6. Tesztelés (Verification and Validation Testing)

"A tesztelés csak a hibák létét bizonyítja, de azok hiányát nem!"

Definitions:

Error: people makes error. Synonym: ”mistake”. When people makes mistakes while coding, we call this mistakes ”bugs”. (tévedés, tévesztés)

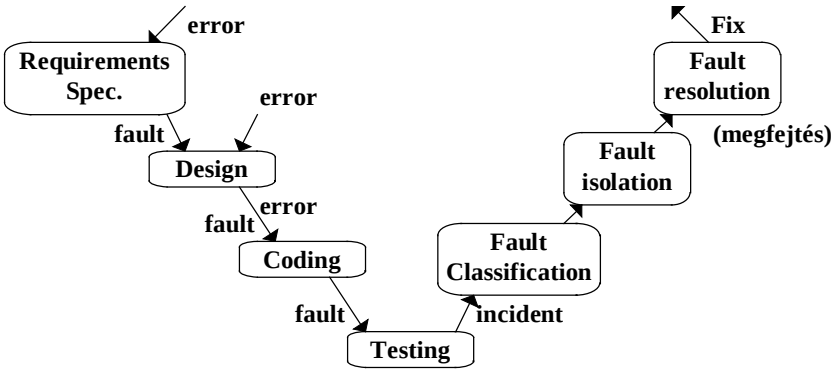
Fault: this is the result of an error. Synonym: ”defect” (bugs also good) Fault is a representation of an error. (hiba)

Failure: a failure occurs when a fault executes. (meghibásodás)

Incident: when a failure occurs, it may or may not be reading apparent to the user.

Test: testing is obviously concerned with errors, fault, failures and incidents. A test is the act of exercising with test cases. 2 goals: - to find failures
- to demonstrate correct execution

Test case: A test case has an identity, and is associated with a program behaviour. A test case also has
- set of inputs
- preconditions
- actual input
- list of expected outputs
- postconditions
- actual output

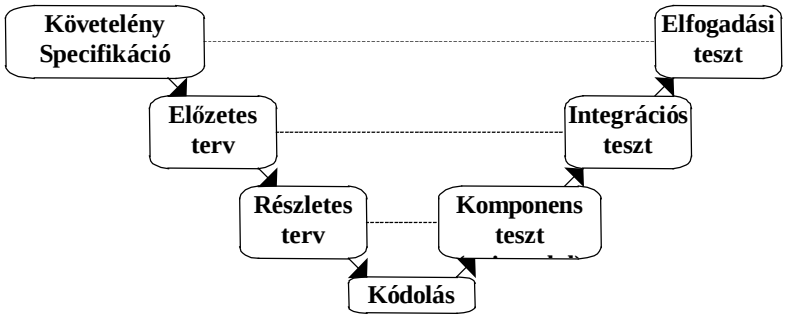


Testing life-cycle

Validation*: 'Are we building the right product'*
as defined by IEEE/ANSI, is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
azaz: 'A megfelelő terméket készítettük-e el?', tehát a fejlesztési ciklus végén elkészült sw a rendelő elvárásainak megfelel-e.

Verification: *'Are we building the product right?'*
as defined by IEEE/ANSI, is the process of evaluating a system or component to determine wheter the products of a given development phase satisfy the conditions imposed at the start of that phase.
azaz 'Megfelelően készítettük-e el a terméket?', tehát az elkészített program helyesen működik-e.. Ez mutatja, hogy a program megfelel-e a specifikációnak.

Testing = verification plus validation
A tesztelésnek kettős funkciója van:
- a programbeli hibák jelenlétét mutatja ki
- segít eldönteni, hogy a program a gyakorlatban használható-e vagy, nem



Az absztrakció és a tesztelés szintjei

Hibák súlyosság szerint osztályozva

1. Mild (enyhe)	Félrebetűzött szó
2. Moderate (mérsékelt)	Félrevezető, vagy redundáns információ
3. Annoying (bosszantó)	Megcsonkított nevek, 0.00$-os számla
4. Disturbing (zavaró)	Néhány tranzakció nem lett feldolgozva
5. Serious (komoly)	Elveszt egy tranzakciót
6. Very serious (nagyon komoly)	Helytelen tranzakció végrehajtás
7. Extreme	Nagyon komoly hibák gyakori előfordulása
8. Intolerable (nem tolerálható)	Adatbázis elromlás
9. Catastrophic	Rendszerleállás
10. Infectious	Rendszerleállás, amely továbbterjed más rendszerekre

A tesztelés szintjei:

Bemeneti/kimeneti hibák: – helyes bemenet nincs elfogadva

	– helytelen bemenet elfogadva – leírás rossz, vagy hiányzik – paraméterek rosszak, vagy hiányoznak – rossz kimeneti formátum – rossz kimenet az eredmény – helyes eredmény rossz idő alatt – nem teljes, vagy hiányzó eredmény – betűzés/nyelvtan
Logikai hiba:	– hiányzó eset(ek) – duplikált eset(ek) – extrém feltételek elhanyagolása – hiányzó feltételek – rossz változók tesztelése – rossz operátor
Számítási hiba:	– helytelen algoritmus – hiányzó műveletek, számítások – helytelen operandus – helytelen művelet – rossz beépített függvény
Interfész hiba:	– I/O időzítés – helytelen megszakítás-kezelés – rossz eljárás hívása – paraméter egyeztetés hibája(típus, szám) – inkompatibilis típusok
Adathiba	– helytelen inicializáció – helytelen tárolás/hozzáférés – rossz flag/ index érték – rossz változóhasználat – helytelen adatdimenzió – helytelen típus – inkonzisztens adat

A tesztelési folyamat részei:

Komponens teszt:

- **egységteszt (unit test):** az egységek, komponensek tesztelése, hogy megbizonyosodjunk a működésének helyességéről. Cél, hogy feltárja nincsenek-e tévműködések, feltáratlan hibák a belső algoritmusban, adatkezelésben. A komponensek más rendszer komponensektől függetlenül vannak tesztelve. /az implementáló feladata, felelőssége/
- **modulteszt (modul test):** egy modul (egymástól függő komponensek együttese), egy szinttel magasabb egység tesztelése. /ez még mindig az implementáló feladata, de ez már megjelenik a dokumentációban is/

Integrációs teszt:

- **alrendszer teszt (subsystem test):** egy alrendszer, azaz az alrendszerbe integrált modulok együttesének tesztelése. Az alrendszerben lévő modulok közti interface-hibák detektálása a fő cél. Az első alkalom, ahol már funkcionális kérdések is előkerülhetnek, ugyanis egy alrendszerhez, már funkció köthető. Itt már eldönthető, hogy a felhasználó által felmerülő kéréseket csinálja-e avagy nem.
- **rendszer teszt (system test):** az egyes alrendszerek közti kommunikációt, azaz a programstruktúrában egymáshoz kapcsolódó elemek együttműködését valamint az alrendszer rendszerbe való integrálását, kapcsolódását vizsgálja, hogy az megfelelő-e. Az együttműködési hibáknak sok oka lehet, pl információk, adatok elveszhetnek, vezérlőjelek elveszhetnek.

Az integrációs tesztet két különböző irányból célszerű elvégezni:

- **top-down:** a funkcióhiearchia legmagasabb szintjéből (root vagy main program) indul ki, majd végigmegy az összes lehetséges útvonalon, leellenőrizve annak működését. Ebben a megközelítésben a:
  - a vezérlő modul, mintegy test-driver működik, és ellenőrzi az összes alárendelt modul működését
  - az egyes almodulok ellenőrzése a már ellenőrzött helyettesítésével történik
  - a tesztelés folyamata a már beintegrált modulok alapján folyik
  - a tesztelés akkor fejeződik be pozitív eredménnyel, ha az összes útvonal minden modulja helyesen viselkedett
- **bottom-up:** a modul- együttműködési vizsgálatot alulról-felfelé, az elemi modulokból kiindulva végzi. A működési helyesség ellenőrzését az alábbi lépések szerint célszerű elvégezni:
  - az egymáshoz tartozó legalsó szintű modulokat önálló alfeladat végzésére képes egységekbe *cluster*ekbe kell integrálni
  - ellenőrizni kell az egyes modulok input/output információit, ehhez célszerű tesztprogramot alkalmazni
  - el kell végezni a clusterenkénti tesztet
  - el kell távolítani a tesztprogramot és egy szinttel feljebb az ellenőrzést meg kell ismételni egészen a legmagasabb vezérlési szintig.

Elfogadási teszt (acceptance test): A rendszer használatba helyezése előtti utolsó tesztelési lépcsőfok. A rendszert már a szimulációs tesztadatok helyett inkább valós adatokkal tesztelik. Ez a teszt feltárja a rendszer követelmény dokumentációjában lévő hibákat, valamint hiányosságokat, mert a valós adatokkal végzett gyakorlat mindig különbözik a tesztadatokkal végzett gyakorlattól. Ezen lépésben dokumentálják, a szerződésben foglaltaknak megfelel-e a program. Ennek viszonylag könnyen kiértékelhetőnek kell lennie. Ezt a tesztelést nevezik még α-tesztelésnek is.

Tesztelési stratégiák:

- β testing: az egész rendszer készen van, működik, és kiadják a potenciális felhasználóknak tesztelésre, akik a hibákat visszajelzik a fejlesztőknek.

- Stress testing: a rendszer hibás viselkedését teszteli. Azt nézi, hogy mi történik olyan körülmények közt amikor az események egy nemvárt kombinációja áll elő. Fontos, hogy ilyen körülmények között se legyen adatvesztés, adathibásodás.
- Regression testing: a rendszert a módosítások után újrateszteljük, és leellenőrizzük, hogy a változtatások után is kielégítően működik a rendszerünk.

Tesztelési technikák:

- dinamikus (időtől függő, az utasítások speciális sorozatának a futtatása szükségeltetik)
- statikus (időfüggetlen, és nem szükséges hozzá a tesztelendő program futtatása)

Statikus verifikálási technika

- a hibák futtatás előtti detektálása
- csak a program és a specifikációja közti kapcsolatot tudja ellenőrizni
- nem tudja demonstrálni, hogy a program működésileg helyes-e vagy nem
- a program kód vizsgálatán, és analízisén alapul
- hatékony a programhibák feltárásában (hibák 60 %a kiszűrhető)
- a formális specifikáció matematikai verifikációval a hibák 90%-át detektálhatja
- óránként 100 soros kódot lehet olvasni
- költség és időhatékony

Statikus verifikálási technikák:
- Program átnézés, vizsgálat (Program inspections)
- Matematikai alapú verifikáció (Matematically-based verification)
- Statikus program analizátor (Static program analysers)
- Cleanroom software development ("Tisztaszoba technika")

Inspection (Vizsgálat)

Az “inspection” egy formális, a leggyakrabban alkalmazott vizsgálati módszer, amelyet a szoftverfejlesztés minden fázisában lehet használni. Egy kis (legalább négy főből álló) csoport átnézi a követelmény-specifikációt, a részletes tervezési definíciókat, adatstruktúra terveket, teszttterveket, és a felhasználói dokumentációt. Ez egy rövid de nagyon hatékony vizsgálatot jelent, mert a terméknek csak egy kis része lehet az átvizsgálandó komponens.

Cél: a hibák, eltérések (defect) felderítése. Lehet ez pl logikai hiba, vagy anomália.

Ennek a módszernek nincsen tanítható struktúrája, formája, de vannak bizonyos feltételek, amelyeknek megfelelően kell végezni ezt az eljárást:
- a csoportnak a szervezeti szabványokat, szabályzatokat ismernie kell
- a csoportvezető követelmény, valamint a kód vizsgálat esetében a tervező, a terv esetében az implementáló,
- a vizsgálatot végző ember számára egy listát kell mellékelni a lehetségesen (esetlegesen) előforduló hibákról

- a vizsgálat eredményét projektmenedzsment szempontjából úgy kell tekinteni, mint a verifikációs eljárás egy lépését, és nem mint egy személyes sikert
- a programvizsgálathoz a programkód pontos definíciója szükséges
- programvizsgálat esetén csak teljesen kész, és szintaktikailag helyes kódot szabad vizsgálni (majdnem készet még nem szabad)

A vizsgálat után a csoportvezető összegzi a hibákat, problémákat, és minden résztvevőnek ad egy listát erről. A hiba kijavítása nem az ő feladata, így nem kell megoldási javaslatot sem tennie. A program mihelyt a tesztelő kezébe került, onnan már az ő felelőssége minden hiba ami a programban van, és ha találnak benne még hibát, akkor őt fogják felelősségre vonni, és nem a programírót.

Matematically-based verification

A formális program verifikáció matematikai módon bizonyítja, hogy a program konzisztens-e a specifikációban foglaltakkal.

A matematika-alapú formális specifikációnak két előfeltételt kell betartania:
- A programnyelv szemantikájának formálisan definiáltnak kell lennie.
- A programot formálisan olyan jelölésrendszerben kell megadni, ami konzisztens a használt matematikai verifikációs technikával

Mindamellett, hogy néhány munka formális nyelven van definiálva, a legtöbb nyelv szemantikája nincs formálisan definiálva. Ez azt jelenti, hogy legtöbb program esetében nem lehetséges a szigorú matematikai értelemben vett bizonyítás.

Azonban, a kevesebb matematikai alapú formális logikai bizonyítások használhatók arra, hogy növeljék a bizonyosságot a programnak a specifikációjához való alkalmazkodására. Ezek a bizonyítékok két dolgot demonstrálnak:
- A program kód logikailag konzisztens a program specifikációjával
- A program mindig befejeződik.

Statikus analizátorok (Static analysis tool)

A statikus program analizátorok olyan szoftvereszközök, amelyek a program forráskódját vizsgálják, és a lehetséges hibákat és anomáliákat detektálják, mint például a nemhasznált kódrészletek, vagy a neminicializált változók.
A program vizsgálat előtt használhatók a statikus analizátorok a kódban rejlő potenciális hibák felfedezésére. Ezek nagyon hasznosak pl a C nyelvnél, mert a C fordító ellenőrzési funkciója igencsak korlátolt. A programozói hibák automatikus detektálásához ezek az eszközök nagyon hasznosak
A statikus analizátorok segítségével detektálható hibaosztályok:

- Adathiba (Data faults):
  - változók inicializálás előtti használata
  - deklarált de sosem használt változó
  - nemdeklarált változó
- Kontrollhiba:
  - nemelérhető a kód
- Input/output hiba

- Interface-hiba
    - paraméter-típus hiányzik
    - paraméter száma hiányzik
    - nem meghívott függvények illetve eljárások
- Tárolókezelési hiba
    - pointer aritmetika

“Tisztaszoba” technika (Cleanroom software development)

A tisztaszoba szoftverfejlesztés a statikus verifikálási technikákon alapuló szoftverfejlesztési filozófia. Ezen technika célja egy hibamentes szoftver előállítása.
Ez a megközelítés azon alapul, hogy a hibákat inkább elkerülni kell, mint detektálni, és kijavítani. Ez egy szoftvertesztelést megelőző pontos, formalizált a hibák felfedézére irányuló vizsgálati eljáráson alapul.
A tisztaszoba fejlesztési technika jellegzetességei:

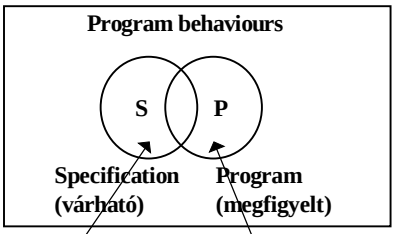
- Formális specifikáció
- Inkrementális fejlesztés: program növekmények külön, egymástól elszeparáltan vannak fejlesztve tisztaszoba technikával
- Strukturális programozás: a programfejlesztési eljárás a specifikáció lépésről lépésre történő finomítása
- Statikus verifikálás: matematikai alapú verifikálás
- Statisztikai tesztelés: a beépített programnövekményeket statisztikusan teszteljük

Nagy rendszerek fejlesztésében 3 csapat vesz részt:

- **Specifikációs csapat (Specification team)**: a rsz-specifikáció fejlesztéséért és nyomonkövetéséért ő a felelős. Ez a csapat készíti a felhasználó-központú specifikációt, valamint a verifikációhoz szükséges matematika specifikációt.
- **Fejlesztő csapat (Development team):** a szoftver fejlesztéséért és specifikálásáért ők a felelősek. A fejlesztési eljárás során a szoftvert nem futtatják, és nem is fordítják
- **Igazoló csapat (Certification team):** a statisztikus teszthez szükséges tesztesetek fejlesztéséért és ezek után a szoftver futtatásáért ők a felelősek. Ezek a tesztek formális specifikáción alapulnak. A teszteseteket a szoftver megbízhatóságának az igazolására használják. Ők döntenek arról, hogy mikor lehet a tesztelést befejezni.

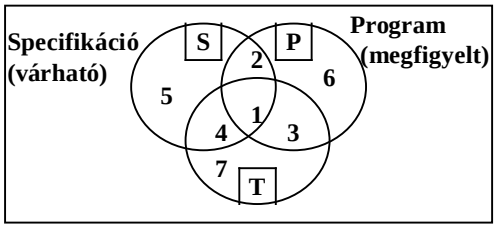
A nagy megbízhatóságú rendszerek fejlesztésekor ezt a technikát használják.

6.2. Dinamikus tesztelés (Dynamic Testing)



Fault of omission
(kihagyás)
(specifikálva volt csak nem lett leprogramozva)

- **Fault of omission**
  (megrendelői hiba)
- **a specifikáció után fellépő hibák teljesek voltak**
  (programozási hiba)

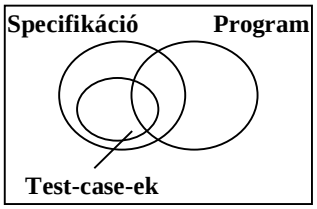


5-2: specifikált viselkedés, amire nincs test-case ® a tesztelés nem teljes

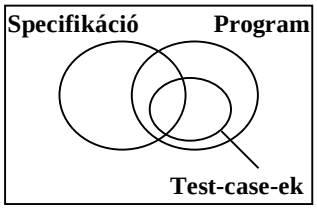
6-3-7: test-case-ek megegyeznek a nem specifikált viselkedéssel:

- **nem garantált a test-case**
- **hibás a specifikáció**

2 alapvető megközelítése van a ”test-case”-ek meghatározásának. Ezek funkcionális tesztelés és strukturális tesztelés néven ismertek. Mindkét megközelítésnek van számos különböző test-case meghatározó eljárása, amelyeket sokkal inkább tesztelési eljárásoknak nevezünk.



Funkcionális tesztelés



Strukturális tesztelés

<u>Strukturális tesztelés</u> (white-box testing):
A tesztelő a program belső struktúráját vizsgálja, azaz hogy miként is működik a program.

Ismert:
- Az implementáció és ezt a test-case-ek meghatározására használják.

Előny:
- A programozói hibák könnyen felismerhetők, mert az implementáció ismert.

Hátrány:
- A hiányos vagy hibás szoftverspecifikációt nem tudja felismerni.

A szoftverspecifikációban definiált, de nem implementált függvények hiányát nem képes detektálni.

Funkcionális tesztelés: (black-box testing)

Azon a szemléleten alapszik, hogy minden program egy-egy függvénynek tekinthető, amely a bemenő értelmezési tartományának értékeit a kimenő értékkészletének értékeire képezi le. Középpontban a program vagy a rendszer funkcionalitása van, és nem ismert, hogy a program az adott feladatot, hogyan is végzi el, tehát hogy a program vagy a rendszer hogyan is működik.

Ismert:
- bemenetek
- a várt kimenetek
- az egyetlen információ, melyet felhasználunk az a szoftver specifikációja.

Nem ismert:
- a fekete doboz tartalma (implementáció, hogyan is kódoltuk a programot).

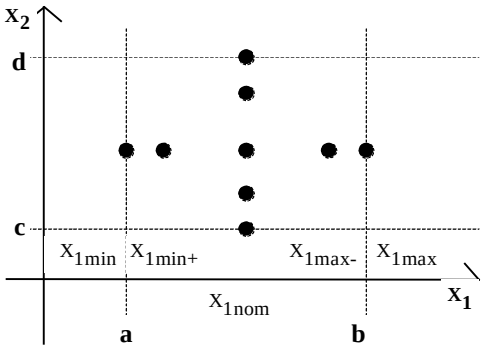
Előnyök:
- Független, hogy a szoftvert hogyan implementálták. (tehát az implementáció változhat)
- A test-case fejlesztés párhuzamosan következhet be az implementációval, ezáltal csökkentve a teljes projekt tervezési intervallumot.

Hátrányok:
- Nem teszteli a rejtett függvényeket, azaz azokat amelyek implementálva lettek, de a funkcionális specifikációban nem szerepelnek, ezáltal az ebből adódó hibákat sem detektálja.

Funkcionális tesztelés típusai:
- határérték vizsgálat
- ekvivalencia-osztály vizsgálat
- döntési tábla alapú vizsgálat

Határérték vizsgálat

Határérték analízis (Boundary value analysis): Ha az F függvény egy programként lett megvalósítva, akkor a bemenő $x_1;x_2$ változóknak lesznek korlátai: $a \le x_1 \le b$; $c \le x_2 \le d$

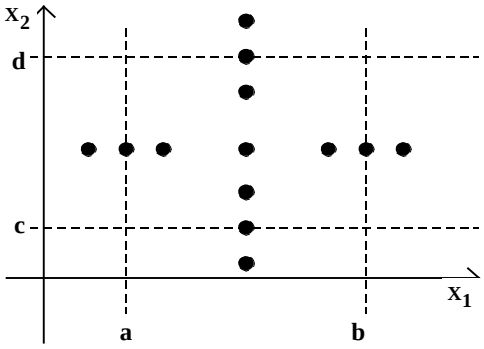


x_{1min}; x_{1min+}; x_{1nom}; x_{1max-}; x_{1max}

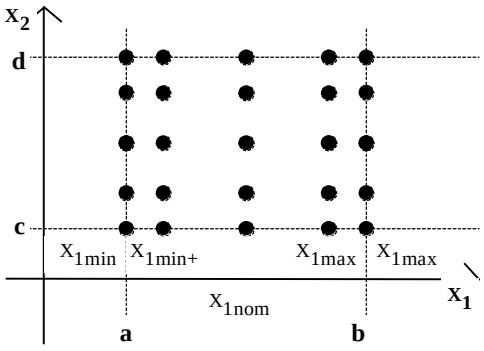
Pl.: 2 változó esetén ezek a test-case-ek:
$<x_{1min};x_{2nom}>$;$<x_{1min+};x_{2nom}>$;$<x_{1max-};x_{2nom}>$;$<x_{1max};x_{2nom}>$;$<x_{1nom};x_{2min}>$;
$<x_{1nom};x_{2min+}>$;$<x_{1nom};x_{2nom}>$;$<x_{1nom};x_{2max-}>$;$<x_{1nom};x_{2max}>$

Test case-ek száma (bemenetek száma = d) $N_d = 4d + 1$, ahol $d-1 \leq n \leq d$ (tehát a nominális változó értékének legalább d-1 változónál kell szerepelnie)

Robusztus tesztelés : $N_d = 6d + 1$
min-; min; min+; nom; max-; max; max+



Legrosszabb eset vizsgálat (Worst case):
$N_d = 5^d$



Robosztusság legrosszabb eset vizsgálat

$N_d = 7^d$

Ekvivalencia osztály tesztelés

- szeretnénk, ha közelítenék teljes vizsgálathoz
- egyazon időben azt remélnénk, hogy elkerüljük a redundanciát.

Ekvivalencia osztályok:
- egy halmaz egy partícióját képezik
- ahol ez a partíció a részhalmazok egy kölcsönösen diszjunkt összességére hivatkozik, melyek uniója egy teljes halmazt alkot.

teljes halmaz → a teljességet biztosítja
diszjunkt-ság → biztosítja a redundanciamentességet

Az ekvivalencia osztály tesztelés ötlete, hogy test-case-eket úgy határozzuk meg, hogy minden egyes ekvivalencia osztályból egy elemet használunk fel. Az ekvivalencia osztály tesztelés kulcsa az ekvivalencia relációk meghatározása
Tegyük fel, hogy programunk egy 3 változós (a, b, c) függvény, és az értelmezési tartománya az A, B és C halmazokból áll. Most tegyük fel, hogy választunk egy "megfelelő" ekvivalencia relációt, amely a következő partíciókat foglalja magába.

$A = A1 \cup A2 \cup A3$ $a_1 \in A1$

$B = B1 \cup B2 \cup B3 \cup B4$ a partíciók elemeit így jelöljük: $b_3 \in B3$

$C = C1 \cup C2$ $c_2 \in C2$

Gyenge ekvivalencia osztály tesztelés: egy test-case-ben egy változót használ minden egyes ekvivalencia osztályból

Test-case	a	b	c
WE1	**a1**	**b1**	**c1**
WE2	**a2**	**b2**	**c2**
WE3	**a3**	**b3**	**c1**
WE4	**a1**	**b4**	**c2**

Ez a test-case halmaz egy értéket használ minden egyes ekvivalencia osztályból. Mindig ugyanannyi gyenge ekvivalencia osztály test-case-ünk lesz, mint a legtöbb részhalmazzal rendelkező partícióbeli osztályok száma.

Erős ekvivalencia osztály tesztelés: A partícióbeli részhalmazok keresztszorzatán alapul. Az $A \times B \times C$ keresztszorzatnak $3 \times 4 \times 2 = 24$ eleme lesz (24 db. test-case)

Test case	a	b	c
SE1	**a1**	**b1**	**c1**
	a1	**b1**	**c2**
	a1	**b2**	**c1**
	a1	**b2**	**c2**
	a1	**b3**	**c1**
	:	**:**	**:**

A keresztszorzat biztosítja a teljességet:

– lefedjük az összes ekvivalencia osztályt
– a bemenetek összes kombinációja rendelkezésünkre áll

Pl.: Háromszög problémák

4 lehetséges kimenet: nem háromszög, egyenlőszárú, szabályos, általános. Ezeket alkalmazhatjuk a kimeneti ekvivalencia osztályok azonosítására:
R1 = {<a,b,c>: a 3szög a,b,c oldallal szabályos}
R2 = {<a,b,c>: a 3szög a,b,c oldallal egyenlőszárú}
R3 = {<a,b,c>: a 3szög a,b,c oldallal általános}
R4 = {<a,b,c>: a 3 oldal: a,b,c nem ad 3szöget}

Ezek az osztályok a test-case-ek egy egyszerű halmazát szolgáltatják:
weak (gyenge):

Test case	a	b	c	Várt kimenet
OE1	**5**	**5**	**5**	**szabályos**
OE2	**2**	**2**	**3**	**egyenlőszárú**
OE3	**3**	**4**	**5**	**általános**
OE4	**4**	**1**	**2**	**nem**

strong (erős): (pluszban jön a gyengéhez)

Test case	a	b	c	Várt kimenet
OE1	**2**	**3**	**2**	**egyenlőszárú**
OE2	**3**	**2**	**2**	**egyenlőszárú**
OE3	**4**	**2**	**1**	**nem**
OE4	**4**	**1**	**2**	**nem**
OE5	**1**	**4**	**2**	**nem**
OE6	**1**	**2**	**4**	**nem**
OE7	**2**	**1**	**4**	**nem**
OE8	**2**	**4**	**1**	**nem**

Ez esetben jobban megéri az outputokra meghatározni az ekvivalens osztályokat, mert az inputokra sokkal több lenne.

Döntési tábla alapú tesztelés:

A döntési táblák az összetett logikai relációk reprezentálására és analizálására szolgálnak. Ha bináris feltételeink vannak, akkor a döntési tábla feltételrésze egy igazságtábla. Ez a struktúra garantálja, hogy a feltételértékek összes lehetséges kombinációját megvizsgáljuk. Ha döntési táblákat használunk a test-case-ek meghatározására, akkor a döntési tábla teljességi tulajdonsága biztosítja a tesztelés teljességét.

Állapotok → input ; akciók → output

feltételek	szabályok									
c1: a,b,c egy háromszög	N	Y								
c2: a=b?				Y				N		
c3: a=c?			Y			N			Y			N	
c4: b=c?		Y	N		Y	N		Y	N		Y	N	

akciók	szabályok									
a1 nem háromszög	X									
a2 általános										X
a3 egyenlőszárú						X			X	X		
a4 szabályos			X									
a5 lehetetlen				X	X			X				

c1-c4: feltételek (conditions)
a1-a5: akciók

feltételek	szabályok										
c1 a < b + c	F	T	T	T	T	T	T	T	T	T	T
c2 b < a + c	-	F	T	T	T	T	T	T	T	T	T
c3 c < a + b	-	-	F	T	T	T	T	T	T	T	T
c4 a=b	-	-	-	T	T	T	F	F	F	T	F
c5 a=c	-	-	-	T	T	F	T	F	T	F	F
c6 b=c	-	-	-	T	F	T	T	T	F	F	F
a1 nem 3szög	X	X	X								
a2 általános											X
a3 egyenlőszárú								X	X	X	
a4 szabályos				X							
a5 lehetetlen					X	X	X				

ß

Test-case-ek

Test case	a	b	c	Várt kimenet
DT1	4	1	2	nem háromszög
DT2	1	4	2	nem háromszög
DT3	1	2	4	nem háromszög
DT4	5	5	5	szabályos
DT5	?	?	?	lehetetlen
DT6	?	?	?	lehetetlen
DT7	?	?	?	lehetetlen
DT8	3	2	2	egyenlőszárú
DT9	2	3	2	egyenlőszárú
DT10	2	2	3	egyenlőszárú
DT11	3	4	5	általános