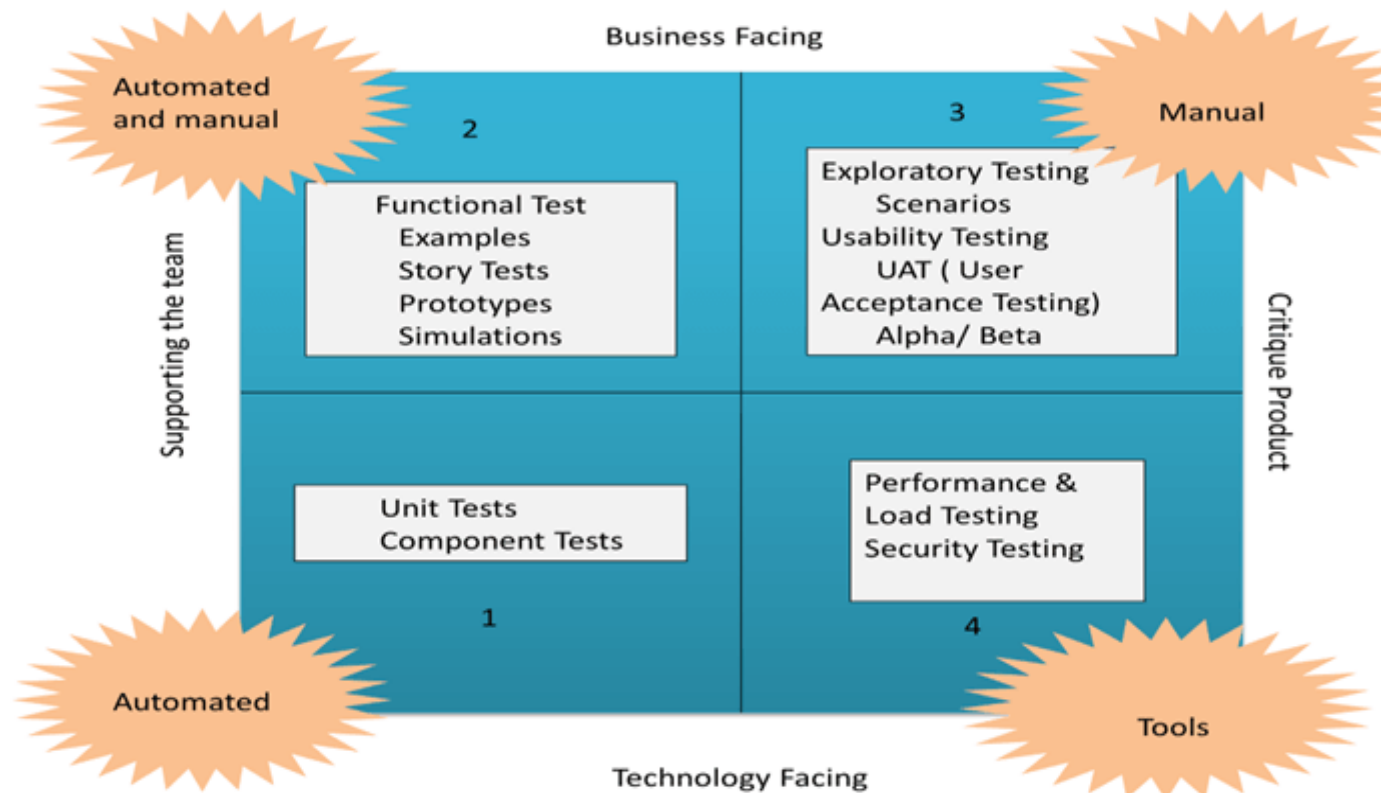


Néhány agilis tesztelési módszer

- Agilis tesztelési kvadránsok
- Tesztvezérelt fejlesztés
- Folyamatos integráció / „Continuous integration”

Lásd még: <http://www.opengov.hu/epitsunk-szolgaltatast/teszteles-agilis-kornyezetben.html>

Agilis tesztelési kvadránsok



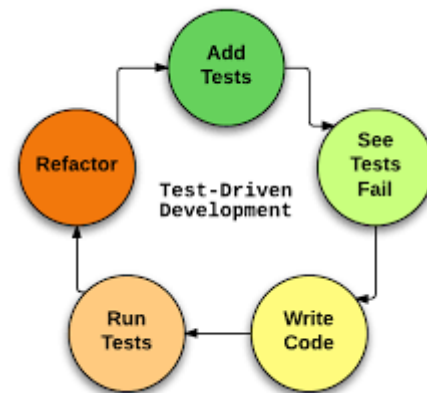
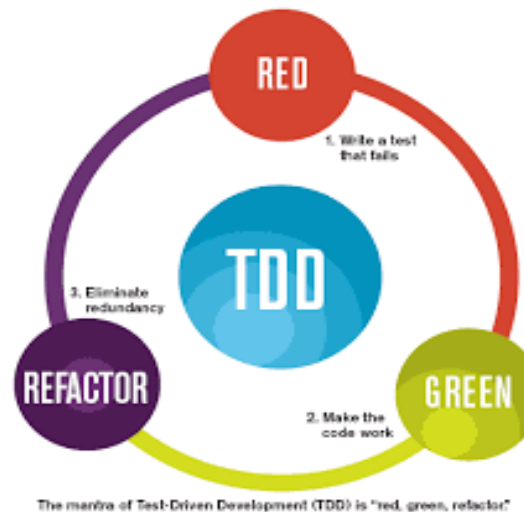
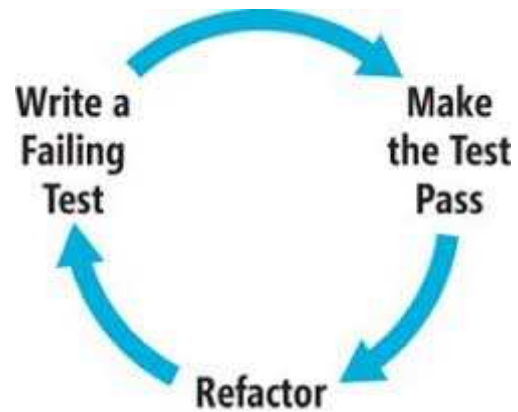
<https://www.guru99.com/agile-testing-a-beginner-s-guide.html>



Tesztvezérelt fejlesztés

- TDD – általában az extrém programozásban és az agilis fejlesztésben használják
 - A programozók először a tesztekét írják meg
 - A tesztek kezdetben sikertelenek lesznek
 - Rendre megírják a tesztekhez a kódot
 - A tesztekét kiegészítik

Test Driven Development





Continuous integration

- “A Continuous Integration – azaz a folyamatos integráció – egy szoftver fejlesztési módszer, melyben a fejlesztőcsapat tagjai az általuk írt kódot **legalább napi rendszerességgel** integrálják a korábbi fejlesztések közé, ez napi többszöri integrálást jelent.
- Minden új kód integrálása során **automatizált tesztek** ellenőrzik, hogy a rendszerbe való illesztés során okozott-e valamilyen hibát az új kódrészlet és ennek eredményeként a lehető leghamarabb visszajelzést ad az integráció eredményéről”

<https://martinfowler.com/articles/originalContinuousIntegration.html>

<https://realtester.wordpress.com/2012/09/30/continuous-integration/>



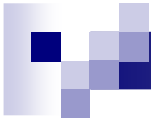
Continuous integration

- **Automatizált build és tesztelés** naponta történik; az integrációs hibák korán és gyorsan kiderülnek.
- Az agilis tesztelők automatizált tesztek futtatnak, és gyors visszajelzést adnak a csapatnak a kód minőségéről.
- Az eredményeket minden csapattag látja.



Az agilis tesztelők

- Korai fázistól kezdve dolgoznak
- A projekt teljes idején
- Átvételi / elfogadási kritériumok
 - Meghatározás – Segít az ügyfélnek
 - Automatizálás - Implementálja
- A tesztelőket bevonják a tervezésbe és főképp a következőkhöz adnak segítséget:
 - Tesztelhető user story-k definiálása, átvételi kritériumokkal
 - Részvétel a projekt és minőséggel kapcsolatos kockázatok elemzésében
 - A user story-khoz kapcsolódó tesztelési ráfordítás becslése
 - A szükséges teszt szintek meghatározás
 - A release-hez szükséges tesztek tervezése



Miről volt szó...

- Tesztelési technikák
- Statikus tesztelési technikák
- Dinamikus tesztelési technikák
- Tesztelés agilis környezetben