Ellenőrző kérdések kidolgozva a Rendszertervezés (VIMIM238) tárgy Scherer Balázs által tartott részéhez

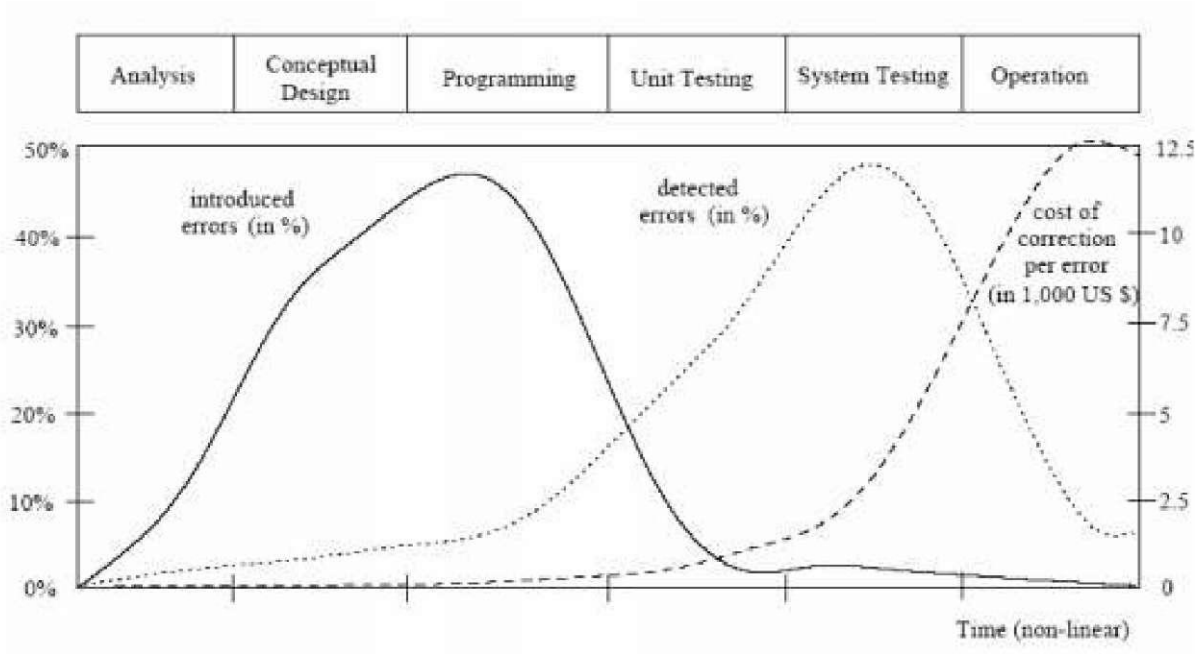
Készítette: Sipos-Takáts Bence
Forrás: Scherer Balázs

1. **Mutassa be, hogy hogyan alakult a *Standish Group* felmérése alapján a sikeresen végrehajtott projectek aránya az elmúlt években (arányokat írjon, pontos számok nem kellenek). Kommentálja a felmérés eredményeit!**

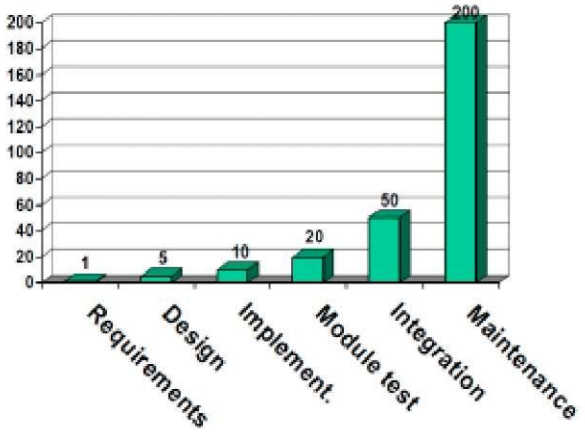
	1994	1996	1998	2000	2002	2004	2006	2009
Successful	16%	27%	26%	28%	34%	29%	35%	32%
Challenged	53%	33%	46%	49%	51%	53%	46%	44%
Failed	31%	40%	28%	23%	15%	18%	19%	24%

Jól látható, hogy az 1994-es évhez képest a 2000-es években már érzékelhető javuláson ment keresztül a szoftverfejlesztői piac. Az elmúlt évtizedben bár csak szinten maradt a sikeres projektek aránya, de ezt annak ellenére sikerült elérni, hogy a projectek komplexitása és mérete közben minimum megnégyszereződött. Ez így már nem rossz teljesítmény. El lehet képzelni, hogy mennyi lenne a sikeres projectek aránya ezen körülmények között, ha az 1994-es mérésnél használt módszereket és tool-okat alkalmaznák manapság is.

2. **Mutassa be, hogy statisztikailag a fejlesztési ciklus mely szakaszában kerül a rendszerbe a legtöbb hiba, azok hol fedeződnek fel és mennyibe kerül a javításuk? Mit segíthet ezen egy jól alkalmazott fejlesztési módszertan?**



From: P. Liggesmeyer et al., Qualitätssicherung Software-basierter technischer Systeme, Informatik Spektrum, 21:249-258, 1998. Quoted after J.P. Katoen, Principles of Model Checking, 2004/5. Copyright © by the authors.



Az oszlopdiagramos ábrából látható, hogy a legtöbb hiba a tervezés végső szakaszában és a programozás során kerül a rendszerbe, ezért a legtöbb fejlesztési módszertan úgy épül fel, hogy ezek a lépések már nagyon pontosan specifikáltak legyenek. Manapság a legtöbb esetben igyekeznek ezeket az utolsó lépéseket automatizálni. A tesztelés során a jól specifikált fejlesztési módszertanok abban tudnak segítséget nyújtani, hogy a szisztematikus tervezés következtében már a komponens szinten - tehát az ábránál a *Unit-test/Module-test* folyamatrészen - igyekeznek a legtöbb hibát megtalálni. Ezáltal az ábra *detected error* görbéjét mindinkább balra tolva csökkenthető a hibajavítás költsége. Egy nem megfelelő architektúrában írt szoftver korai fázisban végzett szisztematikus tesztelése például majdnem lehetetlen a bonyolultság és az egymásra hatások miatt (nem teljesül általában a jól szeparált komponensekre bontás és azok külön tesztelése). Az így a rendszerben benne maradó lappangó hibák csak később jönnek ki, ezzel jóval nagyobb anyagi és emberi erőforrás kárt okozva.

3. **A beágyazott rendszereket fejlesztő cégek miért törekednek az *ISO9001:2000* mellett más szabvány szerinti tanúsításra is. Milyen szabványok jöhetnek itt még szóba?**

A fejlesztési minőségbiztosításban leggyakrabban az ISO9001:2000, a CMMI és az ISO/IEC 15504 vagy más néven SPICE szabvánnyal lehet találkozni.
Az *ISO9001:2000* szabvány egy általánosabb bármilyen folyamatra alkalmazható minőségbiztosítási módszer, amely a termékfejlesztés folyamatának kialakításához, egy minőségügyi rendszer létrehozásához, az erőforrások, és a vevő megfelelő kezeléséhez nyújt keretet. Az ISO 9001-et általánossága miatt (akár egy ropis zacskón is találhatunk ISO9001-es hivatkozást) kifejezetten nehéz értelmezni a szoftver és hardver fejlesztési folyamatokra. Ahhoz, hogy egy ilyen alkalmazás egy cégnél elkészüljön igen tapasztalt rendszerfejlesztők kellenek. Megjelent ugyanakkor egy kiegészítés ISO90003:2000 néven, amely kifejezetten a szoftveralkalmazásokhoz nyújt útmutatást. Ez a kiegészítés sokat merít a CMMI-ból és az ISO-9126-ból. Másik szabvány még: SPICE.

4. **Röviden mutassa be a CMMI főbb jellegzetességeit és célját. Mi a különbség a *folytonos* és *lépcsőzetes* megközelítés között mikor melyiket célszerű használni?**

Capability *Maturity Model® Integration (CMMI)*

Egy cégnél a fejlesztés minősége az abban résztvevő emberek képzettségén, az alkalmazott eljárásokon és módszereken, valamint az ezek végrehajtásához használt eszközökön múlik. Ezeket a komponenseket a cégnél alkalmazott fejlesztési folyamat **(Process)** fogja össze. A CMMI célja, hogy a cégeknél alkalmazott folyamatok fejlesztésére és minősítésére adjon szabványt, több régebbi *Capability Maturity Model-t* integrálva. A CMMI alapvetően kétfajta megközelítést kínál a

folyamatok fejlesztésére a *folytonost (Continous Representation)* és a *lépcsőst (Staged Representation)* (mindkettőnek ugyanaz az eredménye csak az út más).

A *folytonos* megközelítés nagy szabadságfokot enged a fejleszteni kívánt CMMI folyamatok meghatározásánál. Ez a megközelítés elsősorban azoknak ajánlott, akik tisztában vannak a fejlesztési folyamatokkal és tudják, hogy min szeretnének javítani. Tehát a *folytonos* megközelítésében maga a modell alkalmazója dönti el, hogy melyik folyamatot kívánja vizsgálni. A folyamat kiválasztása után lehetősége van az adott folyamat *képességi* fokozatát megállapítani, és útmutatást kap arra vonatkozóan, hogy mit kell ahhoz tenni, hogy a következő *képességi* szintre jusson az. A kezdőknek elsősorban a *lépcsőzetes* megközelítés ajánlott, ahol a több évtizedes tapasztalatokból kiindulva konkrét lépcsőkre van bontva a folyamatok fejlesztése. Ebben az esetben nem külön az egyes folyamatokat vizsgálja a modell, hanem egyben az egész fejlesztési folyamatot, és azt adja meg, hogy az adott *érettségi szintek,* eléréséhez milyen folyamatokat kell. Tehát összefoglalva a *folytonos* megközelítés az egyes folyamatokat és azok *képességeit* vizsgálja és fejleszti külön-külön. A *lépcsőzetes* megközelítés pedig az egész szervezet *érettségét* vizsgálja és fejleszti. Természetesen a két megközelítés erős kölcsönhatásban van egymással, és léteznek megfelelőségek közöttük.

5. **Ismertesse a CMMI *lépcsőzetes* megközelítésének *érettségi* szintjeit!**

1. Initial (kezdeti)
A folyamatok ad-hoc jellegűek, és kaotikusak. A szervezet nem képes stabil környezet létrehozására a project számára. Az eredményesség elsősorban a dolgozók tudásán és hősiességén múlik. Bár az ezen az érettségi szinten álló szervezetek is általában működő dolgokat hoznak létre, de az esetek többségében túllépik az időkeretet, vagy az előre kalkulált költséget. Válsághelyzetben (gyakorlatilag mindig az van) a folyamatokat figyelmen kívül hagyják és képtelenek a sikerek megismétlésére.

2. Managed (menedzselt/irányított)
A szervezet biztosítja, hogy a projectjeiben a folyamatokat tervezik, ellenőrzik, mérik és végrehajtják. A menedzsment számára a project állása és a termék állapota az előre meghatározott pontokon világosan ellenőrizhető. A végtermék teljesíti a kitűzött célokat és megfelel az előre specifikált szabványoknak.

3. Defined (meghatározott)
Ezen a szinten a szervezetnek előre meghatározott szabványos folyamatai vannak, amelyeket folyamatosan fejlesztenek. A projectekre ezeket a szervezeti folyamatokat szabják testre a megfelelő útmutató alapján. Az egyes processzek pontosabban részletesebben leírtak, mint a 2. érettségi szinten. Ehhez a szinthez hozzátartozik a folyamatos oktatás és képzés is.

4. Quantitatively Managed (mennyiségileg menedzselt/irányított)
A folyamatok végrehajtására és a minőségére mutatókat, mérőszámokat határoznak meg. Ezeket a mérőszámokat gyűjtik az egyes projectek során, majd elemzik azokat. A fő különbség a 3. és a 4. érettségi szint között, hogy a 4. szinten a folyamatok teljesítménye jósolható az eddigi statisztikákból, tehát számszerű, mennyiségi becslés áll rendelkezésre.

5. Optimizing (optimalizáló)
A folyamatokat rendszeresen javítják, a mérések alapján felmérik az ingadozások okait, és korrigálják azokat. A folyamatok fejlesztési mutatóit szervezeti szinten határozzák meg, és folyamatosan hangolják az aktuális célokhoz.

6. **Ismertesse a CMMI *folytonos* megközelítésének *képességi* szintjeit!**

0. Incomplete (nem teljes, befejezetlen)
A folyamatot nem, vagy csak részben hajtják végre.

1. Perfomed (végrehajtott)
A folyamatot végrehajtják, teljesülnek a folyamat céljai.

2. Managed (menedzselt/irányított)
A menedzselt folyamatot előre tervezik, a végrehajtását megfelelő képességű emberek a megfelelő erőforrásokkal végzik. A folyamatot követik és ellenőrzik. Egy ilyen szintű folyamatnál biztosítva van, hogy stressz alatt is végrehajtják.

3. Defined (meghatározott)

Ezen a szinten az egyes folyamatok végrehajtási módja szervezeti szinten meghatározott. Az egyes projecteknél testre szabják a folyamatot. A *meghatározott* folyamatoknál sokkal kevesebb projectről projectre az eltérés, mint a 2. szintű *irányított* folyamatoknál, mert itt mindegyik egy közös szervezeti bázisból indul. A 3. szintű folyamatok pontosabban részletesebben is kerülnek meghatározásra.

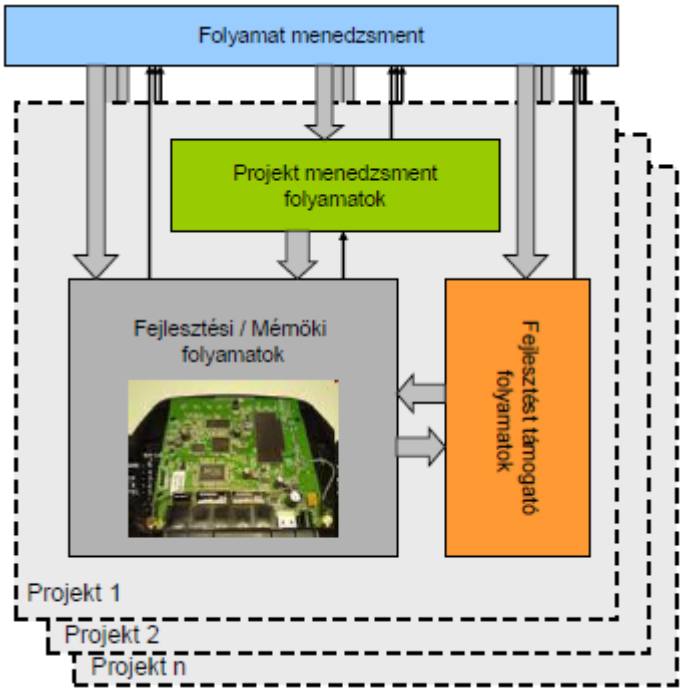
4. Quantitatively Managed (mennyiségileg menedzselt/irányított)

A folyamat végrehajtására mérőszámokat határoznak meg. Ezeket a mérőszámokat gyűjtik az egyes projectek során, majd elemzik azokat. A fő különbség a 3. és a 4. érettségi szint között, hogy a 4. szinten a folyamat teljesítménye jósolható, az eddigi statisztikákból.

5. Optimizing (optimalizáló)

A folyamatokat folyamatosan javítják. A mérések alapján felmérik az ingadozások okait, és korrigálják azokat. A folyamatok fejlesztési mutatóit szervezeti szinten határozzák meg, és folyamatosan hangolják az aktuális célokhoz.

7. **Milyen csoportokba sorolhatóak a CMMI folyamatai és ezek nagy csoportok milyen viszonyba vannak egymással?**



A folyamatcsoportok között próbáljuk jelezni az egymáshoz rendeltség viszonyát, a vastag nyíl szabályozást és megkötéseket, a vékonyabb nyíl adatszolgáltatást és visszacsatolást jelent.

- Engineering (mérnöki, fejlesztési)
- Project management (projektmenedzsment)
- Process management (folyamat menedzsment)
- Support (a fejlesztést támogató folyamatok)

8. **Soroljon fel legalább 3, a CMMI *Maturity level 2* eléréséhez szükséges folyamatot!**

Requirements Management (Követelmény menedzsment) ML2

Célja a project által előállított termékre vonatkozó követelmények elemzése és nyilvántartása az inkonzisztenciák kiszűrésére. A nem megvalósítható követelményeket fel kell fedni, és a megrendelővel egyeztetni kell a további sorsukról. A folyamat végzi el a követelménykövetést, az ún. *requirement tracking-et.*

Project Planning (Projecttervezés) ML2

Célja a projekt tevékenységeinek meghatározása és az ezek végrehajtásához szükséges tervek létrehozása. A tervezés magába foglalja munkatermékek és a feladatok megbecsülését, a szükséges idő és erőforrások meghatározását, a kötelezettségek egyeztetését, az ütemterv megállapítását, valamint a project kockázatainak azonosítását és elemzését.

Measurement and Analysis (Mérés és Analízis) ML2

Ezek a mérések elsősorban a menedzsment információs igényeinek kiszolgálására szolgálnak. Segítik a folyamatok tárgyilagos becslését és a teljesítmény követését. Céljuk mérendő mennyiségek azonosítása, és a metrika meghatározása. Szükséges továbbá az adatok tárolási módjának a meghatározása is.

Configuration Management (Konfigurációmenedzsment) ML2

A termék különféle verzióinak az előállításáért felelős, közük az alapverzióért és az abból származtatott speciális verziókéért. Nyilván kell tartania az egyes komponensek közül azokat a csoportokat, amelyek együtt működőképesek, beleértve a hozzájuk tartozó követelményeket és követelménykövetési dokumentációt. Szükség esetén biztosítania kell, hogy minden időpillanatban legyen egy működő konfiguráció.

9. **Soroljon fel legalább 3 olyan folyamatot, amelyek a CMMI *Maturity level 2* eléréséhez nem szükséges, a CMMI *Maturity level 3* -hoz ugyanakkor már igen!**

Technical Solution (Műszaki megoldás) ML3

Célja a termék megvalósítása. A lehetséges megoldások kiválasztása, a kiválasztás után tervkészítés, majd annak megvalósítása.

Product Integration (Termék Integráció) ML3

Az egyes termékek összeállítása rendszerré, a rendszer működésének ellenőrzése.

Verification (Verifikáció) ML3

Annak ellenőrzése, hogy helyesen terveztünk-e.

Validation (Validáció) ML3

Annak ellenőrzése, hogy jót terveztünk-e.

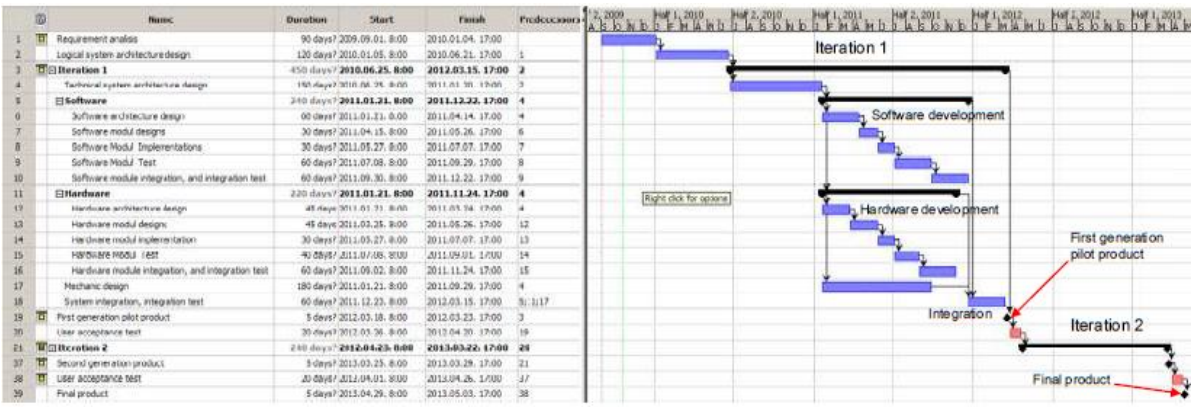
10. **Ismertesse a *Project menedzsment* folyamatokat lényegét összefoglalóan!**

- Becslések végzése
  - Végtermékek és feladatok becslése
  - A project életciklusának meghatározása
  - Ráfordítás és költség becslése
- Projectterv kialakítása
  - Költségek és ütemterv meghatározása
  - Kockázatok azonosítása
  - Erőforrások tervezése
  - Szükséges tudás és szakképzetség meghatározása.
  - Adatmenedzsment megtervezése
  - Projectterv dokumentálása
- Projectkövetés a terv alapján
  - Projecttervezési paraméterek követése
  - Kockázatok követése
  - Adatmenedzsment követése
  - Mérföldkő ellenőrzések.
- Javítás
  - Problémák elemzése
  - Javító intézkedések megtétele és menedzselése

11. **Mi az a Gantt diagramm? Milyen adatokat tartalmaz? Rajzoljon egy egyszerű fejlesztési folyamathoz illeszkedő Gantt diagrammot!**

A feladatok egymásutániságának és egymásra épüléseinek megjelenítésére legtöbbet használt eszköz talán az ún. *Gantt diagramm*, amelyet a legtöbb CAD rendszer támogat.
A sorokban az egyes feladatokat találjuk, az oszlopokban pedig a hónapokat. Egy feladat végrehajtási idejét egy-egy vízszintes vonal jelöli, míg az egyes folyamatok egymásra hatását nyilak jelzik. Egy *Gantt diagramm* létrehozásához a következő adatokat kell megadni:

- Folyamat neve
- A folyamat időigénye
- A folyamat legkorábbi lehetséges kezdési időpontja
- A folyamat függése más folyamatoktól



12. **Milyen módszereket használnak a project tervezés alatt a szükséges idő meghatározására?**

A projekttervezés alapvető problémája a szükséges erőforrások és idő megbecsülése. Erre a becslésre többfajta lehetőség van:

- Analógia más fejlesztési projektekkel
- Szakértői vélemény
- Cost modell alapú becslések

Szinte az összes magas CMMI minősítéssel rendelkező cégnek (Maturity Level 4 és felette) részletes statisztikai nyilvántartása van az eddig elkészült fejlesztéseiről és azok időigényéről. Azokban az esetekben, ahol ilyen előzetes adat nem áll rendelkezésre sokszor *Cost modell* alapú becsléseket adnak.

Egy ilyen *Cost modell* alapú becslésre egy klasszikus példa a COCOMO (Constructive Cost Model)**.**

13. **Mutassa be a COCOMO alapgondolatát (képletek nem kellenek)!**

(Ezek képletek – ebből kéne leszűrni a lényeget…)

Eff: Effort Applied (Szükséges erőforrás)

KLOC: Kilo Line of Code (1000 kódsor)

Dt: Development Time (Fejlesztési idő)

P: People required (Szükséges fejlesztők szám)

$$Eff = a\ (KLOC)^b \text{ [man months (ember hónap)]}$$

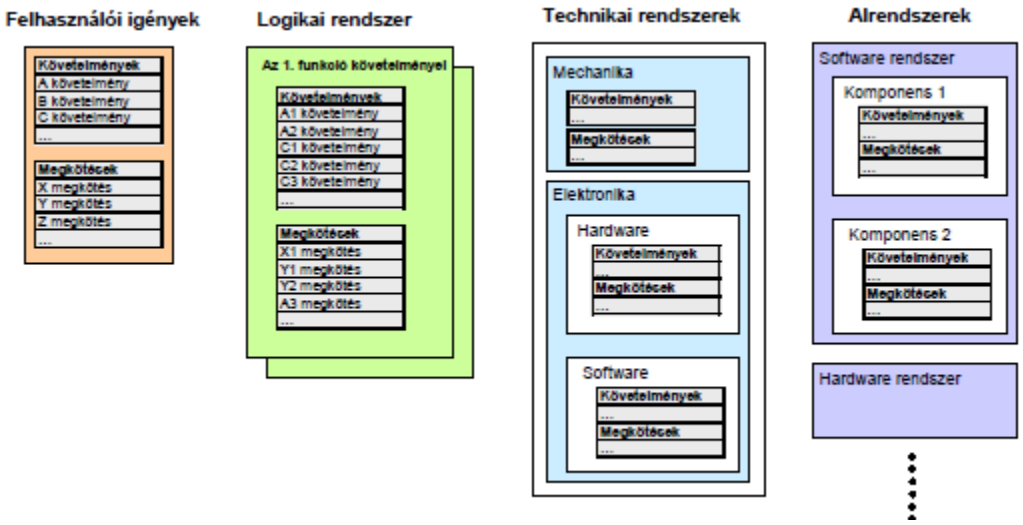
$$Dt = c\ (Eff)^d \text{ [months (hónap)]}$$

$$P = Eff / Dt \text{ [people (ember)]}$$

Az *a, b, c, d* konstansok project típus függőek.

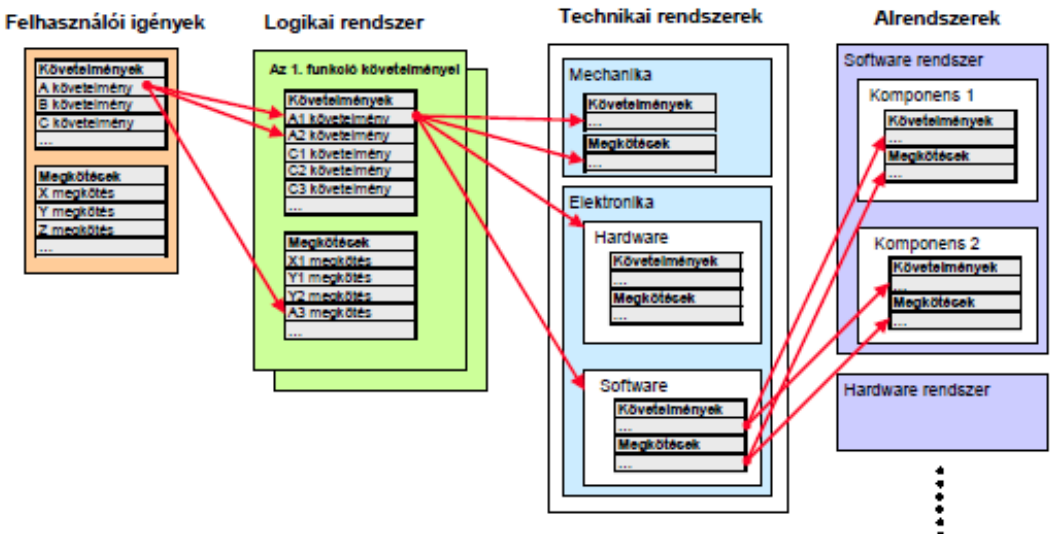
14. **Mutassa be a *Requirement Management* folyamatot!**

A követelmény menedzsment folyamat célja a projekt által előállított termékre vonatkozó követelmények elemzése, nyilvántartása és azok inkonzisztenciáinak kiszűrésére. A nem megvalósítható követelményeket fel kell fedni, és a megrendelővel egyeztetni kell a további sorsukról. A gyakorlatban a legtöbb cégnél egyszerű listaként, például Excel táblázatokban tartják számon a követelményeket. Az egyes alrendszerek, modulok követelményeinek nyilvántartása a tervezés minden fázisában fontos.



15. **Mi az a *Requirement tracking?* Mutasson rá példát!**

Mivel a világunk nem ideális, ezért a nyilvántartásnál azt is figyelembe kell venni, hogy a követelményeket a projekt elején nem fog sikerülni 100%-os lefedettséggel előállítani. Tehát sokszor szükség lehet arra, hogy fejlesztés közbeni követelményváltozásra reagáljunk. Ezért fokozottan fontos, hogy nyomonkövessük, egyrészt a felhasználói követelményrendszer változását, másrészt azt, hogy a felhasználói követelmények közül melyik hova lett beépítve a logikai és a technikai rendszer architektúrába. Ez a követelménykövetés, vagy az ún. *requirement tracking*, amelynek célja, hogy a követelmények és a termék funkciói, tulajdonságai közötti kétirányú megfeleltetést biztosítsa: egyrészt vissza kell tudni követni, hogy mely rendszerjellemzők milyen követelmények miatt lettek kialakítva, illetve tudni kell azt is, hogy az egyes követelménypontok hogyan képződtek le funkciókká, megkötésekké. Bár a *requirement tracking* egyszerű Excel munkalapok segítségével is elvégezhető, a cégeknél sokszor olyan tool-okat használnak, amelyek képesek a követelmények közötti kapcsolatot, valamint a követelmények, megkötések beépülését a termékbe követni. A talán legelterjedtebb ilyen tool a IBM Rational® DOORS®



16. **Soroljon fel legalább 8 követelmény osztályt beágyazott rendszerek esetére, amivel a *Requirement Management* folyamatnak foglalkoznia kell!**

- User Interface requirements (követelmények a felhasználói interfésszel szemben)
- Funcionality requirement (követelmények a működéssel szemben)
- Open-Loop, close Loop requirements (szabályozási követelmények)
- Real time requirement (valósidejűségi követelmények)
- Reliability requirements (megbízhatósági követelmények)
- Safety requirements (megbízhatósági követelmények).
- Instalation space, weight, current draw requirements (elfoglalt terület, súly, fogyasztás követelmények)
- Scalability and variant requirements (skálázhatóság, variáns)
- Quality Requirements (minó'ség követelmények)
- Cost, effort, time to market requirements (költség, fejlesztési munka, piacrakerülési idő követelményei)

17. **Mi a konfigurációmenedzsment folyamat célja, milyen lépéseket tartalmaz?**

A konfiguráció menedzsment a termék különféle verzióinak az előállításáért felelős, köztük az alapverzióért, és az abból származtatott speciális verziókéért. Nyilván kell tartania az egyes komponensek közül azokat a csoportokat, amelyek együtt működőképesek, beleértve a hozzájuk tartozó követelményeket és követelménykövetési dokumentációkat. A konfigurációmenedzsmentnek biztosítania kell tudni, hogy minden időpillanatban legyen egy működő konfiguráció. A konfigurációmenedzsment felelős továbbá azért is, hogy a vevőknek leszállított verziók konzisztensen archiválva legyenek, így egy esetleges panasz esetén vissza lehet keresni, hogy mi volt a probléma, és milyen fejlesztések történtek azóta. A konfiguráció menedzsment tehát nemcsak a fejlesztésnél és a termék kiszállításnál fontos, hanem a karbantartási fázisnál is.

Résztvevő eszközök azonosítása

A konfiguráció menedzsment fontos feladata, hogy azonosítsa azokat a termékeket, eszközöket, amelyek együtt egy konfigurációt alkotnak. Fontos, a konfiguráció menedzsment folyamat működése a fejlesztési iterációk alatt, hiszen egy termékből a végleges verzió előtt 3-4 fejlesztési verzió is készül. Ezek konfigurációit nyilván kell tartani, mert fontos alapot szolgáltatnak a következő verziókhoz, illetve nem ritkán folyamatosan használtak a fejlesztés során.

A konfigurációmenedzsment rendszer létrehozása

Gyakorlatilag az adatbázist, és a konfiguráció menedzsment procedúrát kell létrehozni, és az ezek kezelésére szükséges eszközöket felállítani. Létrehozza a konfiguráció menedzsmentben résztvevők hierarchiáját: Konfiguráció felügyelő, fejlesztő stb. Általában 3 tárolási helye, szintje szokott lenni egy konfiguráció menedzsmentnek.

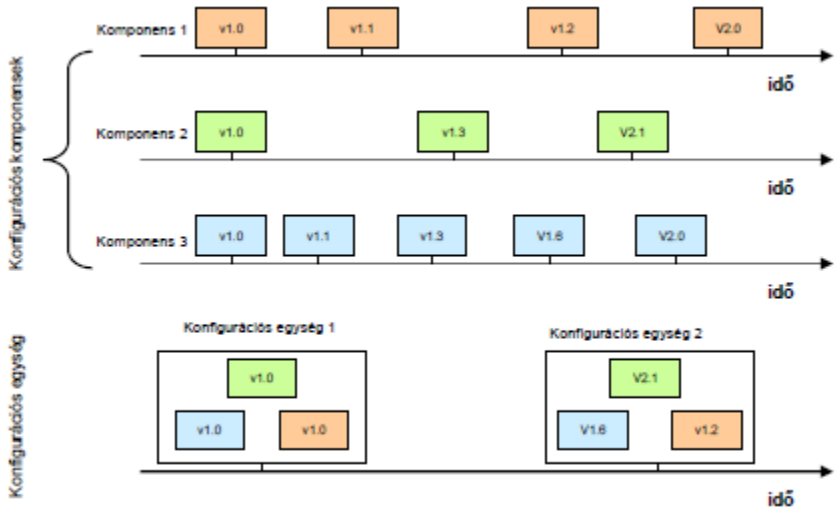
Változások követése, az integritás megtartása

A változások nyilvántartásánál, azt is követni kell, hogy ki, mit, mikor és miért változtatott. Továbbá a változások egész rendszerre gyakorolt hatását is fel kell mérni.

Releasek kibocsájtása.

A *release* egy olyan kombinációja az egyes konfiguráció menedzsmentben résztvevő komponenseknek, amelyek egy komplett konfigurációs egységet alkotnak. Konfigurációs egységeket és *release*-eket nem csak a teljes termékre lehet létrehozni, hanem részfolyamatokra is, például rendszer tervre, specifikációkra. Ezeket a konfigurációs egységeket általában az egész fejlesztési csapat, hagyja jóvá, tehát a megfelelő konfigurációs komponensek verzióinak kiválasztása nem egy ember dolga.

18. **Mik azok a konfigurációs komponensek és konfigurációs egységek (Mondjon példát mindkettőre)?**



2.4 ábra A konfigurációs komponenseket és konfigurációs egységek viszonya

19. **Mutassa be a konfigurációs egységek és konfigurációs komponensek viszonyát egy életszerű példán!** (Ugyanaz, mint 18)
20. **Mutassa be, hogy hogyan történik a konfigurációmenedzsmentben résztvevő komponensek azonosítása! Milyen szempontok vannak?**

A konfiguráció menedzsment fontos feladata, hogy azonosítsa azokat a termékeket, eszközöket, amelyek együtt egy konfigurációt alkotnak. Fontos, a konfiguráció menedzsment folyamat működése a fejlesztési iterációk alatt, hiszen egy termékből a végleges verzió előtt 3-4 fejlesztési verzió is készül. Ezek konfigurációit nyilván kell tartani, mert fontos alapot szolgáltatnak a következő verziókhoz, illetve nem ritkán folyamatosan használtak a fejlesztés során.
A konfiguráció menedzsmentben részvevő eszközök kiválasztása az alábbi szempontok szerint szokott történni:

- Termékek, amelyek kiszállításra kerülnek a felhasználónak
- Termékek, amelyek fontosak a belső munkafolyamatokhoz és változhatnak az időben.
- Eszközök, amelyek az egyes munkafolyamatokhoz kellenek

A konfiguráció menedzsment alá a következő termékeket, eszközöket szokás vonni:

- Követelmények és követelmény menedzsment dokumentumok

- A fejlesztési folyamatok terve
- Termék specifikációk, interfész leírások
- Programkód
- Termékre vonatkozó fejlesztési eredmények
- Tesztek, teszt tervek
- Fordító programok és fejlesztési környezetek

Fontos azt is meghatározni, hogy a fejlesztési folyamat melyik stádiumában és mikor kerüljön egy termék a konfiguráció menedzsment folyamat tevékenysége alá.

21. **A konfiguráció menedzsment rendszernek milyen tárolási helyei szoktak lenni?**

Általában 3 tárolási helye, szintje szokott lenni egy konfiguráció menedzsmentnek.

- Dinamikus: Helyben a fejlesztőnél tárolt verziók.
- Kontrolált, vagy központi tárolási pont: Központi tároló pontja a jelenlegi fejlesztésnek.
- Statikus, archívum: A már kibocsájtott *release-ek* archívuma.

22. **Mutassa be a Triviális verziókövetés („*Total Commander módszer")* előnyeit és hátrányait!**

A legtriviálisabb verziókövetés, hogy naponta egy külön könyvtárba bemásoljuk az aznapi munkánk eredményét. Ha nagyon kulturáltak akarunk lenni, akkor a project mellé mellékelünk egy *change* file-t is, amiben leírjuk, hogy mit és miért változtattunk Bár ez a módszer minden előismeret nélkül könnyen alkalmazható, az alkalmazás során számos probléma, kérdés felmerül:

- A másolatok komplett másolatok, tehát sok helyet igényelnek.
- Milyen gyakorisággal készítsünk másolatot?
- Csak működő verziót másoljunk fel, vagy köztes verziót is?
- Hogyan tudjuk követni, hogy min változtattunk?
- A *change* file-t nagyon pontosan kell nyilvántartani, különben inkább kártékony, mint hasznos, de nincs semmilyen automatizmus, ami erre ösztönözne.

Ezeknek a kérdéseknek, problémáknak egy része általános és nem csak a Total Commander- es megoldásra specifikus.

Ez a módszer meglepően hatékony, amikor egyedül dolgozunk, de szinte teljesen hasznavehetetlenné válik, nagyobb csoport alapú fejlesztésnél:

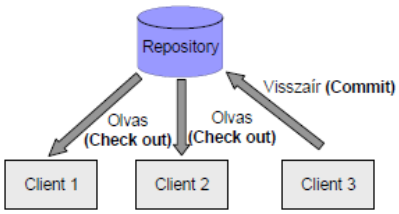
1. Változtatunk a működő alkalmazáson
2. De valaki más is beleír egy picit
3. Az alkalmazás lefagy

Az ilyen problémák orvoslására jöttek létre a verziókövető rendszerek. Ezek segítségével egységesen nyomon tudjuk követni, hogy ki mikor, mit és miért változtatott.

23. **Ismertesse a verziókövető rendszerekhez tartozó alapfogalmakat *(repository, working-copy, checkout, commit, update, merge)!***

- **Repository (raktár):** Központi nyilvántartás az adatoknak vagy projectnek (a mastercopy).
- **Client:** Felhasználó, aki dolgozni kíván a projecten.
- **Working copy:** Egy *Client* által a projectből létrehozott munkaváltozat, amit szabadon változtathat.

- **Check out:** olvasás
- **Commit:** visszaírás



- **Update:** saját változat frissítése (ha más mergelt és commit-telt
- **Merge**: A létrejövő konfliktusokat **Merge**-gel, tehát fuzionálással oldják fel, és így hoznak létre egy új verziót.

24. **Ismertesse a Lock-Modify-Unlock megközelítés és annak előnyeit, hátrányait!**

• Módosítás előtt le kell lock-olni egy file-t.

• Tehát egyszerre csak egy ember tudja módosítani a file-t. Olvasni tudja más is.

Lock-Modify-Unlock megközelítésnek a következő problémái vannak:

- Adminisztratív problémákhoz vezethet:
  - Ha egy fejlesztő elfelejt ki lock-olni egy file-t, akkor más nem férhet hozzá.
  - Ha szabadságra megy, akkor pl. rendszergazda kell a lock feloldásához.
- Felesleges egymásra várást okozhat.
  - Egy C file-on belül például valaki az F1 függvényt akarja módosítani, más valaki pedig az F2-t. Semmi köze a kettőnek egymáshoz mégsem tudják egyszerre megcsinálni.
- A biztonság hamis illúzióját keltheti.
  Például két fejlesztő dolgozik ugyanazon a projecten, az egyik lock-olja az A file-t, a másik a B file-t. A két file között függőség áll fent. Mindketten azt hiszik, hogy biztonságban vannak, holott mégsem

A Lock lehetősége ugyanúgy benne van a modern verziókövető rendszerekben, mint elődjeiknél. A Lock megközelítést kell használni például olyan bináris jellegű file-ok esetében, ahol a *merge* nem megoldható (hangfile-ok, bináris file- ok, NYÁK-tervek, kapcsolási rajzok, stb.)

25. **Ismertesse a Copy-Modify-Merge megközelítés, annak előnyeit, hátrányait!**

- Egyszerre több fejlesztő is ki „***Check*** **out**"-olhatja ugyanazt.
- Mindenki a saját ***Working copy****-ját* használja. ***Working copy:*** A Repository (vagy annak egy részének) saját gépen található leképezése.
- A létrejövő konfliktusokat pedig **Merge**-gel oldják fel, és így hoznak létre egy új verziót.
- A ***Merge,*** bár támogatva van a verziókövető rendszer által, alapvetően mégis emberi döntéseket követel, tehát nem automatikusan történik.

Copy-Modify-Merge megközelítés tulajdonságai összefoglalva a következők:

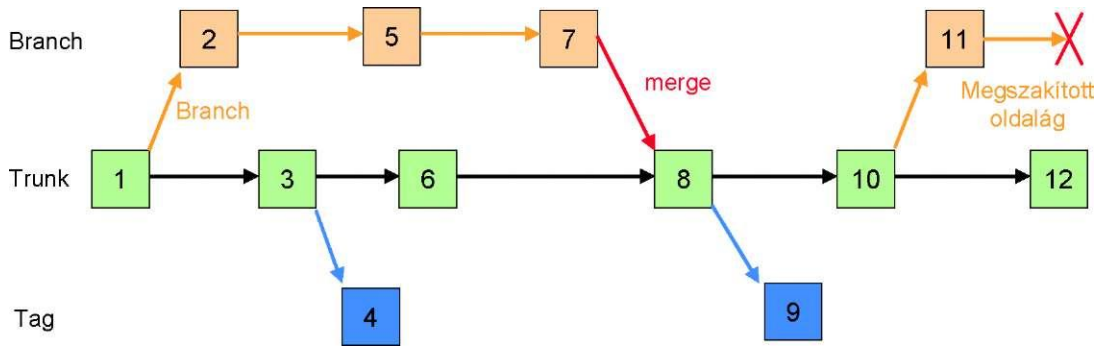
- Egyszerre több fejlesztő is dolgozhat ugyanazon a kódon.
- *Commit-nél* az esetleges konfliktusok kiderülnek.
- Embereknek kell döntenie a konfliktus feloldásáról.

Nagyon fontos azt hangsúlyozni, hogy a verziókövető rendszer nem helyettesíti az emberek közötti kommunikációt. Csak segít a tisztánlátásban és a munkafolyamatok követésében. A jó kollegiális viszonyra mindenképpen szükség van, anélkül semmi sem működik.

26. **Mutassa be az SVN-nél is alkalmazott Trunks, Tags, Brach ágak szerepét, egy példán keresztül.**

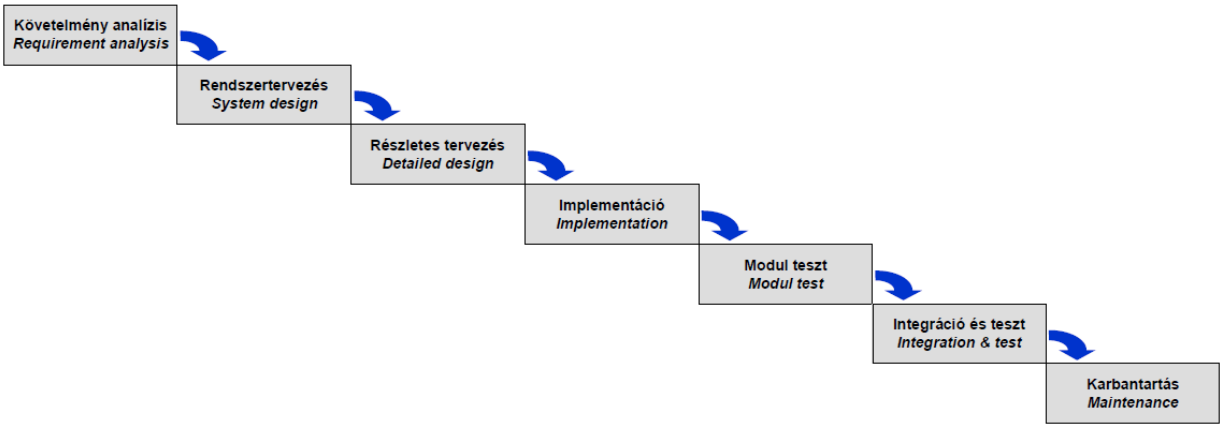
Egy komplex project verziókövetése során nem csak az egyes file-ok verzióit, hanem az egész project konfigurációját nyomon kell követni. Egy project esetében három ehhez kapcsolódó fogalmat kell megjegyezni:

- **Trunks:** Ez az ág tartalmazza a project törzsét, tehát a hivatalos fejlesztési irányt.
- **Branch:** Az elágazások a project fejlesztési irányától kicsit eltérő funkciók kipróbálására, megvalósítására szolgálnak. A lényege az, hogy ilyenkor a fejlesztés eredeti menete ne bonyolódjon, hanem egy saját leágazást lehessen nyúzni. Az esetek többségében az elágazások visszaintegrálódnak az eredeti projectbe.
- **Tags:** A címkézett ág felel meg az egyes stabil és működőképes release-eknek. Egy project fejlesztése során időről időre szoktak ilyen release verziókat létrehozni. Ez azért fontos, mert a Trunk ág folyamatos fejlesztésben van, és így általában nem tartalmaz komplett és jól működő verziót. Van úgy, hogy a Trunk ág éppen el sem tud indulni, vagy nem is fordul le.



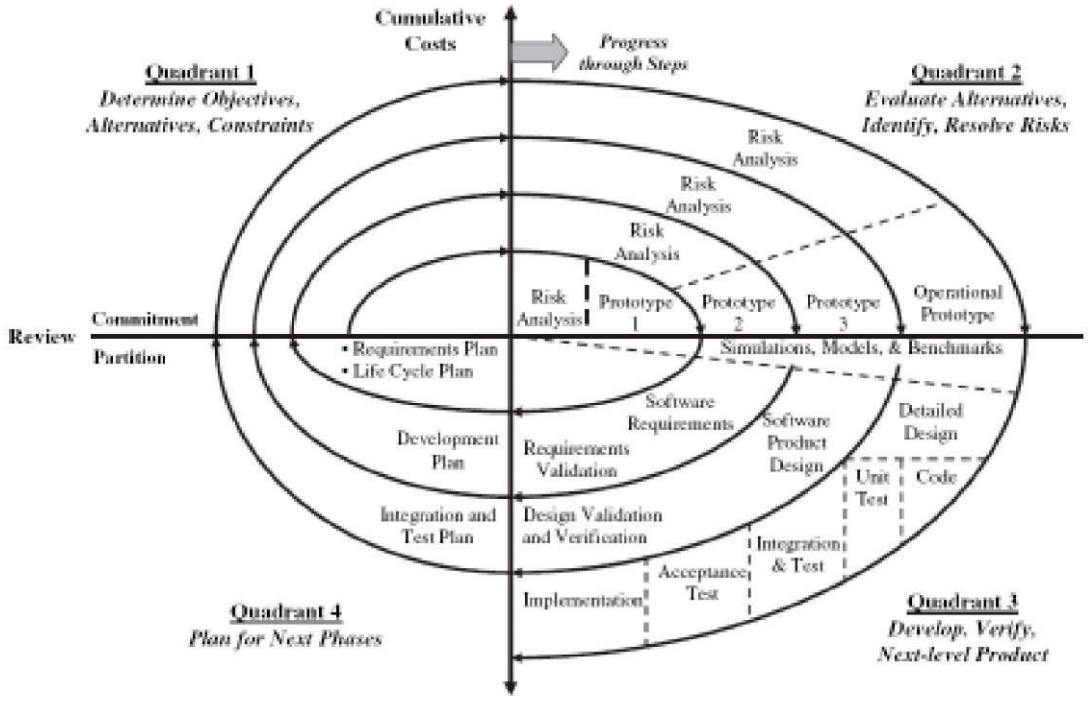
Egy project verziókövetési életciklusa

27. **Mutassa be röviden a *Vízesés* életciklus modellt! Ismertesse annak tulajdonságait!**



A *vízesés modell* gyakorlatilag formába önti az egyes életciklus lépéseket. A lépések formalizáltak, sorrendjük kötött. Csak az egyik lépés után hajtható végre a másik. Hátránya, hogy nem jól alkalmazható azokban az esetekben, ahol a követelmény nem előre jól megadott. A vízesés modell további hátránya, hogy nem jelzi a párhuzamot a tervezési és teszt lépések között, illetve nincs tisztázva a viszonya az iterációkkal.

28. **Mutassa be röviden a *Spirál* életciklus modellt! Ismertesse annak tulajdonságait!**



A Spirál modell egy erősen iteratív jellegű megközelítés. Az egyes fázisok után olyan prototípusok állnak rendelkezésre, amelyek a megrendelővel való egyeztetés alapján segítenek megalapozni a következő prototípust, majd végül a kész rendszert. A Spirál modell elsősorban ott használható, ahol a követelmények és kívánalmak nem állnak a projekt elején rendelkezésre, így az egyes prototípus fázisoknál lehetőség van azoknak a pontosítására. Gyakran alkalmazzák olyan területeken is, ahol az

egyes prototípusok használatával nagy megbízhatóságú rendszert akarnak létrehozni. A Spirál modell határozott hátránya, hogy nagyon tapasztalt vezető kell hozzá, illetve egyáltalán nem bánik gazdaságosan az erőforrásokkal.

A Spirál modell fázisai (negyedei):

Determine Objectives, Alternatives, and Constraints (A célok, alternatívák és megkötések azonosítása)
Ennek a negyednek a fő funkciója, hogy azonosítsa a termék/projekt céljait, mind funkcionalitás, mind teljesítmény vonatkozásában. Felméri a megvalósítási alternatívákat (új rendszer tervezése, meglévő komponensek használta stb.)

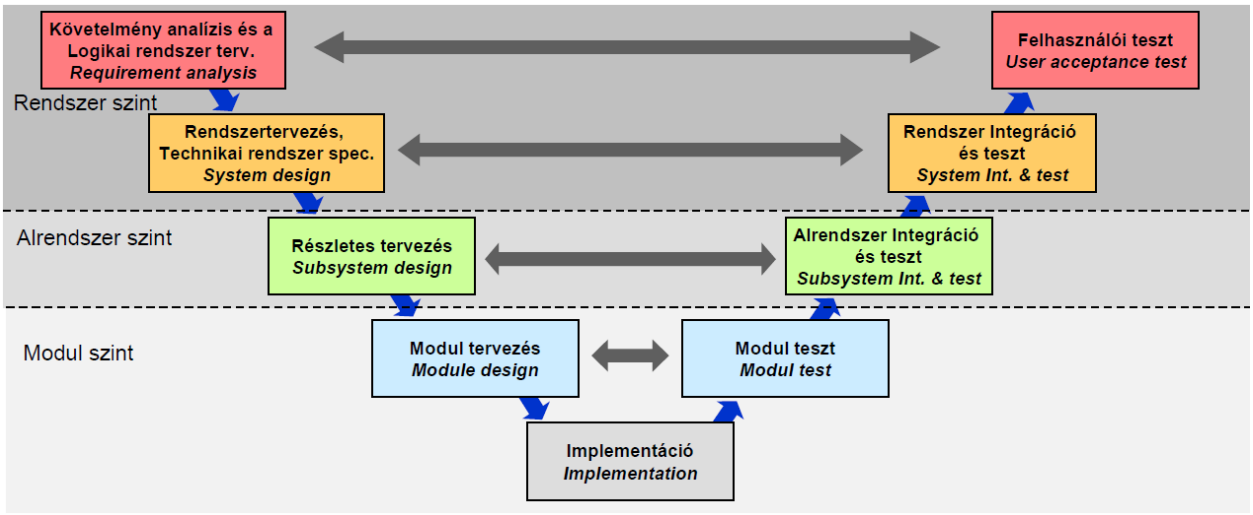
Evaluate Alternatives, Identify, Resolve Risks (az alternatívák kiválasztása, az alternatívák kockázatainak azonosítása)
Kiválasztja a legjobb megoldásokat, amelyek a céloknak legjobban megfelelnek, és megvizsgálja azok a kockázatait. A legjobbnak tűnő alternatívához prototípust készít. Ezekkel az iteratív prototípusok által feltérképezhető és kipróbálható kockázat megelőző megoldásokkal igyekszik a modell a végtermék kockázatainak lehető legteljesebb kivédésére.

Develop, Verify, Next-Level Product (Fejlesztés, ellenőrzés, következő termék előkészítése)
A következő fázisnak megfelelő termék kiválasztása, és a következő fázis előkészítése. Ha az előző prototípus már teljesen működő verzió volt, akkor egy vízesés modellszerű fejlesztés befejezése.

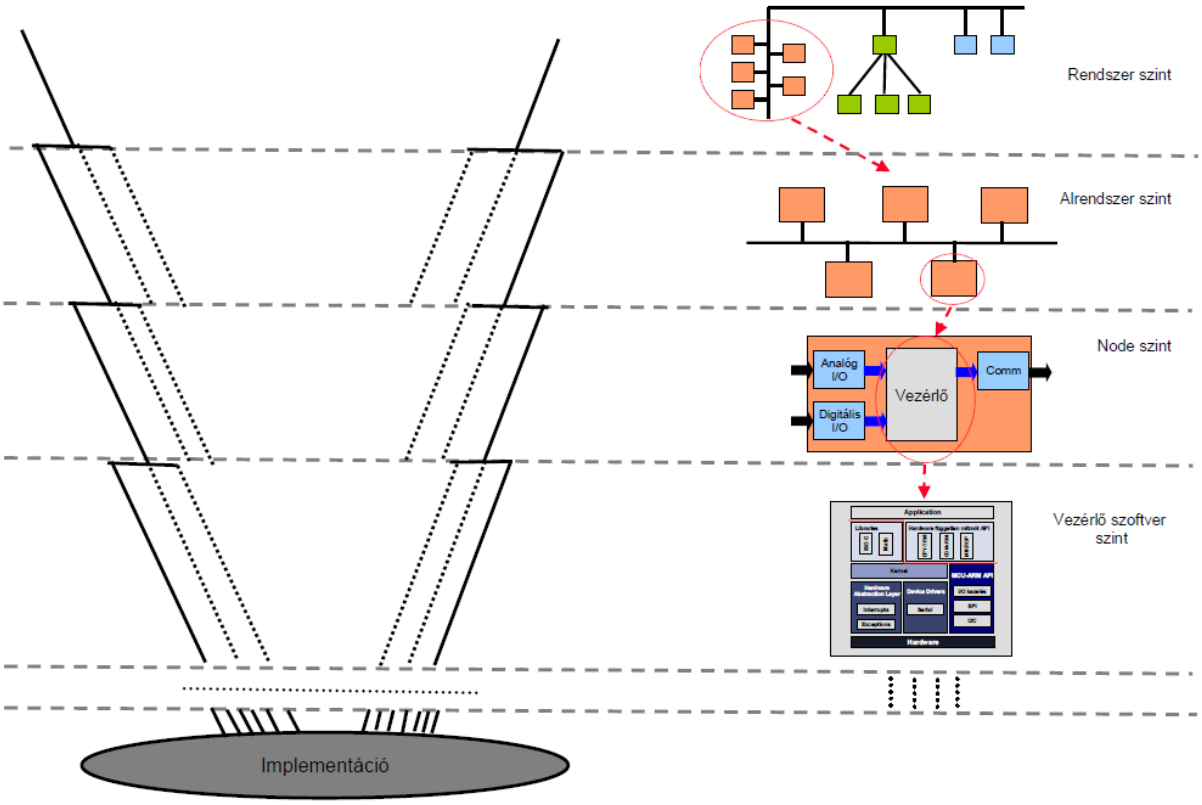
Next Phases (A következő fázis)
A következő lépés és annak ütemének megtervezése.

29. **Mutassa be az alap *V-modellt!* Miben tér el ez a *Vízesés* modelltől?**

V-modell gyakorlatilag a *vízesés modell* továbbfejlesztése: fontos és látványos eltérés a tesztelési ág visszahajtása. Ezzel azonos hierarchia szintre emelkedtek az összetartozó tesztelések és a tervezések. A rendszer egyik nagy előnye a fejlesztési absztrakciós szintek bevezetése, és az azonos absztrakciós szinthez tartozó teszt és fejlesztési lépések összekapcsolása (később látni fogjuk ennek a megvalósítását).



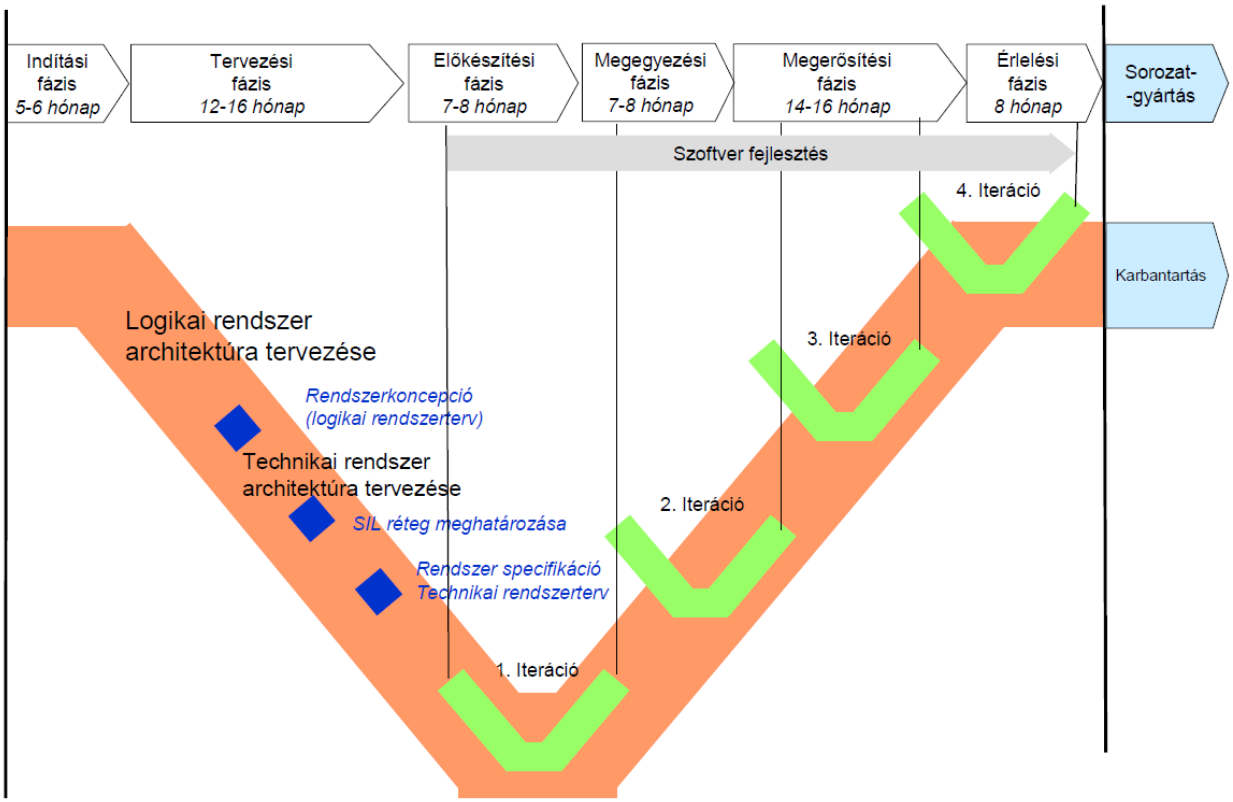
30. **Ismertesse egy tényleges komplett rendszer tervezésénél alkalmazott hierarchia szinteket, hogyan képesek ezek a szintek beépülni a *V-modell* szemléletébe?**



Az absztrakciós szintekre való bontás után érezhető, hogy nincs olyan fejlesztési modell, ami elágazás nélkül végig tudná követni ezt a sok szintet. Azért, hogy a gyakorlatban ezeket, a modelleket használni lehessen, a legtöbb esetben a fejlesztési modell az egyes szinteken külön ágakra, külön feladatokra osztódik szét. Ezeknél a fa jellegű elágazásoknál történik meg tipikusan a feladatok alvállalkozóknak, fejlesztési csoportoknak való kiadása is.

31. **Mutasson be egy tipikus rendszer fejlesztését *V-modell* alapján, amely iterációkat is tartalmaz!**

Ahhoz azonban, hogy az ilyen modell a valóságban is működjön fontos figyelembe venni olyan praktikus tapasztalatokat, mint például, hogy a valóságban elsőre sohasem készül tökéletes rendszer. Tehát a modellnek, vagy annak gyakorlati végrehajtásának lehetőséget kell adnia valamilyen iterációra. Az iparban egy termék elkészítésénél tipikusan 3-4 iterációs ciklussal számolnak.

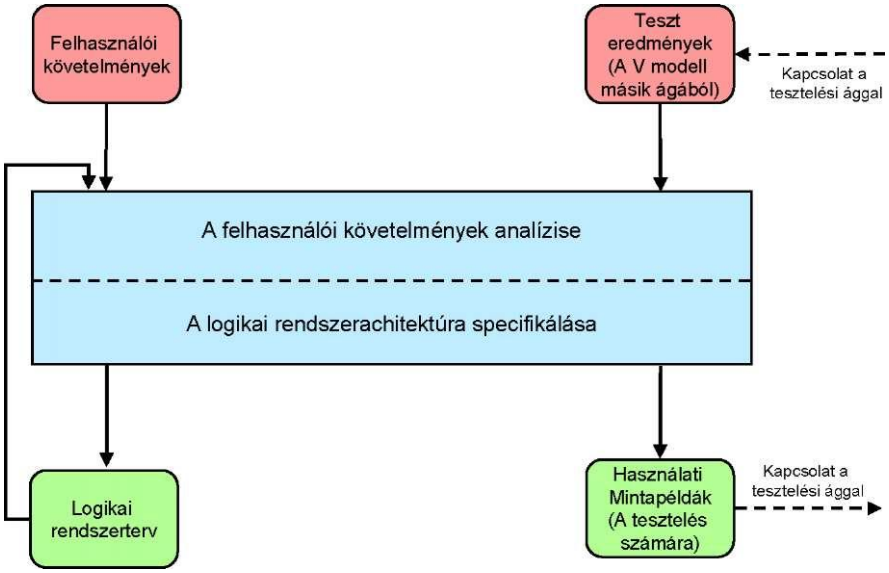


32. ***Mutassa be a* Követelményanalízis, és a Logikai rendszerterv elkészítése *lépést.***

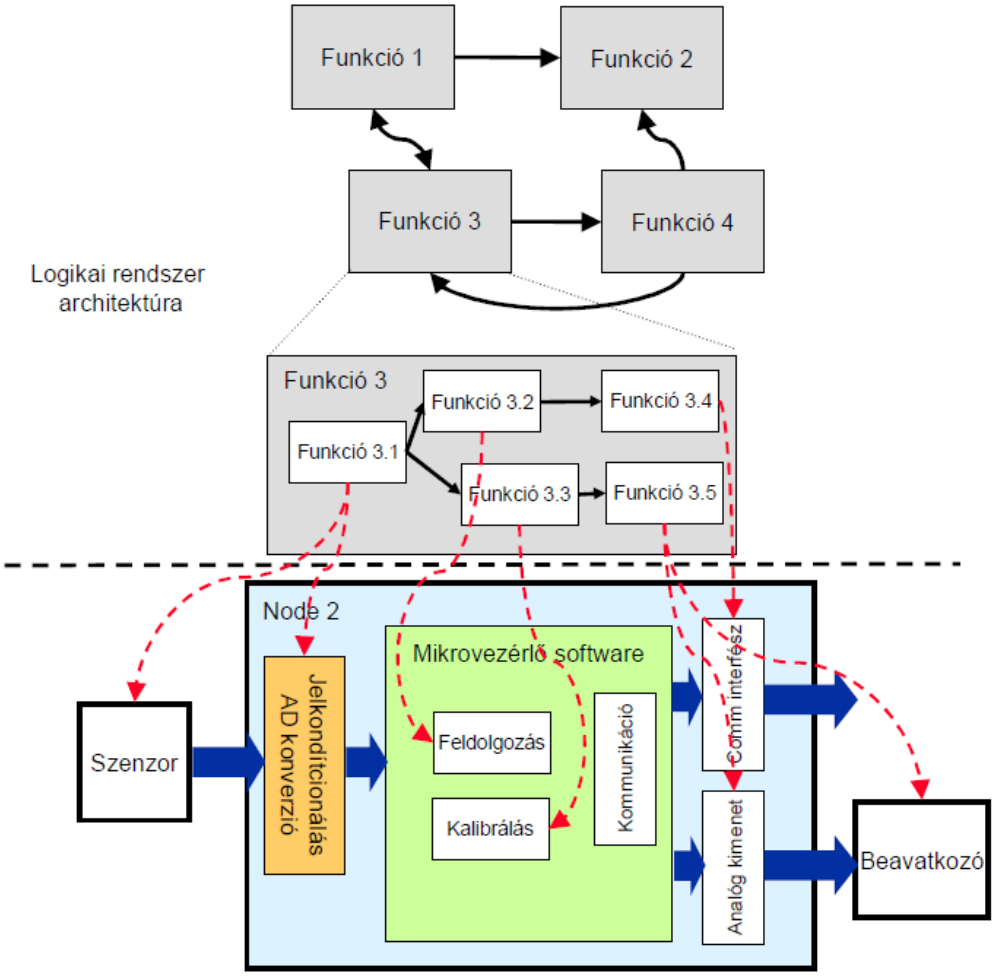
A lépés első feladata, hogy felmérje és elemezze a felhasználói követelményeket. Az esetek többségében a bemenő felhasználói követelményeket a *Requirement development* folyamat biztosítja. Ennek a lépésnek a feladata, hogy megtalálja a követelményekben rejlő inkonzisztenciát, illetve műszaki nyelvre fordítsa le a felhasználó igényeit. A lépés második feladata, hogy a megismert felhasználói igények alapján elkészítsék a rendszer logikai vázát, rendszertervét. A fő cél a rendszert alkotó logikai egységek és funkcióik, interfészeik feltárása, definiálása. A logikai rendszerarchitektúra semmilyen megvalósítási részletkérdéssel nem foglalkozik. Csak azzal, hogy milyen tulajdonságú rendszer fog létrejönni. Röviden összefoglalva a logikai rendszerarchitektúra az alábbi két dolgot tartalmazza:

- Definiálja azokat a tulajdonságokat, funkciókat, amikkel a rendszer rendelkezik
- Definiálja azokat a tulajdonságokat, funkciókat, amikkel a rendszer nem rendelkezik

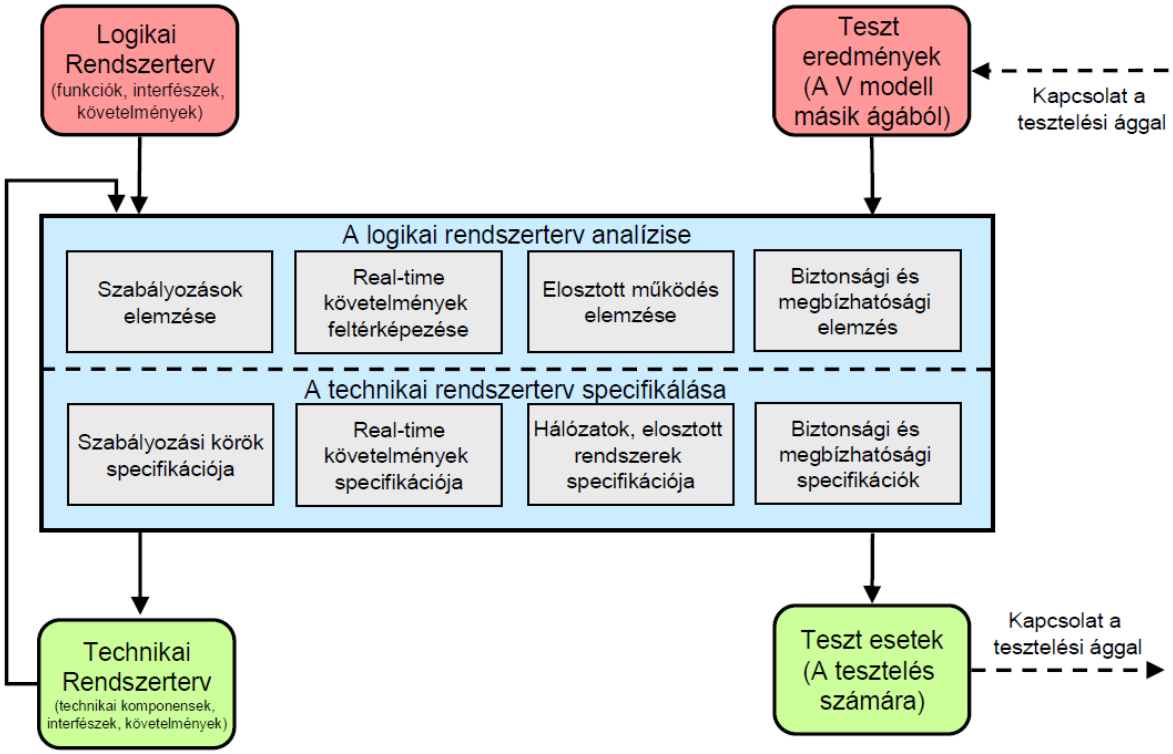
Természetesen az ilyen logikai rendszerarchitektúrák létrehozásánál a szöveges leíráson kívül különféle modellezési eszközöket is használnak, elsősorban az UML alapú leírások közkedveltek. Egy esetleges iterációnál felhasználják a tesztek eredményeit, és az előző ciklusban létrehozott rendszertervet. Fontos azonban megjegyezni, hogy a gyakorlati alkalmazásban igyekeznek ezt a lépést úgy megvalósítani, hogy iterációra ne legyen szükség, hiszen egy ezen a ponton történő változtatás az összes hierarchiában alacsonyabb szinten álló folyamat újragondolását hozza magával. Például, a 3.6 ábrán megfigyelhetjük, hogy a logikai rendszerterv hozzávetőlegesen az első év végére előáll, és utána már kötöttnek tekintett. Ez egy normál fejlesztésnél megoldható így, hiszen tudjuk, mit akarunk létrehozni, nagy funkció módosításra már nem lesz szükség, a kisebb specifikáció változtatásokat pedig a *requirement management* folyamaton keresztül le lehet kezelni.



33. **Mutassa be egy ábrán a logikai rendszerterv és technikai rendszerterv közötti kapcsolatot!**

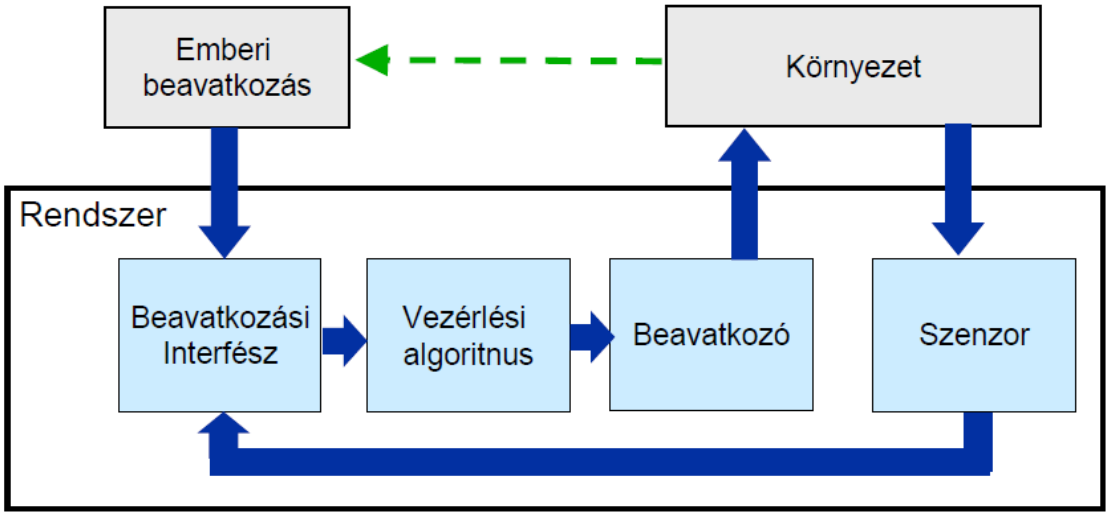


34. **Ismertesse a technikai rendszer architektúra specifikálásának lépéseit!**



35. **Mit a jelentősége a technikai rendszer architektúra specifikálása során elvégzett *„szabályozási körök elemzése és specifikációja"* lépésnek.**

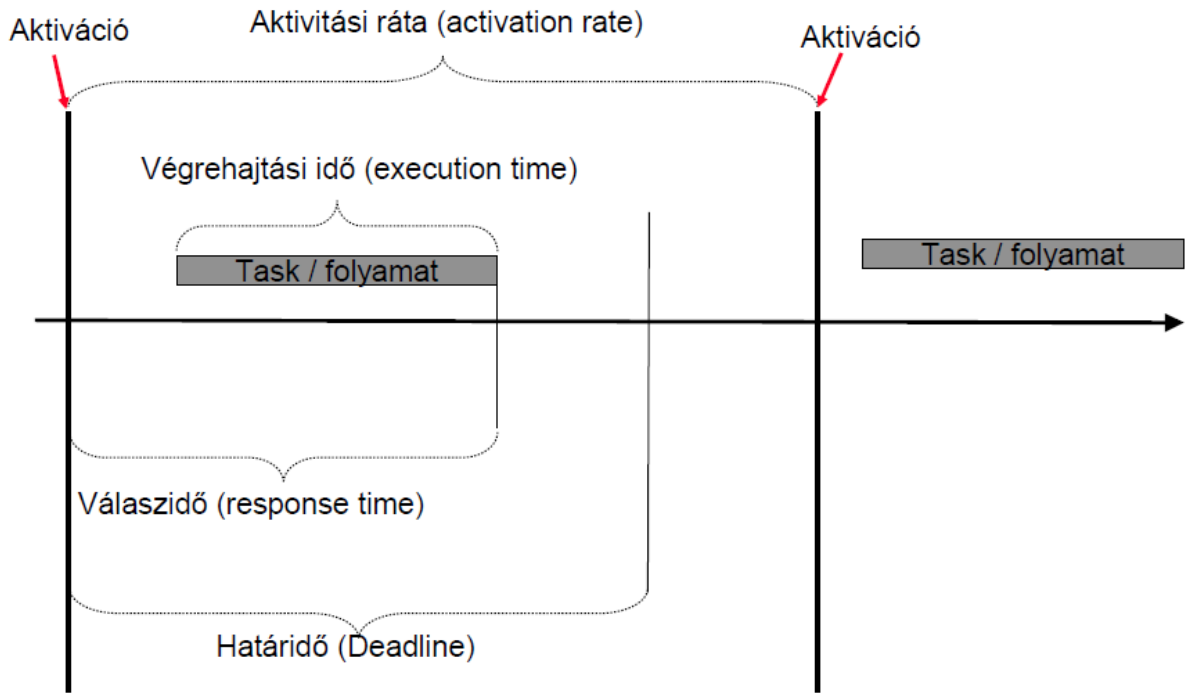
Amikor egy technikai rendszerarchitektúrát specifikálunk, akkor a legtöbb esetben először is felmérjük a rendszerben található szabályozási köröket. Legyenek azok nyílt, vagy zárt hurkúak. Erre azért van szükség, mert a szabályozási körök felmérése során meg tudjuk állapítani, hogy mit és milyen pontossággal kell mérnünk, a beavatkozásnak milyen pontossággal kell végrehajtódnia. Továbbá a be és kimenetek elemzésén túl lehetőségünk van annak a meghatározására is, hogy az egész folyamat meddig tarthat, tehát milyen real-time követelményeknek kell megfelelnünk, hogy az adott szabályozás stabil legyen. A szabályozások feltérképezése során lehetőségünk van képet alkotni a szükséges elosztottság mértékéről, tehát a később specifikálandó hálózati elrendezésről és számítási kapacitásról is. Gyakorlatilag ez a lépés abból áll, hogy a logikai rendszertervben a 3.10 ábrán láthatóhoz hasonló vezérlési köröket keresünk, azonosítjuk azok komponenseit és felmérjük a komponensekhez, valamint az egész rendszerhez tartozó technikai paramétereket, igényeket.
Már ezen a ponton elkezdődik a tényleges hardver figyelembe vétele, hiszen a digitális rendszerek tulajdonságukból adódóan mindenképpen diszkrét idősorokon alapuló szabályozást valósítanak meg. Figyelembe kell venni a mikrovezérlők véges feldolgozási sebességét, beleértve a kisebb számábrázolási felbontásból eredő kerekítési hibákat, az AD átalakítók késleltetését, nemlinearitását stb.



36. **Mit a jelentősége a technikai rendszer architektúra specifikálása során elvégzett *„Real-time követelmények feltérképezése és specifikációja"* lépésnek?**

A szabályozási hurkok meghatározása után a rendszer mintavételezési és reagálási ideje elemezhetővé és specifikálhatóvá válik, így össze lehet állítani az egész rendszerre vonatkozó időzítési követelményeket, amik aztán lebonthatók az egyes részegységekre vonatkozó követelményekre. Az időzítési viszonyok vizsgálata szolgál alapul a későbbi döntéseknél, amikor meghatározzuk mind az implementálásnál használt hardvert, mind a kommunikációs hálózatot, mind az olyan szoftver csomagokat, mint pl. a real-time operációs rendszerek.

Az elemzés célja gyakorlatilag a vezérlési/szabályozási kör egyes részeire időkorlátok adása. Az összes folyamatot tekintve a szokásos megállapítandó paraméterek: *végrehajtási idő (execution time), válaszidő (response time), határidő (deadline), aktivitási ráta (activation rate)*



37. **Mit a jelentősége a technikai rendszer architektúra specifikálása során elvégzett *„Elosztott működés elemzése, az elosztott rendszerek és hálózatok specifikálása"* lépésnek?**

A real-time időzítések és a szabályozási követelmények ismeretében a rendszer funkciói szétosztásra kerülnek az egyes node-ok, egységek között. Tehát ténylegesen specifikáljuk, hogy melyik funkciót melyik hardver egység és hol hajtja végre. Ahhoz, hogy ezt megtegyük először elemezni kell, hogy mely műveletek párhuzamosíthatóak, illetve melyeket lehet és célszerű fizikailag nem egy helyen, hanem elosztottan végrehajtani. Amikor ezek kiválasztásával végeztünk, akkor specifikálni kell a kommunikációs hálózatot, hálózatokat, amely az egyes különálló hardvereken működő funkciókat összeköti. Gyakorlatilag ennél a lépésnél már akár a hálózat típusát is ki lehet választani, hiszen már minden paramétert tudunk ahhoz, hogy a hálózattól elvárt átviteli sebességet meg tudjuk határozni. A küldött, fogadott üzeneteket és azok gyakoriságát is specifikálni lehet már. Összegezve tehát a rendszer funkciók csomópontokra (node)-ra való felosztását illetve a kommunikációs mátrix meghatározását végezzük ezen a ponton. Itt a node-ok meghatározásánál kell arra törekedni, hogy a már meglévő komponenseket újrafelhasználjunk, ha lehetséges. Amennyire lehet, gondot kell arra is fordítani, hogy a létrejövő új node-okat később a lehető legtöbb helyen lehessen alkalmazni, újrafelhasználni. Tehát, sokszor ehhez a ponthoz tartozik a rendszer modularitásának megtervezése is.

38. **Mit a jelentősége a technikai rendszer architektúra specifikálása során elvégzett *„A megbízhatóság és biztonság szempontjából fontos részek azonosítása"* lépésnek?**

A beágyazott rendszerek jelentős részénél a biztonságosság és megbízhatóság alapkövetelmény, így már a tervezés elején figyelembe kell ezeket venni. A *technikai rendszerarchitektúra tervezése* ezért tartalmaz egy biztonsági és megbízhatósági analízist, amely meghatározza a későbbi fejlesztési lépésekben alkalmazott eszközöket és módszereket. (bővebben: következő kérdés)

39. ***Mutassa be a* „biztonságossági és megbízhatósági analízis" *lépéseit!***

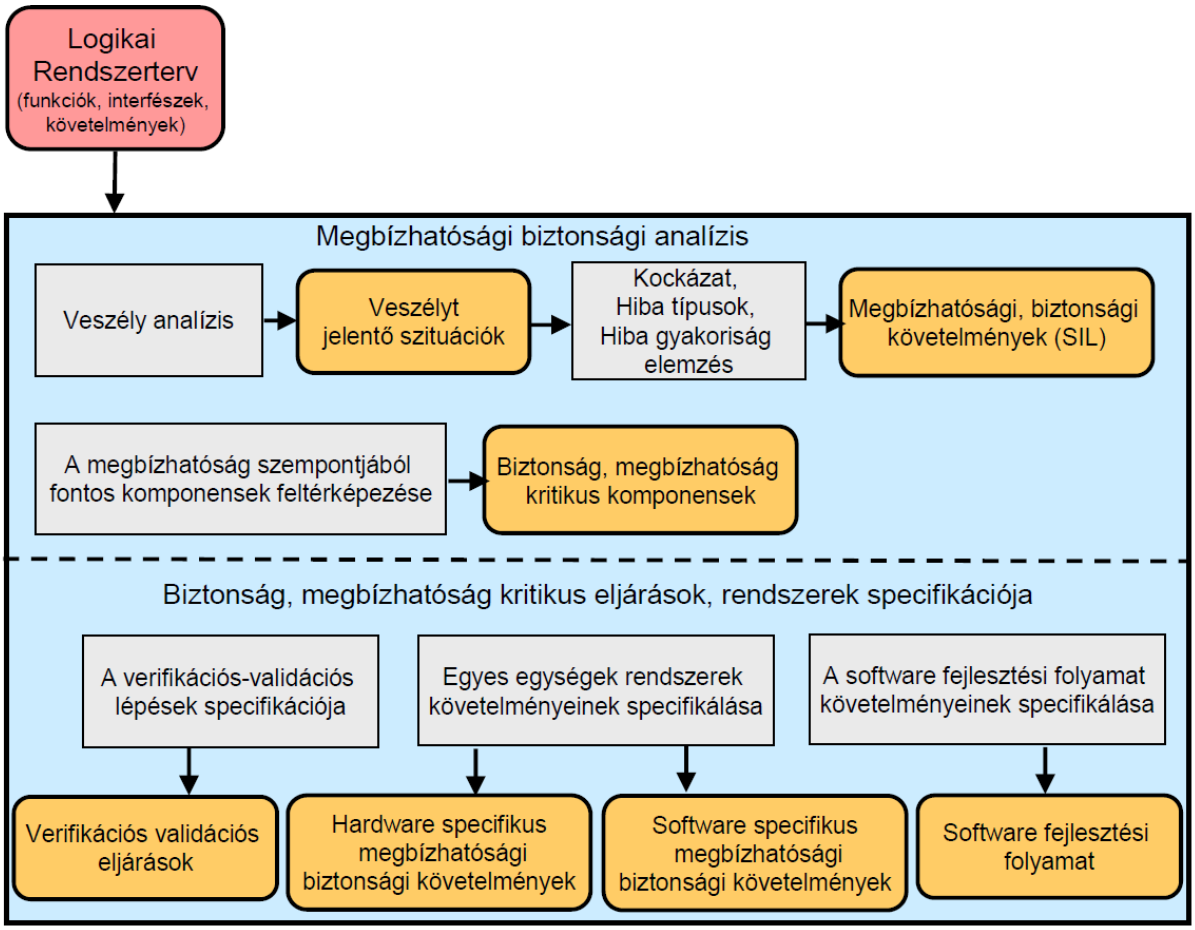
A biztonságossági és megbízhatósági analízis első lépése, hogy felmérjük a rendszerre vonatkozó lehetséges veszélyhelyzeteket. Miután a veszélyes szituációkat ismerjük, elemzésre kerül az ezek által jelentett kockázat is.

Risk (kockázat): Valaminek a kockázatát nem lehet egzakt módon megadni, a kockázat megadásra a legelfogadottabb mód, hogy a baleset valószínűségét és az ebben az esetben bekövetkező kár mértékének a szorzatát vesszük. R = Probability of Accident * Accident damage

A káron itt összevonva értünk emberi sérülést és anyagi, környezeti kárt. A kockázat meghatározás szerves része, hogy meghatározzuk a veszélyeket kiváltó hibákat és azok gyakoriságát is.

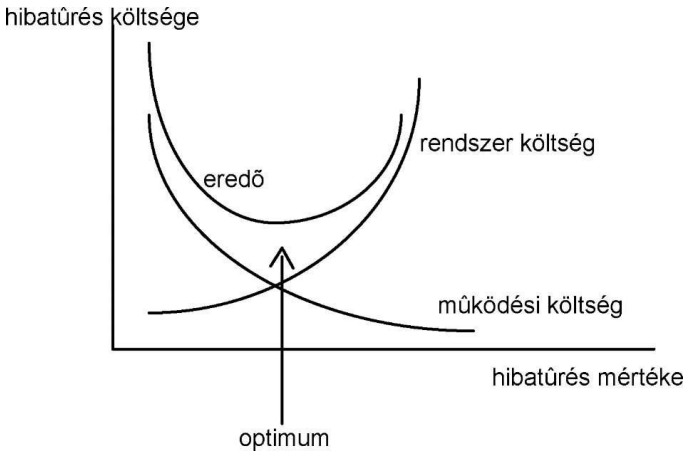
Miután a rendszerre vonatkozó kockázatot felmértük, meg kell határoznunk az egész rendszer hibatűrési szintjét is. Ez tipikusan az ún. *limit-risk*, vagy kockázati határ megtalálásával kezdődik.

Miután meghatároztuk a rendszerre jellemző megbízhatósági szintet a következő feladat a gyengepontok felderítése, ahol szükséges redundancia specifikálása.



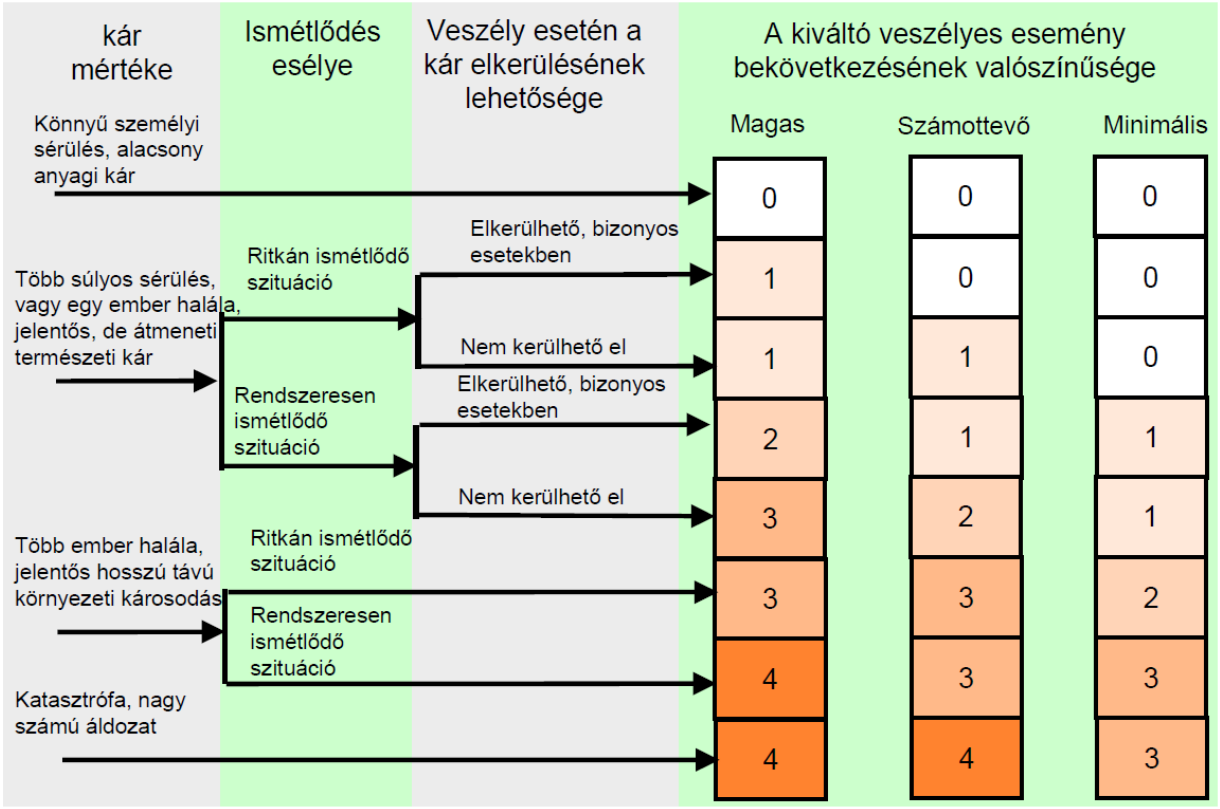
40. **Mi az a *„Limit Risk (kockázati határ) "* és mi köze van a rendszertervezéshez?**

A kockázati határt szintén nem lehet egzakt módon meghatározni (függ a hatályos törvényektől, piaci pozíciótól stb.), a legjobb definíció rá, hogy egy olyan optimum, ahol a rendszer költsége (redundáns vezérlő stb.) és a rendszer működésének költsége (kártérítési perek, visszahívások) együttesen a minimumot adják. Természetesen ezt az optimumkeresési folyamatot különböző szabványok támogatják. Például az IEC61508-as szabvány által specifikált ***SIL (Safety Integrity Level)*** rétegek egy ilyen besorolást nyújtanak.



41. **Mi az SIL *Safety Integrity Level és* mik alapján kerül meghatározásra?**

A ***SIL*** a legelterjedtebben alkalmazott megbízhatósági besorolási mód. Gyakorlatilag mindenhol ezt használják és a későbbi fejlesztési lépések az itt megállapított szintektől függenek. A SIL rétegek meghatározásának egy módja:



42. **Hogyan befolyásolja a meghatározott SIL *(Safety Integrity Level)* besorolás a rendszertervezés további lépéseit? Mutasson rá példát!**

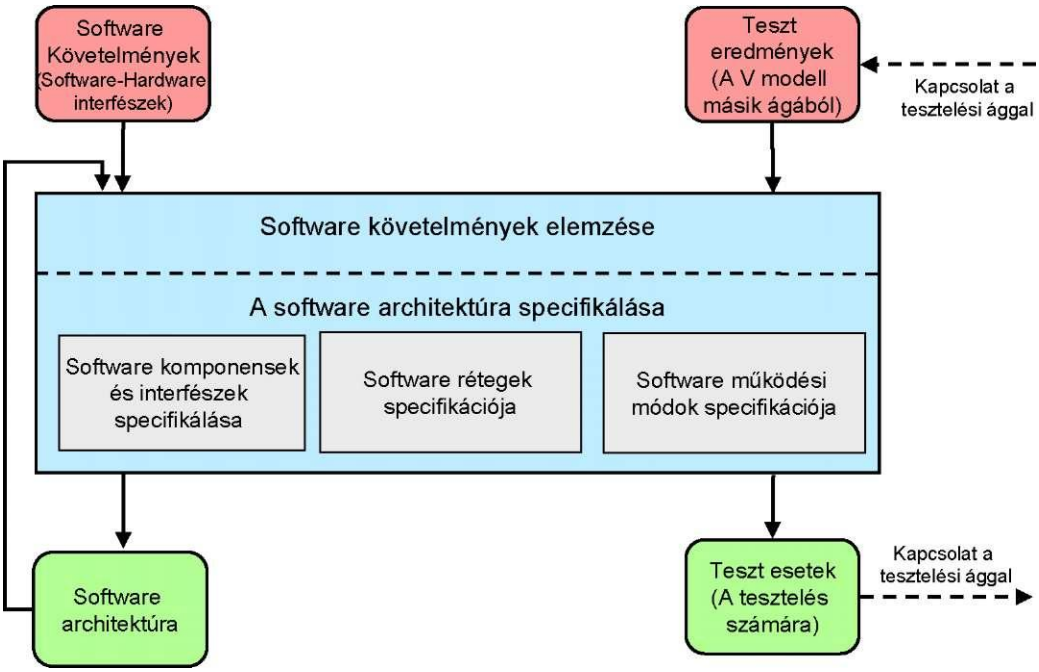
Miután meghatároztuk a rendszerre jellemző megbízhatósági szintet a következő feladat a gyengepontok felderítése, ahol szükséges redundancia specifikálása.
Ez gyakorlatilag a technikai rendszerterv elején azonosított szabályozási, vezérlési körökön alapul. Lényege, hogy ezeknek a köröknek az elemeit, egyes komponenseit vizsgáljuk szokásos megbízhatósági analízis módszerek segítségével. Ilyen megbízhatósági analízis módszer például a BOOLE modell alapján, párhuzamos, illetve soros rendszer összetevők segítségével végzett számítások. Egyéb - például Markov láncokon alapuló - módszereket is gyakran használnak ennél a lépésnél. Ez a megbízhatósági analízis kimutatja, hogy mely rendszerkomponenseknél kell különösen odafigyelni. Fontos itt megjegyezni, hogy SIL szintet nem csak a rendszerhez, de komplexebb rendszerkomponensekhez is szoktak rendelni. Ehhez nyújt segítséget, amikor meghatározzuk az egész rendszer, illetve az egyes komponensek megbízhatósági paramétereit.

A megbízhatósági analízis, illetve a SIL besorolás a további lépéseket erősen befolyásolni tudja. Számos későbbi fejlesztési lépés, mint a szoftverfejlesztési módszerek, verifikációs és validációs eljárások függnek ettől a besorolástól. Minden cégnél van arra vonatkozó utasítás, hogy milyen SIL szinten milyen fejlesztési módszereket kell használni.

Tevékenység	SIL0	SIL1	SIL2	SIL3
A funkcionális követelmények belső csoport szintű átbeszélése	++	++	-	-
A funkcionális követelmények külső személy általi ellenőrzése	+	+	++	++
Prototípus készítés	0	0	+	++
Simuláció	+	+	++	++
Hiba ok diagrammok készítése	+	+	+	++
Vezérlési folyam analízis	+	++	++	++
Adat folyam analízis	+	++	++	++

43. ***Mutassa be a* Szoftver architektúra megtervezésének *lépéseit!***

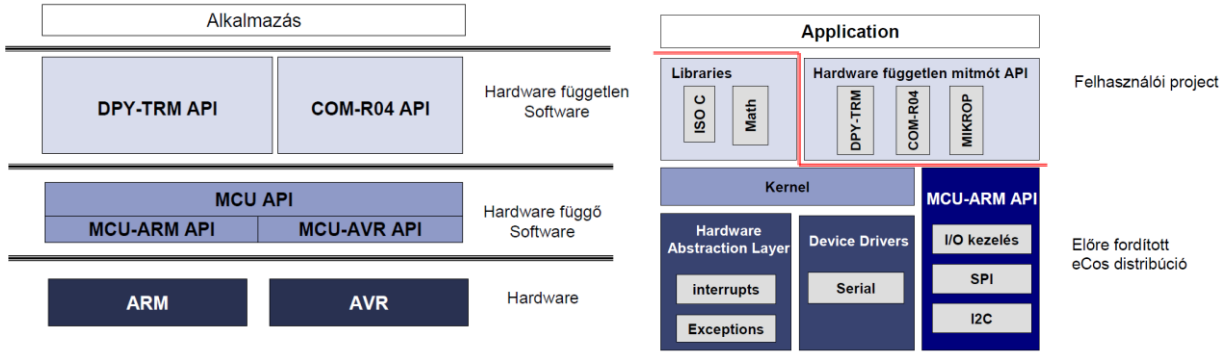
- A szoftver rendszer határainak specifikációja
- A szoftver komponensek és azok interfészeinek specifikációja (nem tartalmazza az egyes komponensek részletes specifikációját, csak azt, hogy a rendszer milyen komponensekből áll és azoknak mi a kapcsolata)
- A szoftver rétegek specifikációja.
- A szoftver működési módjainak specifikációja.



44. **Mutassa be egy általános beágyazott rendszer szoftverrétegei és magyarázza meg az egyes rétegek szerepét!**

A szoftver funkciókat a legtöbb esetben rétegekre bontják. Ezek a rétegek tipikusan a *hardware abstraction layer, kernel / device drivers, application support layer* és az *application layer.* A kommunikációs protokolloktól eltérően a szoftver rétegek általában nem az ún. *linear-order layer model* alapján épülnek fel, mint például az OSI rétegek, hanem az ún. *strict-order layered model* alapján. A *linear-order layer model* azt jelenti, hogy egy felsőbb réteg csak a követlen alatta lévővel

kommunikálhat, míg a *strict-order layered model* csak azt definiálja, hogy egy alacsonyabb rétegbeli nem férhet hozzá magasabb rétegbeli szolgáltatásokhoz, de a magasabb rétegbeli elemek minden alacsonyabb réteghez hozzáférhetnek. A szoftvertervezés egyik legfontosabb része ennek a réteges szerkezetű API *(Application Programing Interface)* hierarchiának a megteremtése. Egy jól létrehozott réteges szerkezet biztosítja a kód könnyű hordozhatóságát, valamint nagyfokú újrafelhasználhatóságát. Rossz struktúra garantáltan káoszhoz vezet.



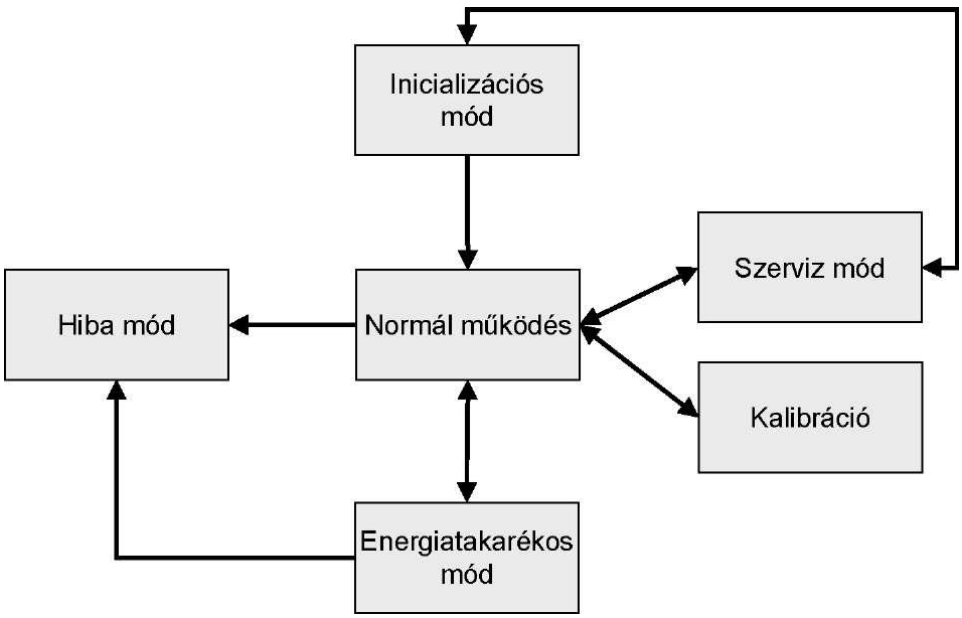
3.19 ábra A mitmót API-k hierarchikus szerkezete

45. **Ismertesse egy általános beágyazott rendszer tipikus működési módjai és azok kapcsolatai! Röviden ismertesse az egyes működési módokban elvárt viselkedést!**

Egy beágyazott rendszeren futó programnak általában több működési módja van. Ilyen módok tipikusan a Bemérési mód - Kalibrációs mód - Szerviz mód - Energiatakarékos üzemmód - megbízhatóságkritikus csökkentett üzemmód - Inicializációs mód - Normál működés - Hibamód stb. Természetesen nem minden üzemmód létezik minden rendszernél.

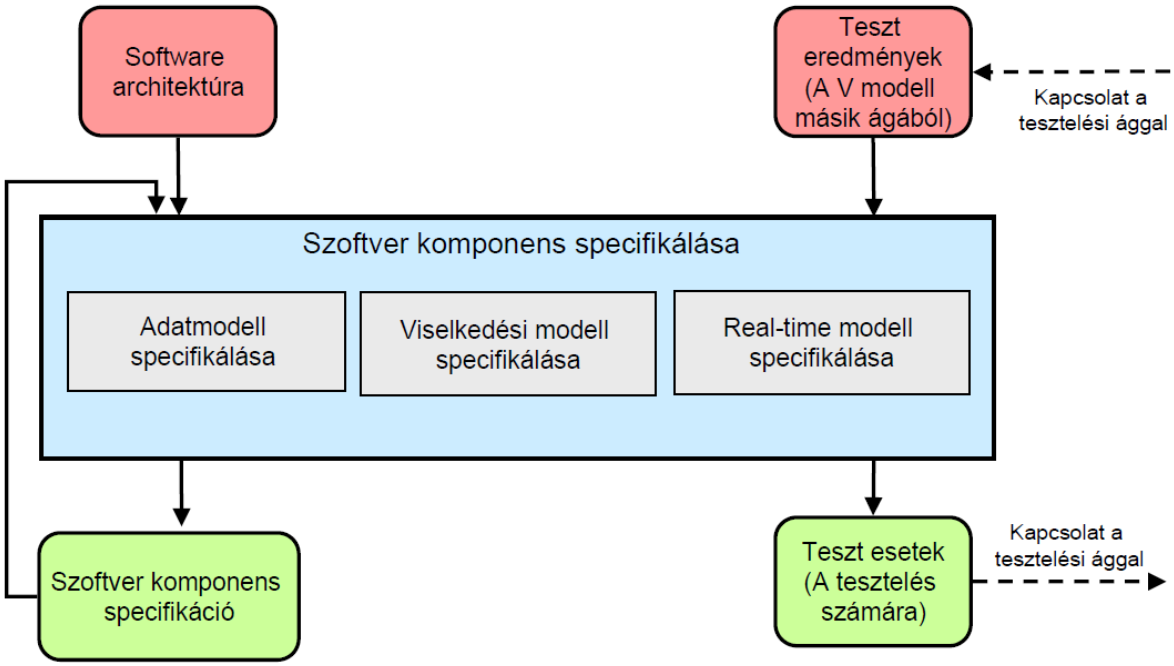
A normál működés közben nem látható módok: *bemérési mód, szerviz mód* stb. Ezek tipikusan a rendszer terepre kihelyezését, a gyártás ellenőrzését könnyítik meg, kimaradásuk jelentősen csökkenti a gyárthatóságot és használhatóságot is.

A működési módok definiálása a legtöbb esetben valamilyen állapotgépes megadással történik, ahol a működési módokon kívül a működési módok közötti átjárhatóság feltételeit is megadjuk. Erre általában UML *State diagrammot* használnak.



46. ***Ismertesse a* Szoftver komponensek specifikálása *lépést!***

A szoftver architektúra megtervezésénél specifikálásra került, hogy a rendszer milyen szoftver komponensekből áll, és hogy ezeket milyen interfészek kötik össze. Ennek a lépésnek a célja, hogy kidolgozza az egyes komponensek részletes viselkedését. Egy szoftver komponens specifikációját három részből: az adatmodellből, a viselkedési modellből és a real-time modellből állítják össze.



47. **Indokolja, hogy miért célszerű sok esetben a szoftverkomponensek tervezésénél viselkedési és az adatmodellt szétválasztani!**

Az adat és a viselkedési modellt és azok verzióit néhány esetben élesen szét is választják külön adat és vezérlő szoftver verziókat létrehozva. Ezt a szétválasztást tipikusan csak azokra a részekre alkalmazzák, amelyek kizárólag az adatok megváltoztatásával új rendszer verziót képesek létrehozni. Ilyen például egy szabályozási kör, ahol a PID szabályozás paramétereinek megváltoztatásával egészen eltérő vezérlések hozhatóak létre ugyanazzal a programszerkezettel és hardverrel (gyakorlati példa egy motorvezérlő, ahol a motor teljesítményének és típusának változásával a vezérlés adatai módosulnak, de a vezérlési folyam és a hardver nem feltétlenül).

48. **Ismertesse a szoftver komponensek, illetve rendszerek real-time viselkedésének tervezésénél alkalmazott fogalmakat és módszereket!**

A *real-time modell specifikálása* rész foglalkozik azzal, hogy a szoftverkomponens egyes funkcióit taszkokba rendezze. Illetve, hogy megszabja az így taszkokba rendezett vezérlések időlimitjeit, úgy hogy a szoftverrendszer majd teljesíteni tudja a technikai rendszer architektúra által megszabott real-time kritériumokat. Tehát az egyes szálakra a szokásos paramétereket, mint *végrehajtási idő (execution time), válaszidő (response time), határidő (deadline), aktivitási ráta (activation rate)* kell megállapítani úgy, hogy együttesen betartsák a technikai rendszerterv real-time kritériumait. (36-os kérdés)

A taszkokra való szétszedésnél az adatfolyamot szokták feldarabolni a legtöbb esetben. A taszkok létrehozásánál fokozottan figyelembe kell venni az alkalmazni kívánt futtatókörnyezet, real-time operációs rendszer tulajdonságait. Preemptív működés esetén meg kell tudni határozni azt, hogy az egyes szálak hogyan és mennyire késleltetik egymást, és ezzel mennyire befolyásolják a rendszer működését. Erre egy elterjedt és széles körben alkalmazott módszer például a Deadline Monotonic Analysis (DMA).

49. Ismertesse a „A *szoftver komponensek implementálása"* lépéssel kapcsolatos problémákat, illetve nehézségeket!

Hardver limitációk

A megvalósítási folyamatnál az egyik fontos megkötést jelentek a hardver által szabott korlátok. A hardver költség a legtöbb iparágban fontos kérdés. Lényeges, hogy akár csak pár centtel is, de olcsóbb legyen a hardver. Ezért sokszor plusz nehézség rakódik a szoftverre, mert számos esetben inkább megnövelik a szoftver fejlesztési költségeket, ha spórolni tudnak a hardveren, ami nagyobb darabszámnál általában meg is térül. Bár a hardver-szoftver szétválasztás a korábbi a *technikai rendszer architektúra* tervezésénél dőlt el, de annak hatásai itt is érződnek. Például a költségek kímélése miatt a szabályozási körben alkalmazott processzorok nem lebegőpontosak, hanem fix pontosak. Az ilyen fontos megkötések pedig rányomják a bélyegüket az egész programra, hiszen például az adott esetben fokozott figyelmet kell fordítani a számábrázolás miatti kerekítési, alulcsordulási hibákra, ami miatt rossz esetben akár az egész szabályozási kör is instabillá válhat. Szintén fontos megkötést jelenthetnek a ROM és RAM területek korlátai. Egy szokásos mikrovezérlőben jóval több ROM terület található, mint RAM. Emiatt, illetve amiatt, hogy ne kelljen nagyobb memóriájú mikrovezérlő variánst venni, a programozóknak, ahol lehetőség van rá, spórolni kell a RAM területtel és inkább ROM-ot használni. (Például: néha bonyolult sok változót igénylő számítások helyett look-up table-t használnak fel). Természetesen ezek az optimalizációs lépések nem mehetnek a minőség és megbízhatóság rovására.

Az adatmodell implementációja

Az adatmodell megvalósításánál a már megtervezett változókat implementáljuk. Itt a legfontosabb lépés a tervezés és az implementáció közötti különbségek kezelése. Például amikor egy vezérlő külső és belső RAM-mal is rendelkezik az sem mindegy, hogy mely változókat rakjuk a belső RAM területre (gyorsabb elérés) és melyeket a külső RAM-ba. Ennél a lépésnél szokták pontosítani a változók nyers értéke és a fizikai világ közötti viszonyt is. Például, ha egy hőmérséklet értéket egy 16 bites változóban tárolunk egész számokként 100-ad fokos felbontással, akkor itt adjuk meg és dokumentáljuk ezt a kódolási formát (specifikáljuk az ofszetet és faktort). Ezeket az adatokat néhány helyen külön leíró file-ba rögzítik, és fontos szerepet kapnak a tesztelési szakaszban (ilyen terület például az autóipar, ahol diagnosztikai és kalibrációs protokollok használják ezeket a leíró file-okat a tesztelési, bemérési lépéseknél). A fizikai megvalósítása során további limitációk adódhatnak a különböző aritmetikai és számítási módok miatt. Ilyen hibaforrást jelenthetnek a kerekítési hibák. Ahogyan már szó volt róla, ezek tipikusan az alkalmazott mikrokontroller korlátaiból adódnak. Az ún. approximációs hiba az alkalmazott számítási eljárásokból adódik (például egy nemlinearitás „ideális", harmadfokú kompenzációja helyett csak másodfokú kompenzációt alkalmazunk) Ezeket a problémás részeket fel kell tárni, és meg kell vizsgálni, hogy okozhatnak e működésbeli hibát.

A viselkedési leírás implementációja

A viselkedési modell implementációja a specifikációt hivatott kiegészíteni az adott processzor, hardver korlátait figyelembe véve, bár ezeket sokszor nehéz pontosan megadni.

A real-time modell implementálása

Ennél a résznél a real-time modell specifikációjának betartásához az implementáció előtt pontosan meg kell vizsgálni az adott hardver tulajdonságait, például interrupt modelljét, interrupt latency-ét stb. Az adott vezérlő ismeretén túl nagyon fontos az operációs rendszer adott vezérlőn való működésének feltérképezése is. Bár a Real Time operációs rendszerek esetében a legtöbbször elmondható, hogy egy operációs rendszer szolgáltatás végrehajtásának viszonylag stabil időszelete van (a minimális és maximális idő közel megegyezik) ezeket, az időket azonban minden architektúrára külön mérésekkel ellenőrizni kell. További fontos lépés még az operációs rendszer felkonfigurálása is. A beágyazott operációs rendszerek némelyikénél például lehetőségünk van az ütemezési algoritmus, vagy a prioritás inverzió elkerülésére alkalmazott protokoll megadására is. Ezek helytelen beállítása könnyen okozhat nem várt működési rendellenességet.

50. **Mi a jelentősége a *megfelelő implementációs környezet kiválasztásának* a szoftverkomponensek implementációja során? Mik az implementációs környezet kiválasztásának főbb szempontjai?**

A legtöbb cégnél a megvalósítási lépéseket igen szoros szabályokhoz kötik, amelyek természetesen függnek attól, hogy az adott megvalósítandó komponens megbízhatóságkritikus modul része-e, vagy nem.

Szabály	SIL0	SIL1	SIL2	SIL3
Megfelelő programozási nyelv használata	++	++	++	++
Coding style guide használata	++	++	++	++
Elnevezési konvenció használata	++	++	++	++
A nyelvi készlet korlátozása	++	++	++	++
Deffenziv programozás	0	0	+	++
Dinamikus memóriakezelés kerülése	+	+	++	++
Interuptok korlátozott használata	+	+	+	++
Pointerek korlátozott használata	+	+	+	++

51. **Mutassa be egy általános *Coding style guide* által tárgyalt területeket!**

- File struktúra és file elnevezési konvenció
- Program file-ok szerkezete
- Header file-ok szerkezete

52. **Mutasson példát arra, hogy egy *Coding style guide-ban* hogyan szabályozzák a forrásfájlok nevét és belső szerkezetét!**

A file nevének karakterrel kell kezdődnie. A file neve nem lehet 8 karakternél hosszabb, a kiterjesztése pedig 3 karakternél hosszabb. A szokványos file típus neveket kell használni: .c, .asm/.s, .o, .bin, .hex etc.

A program file szerkezete a következő:

1, Komment a file nevéről, rendeltetéséről, szerzőjéről, verziójáról, és verzió history-árol. Egyes verziókövető rendszerek automatikusan generálják, módosítják a file verzióra vonatkozó kommenteket. Ilyenkor ezeket a fejlesztőnek általában tilos kézzel módosítani.
2, Header file include-ok
3, Definiciók a következő sorrendben: Típus definíciók, Define-ok, Konstans makrók függvény makrók,
4, Globális változók: extern, nem static, static global sorrendben
5, függvények: általában hívási sorrendben, ha egy mód van rá.

Header file-ok szerkezete

1, Komment a file nevéről rendeltetéséről, szerzőjéről, verziójáról, és verzió history- árol. A file neve sohasem lehet normál C inklúdnévvel egyező pl.: „math.h".
2, A következő file kezdési szerkezet kötelező:

```
  #ifndef EXAMPLE_H
  #define EXAMPLE_H
...      / body of example.h file */
#endif /* EXAMPLE_H *
```

3, Fejléc file-okban ne deifiniáljunk változókat ha egy mód van rá.

53. **Mutasson példát arra, hogy egy *Coding style guide-ban* hogyan szabályozzák a változók elnevezését!**

Ezeket általában külön leírt szabályok adják meg, ilyen például az egyik leghíresebb az ún. *Hungarian Notatinon*, amelyet Simonyi Károly vezetett be a Microsoftnál. Az elnevezési szisztéma a magyar elnevezési logikából indul ki, ahol a vezetéknév megelőzi az „adott" keresztnevet. Ezt próbálja a változókra is bevezetni, ahol először a típus, vagy alkalmazási kör szerepel és utána csak az „adott" név. Általában kétfajta konvenciót különböztetnek meg a *System-et* és az *Application-t*, a *System* esetében a változó típusa, az *Application* esetében az alkalmazás típusa van belekódolva a változó nevébe.

Néhány példa:

- bBusy : boolean
- cApples : count of items
- dwLightYears : double word (system)
- fBusy : boolean (flag)
- nSize : integer (system) vagy count (application)
- iSize : integer (system) vagy index (application)
- fpPrice: floating-point
- dbPi : double (system)
- pFoo : pointer
- rgStudents : array, vagy range
- szLastName : zero-terminated string
- u32Identifier : unsigned 32-bit integer (system)
- stTime : clock time structure
- fnFunction : function name

Ez a jelölésrendszer sokszor kiegészül a láthatóságot befolyásoló előtagokkal is:

- `g_nwheeis` : member of a global namespace, integer
- `m_nwheeis` : member of a structure/class, integer
- `m_wheeis` : member of a structure/class
- `s_wheeis` : static member of a class
- `wheels` : local variable

54. **Milyen szabályhalmazokat ismer az implementációs nyelvi készlet korlátozására?**

Még a szabványos ISO C változatok specifikálása is igen szabatos. Rengeteg szintaktilag helyes, de szemantikailag rossz megoldást lehet létrehozni, amely olyan iparágakban, ahol valamennyire is megbízhatóságkritikus az alkalmazás, nem megengedett.

Ahhoz, hogy elkerülhessük ezeket a szintaktikailag helyes, de szemantikailag veszélyes részeket, a C nyelvi készletét korlátoznunk kell. Jelenleg két ilyen nyelvi korlátozás terjedt el. A MISRA-C, és a CERT Secure C szabálycsomagja.

55. **Mutassa be röviden a MISRA-C szerepét és céljait!**

MISRA C (Motor Industry Software Reliability Association)

Célja az volt, hogy javítsa az UK (United Kingdom) autóiparában használt szoftverek minőségét. A MISRA-C 1998 sikere nem csak az autóiparra terjedt ki, hanem széles körben alkalmazásra került a repülőgépiparban valamit az egészségügyi iparban is.

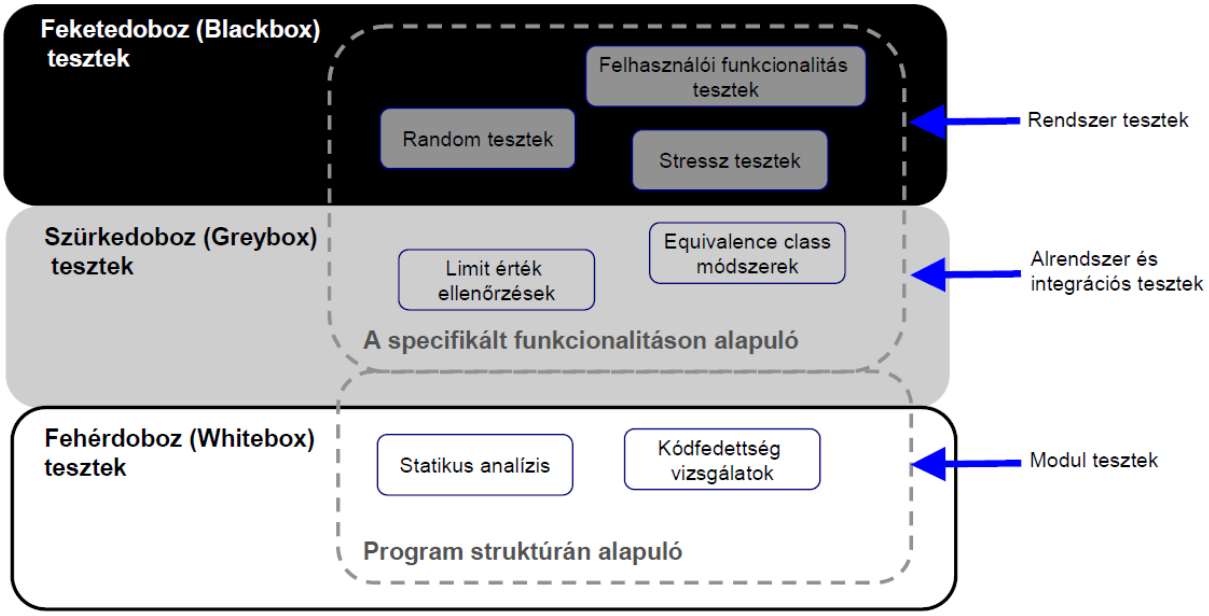
A MISRA-C 2004 a szabvány jelenlegi verziója, amely elfogadásra került az USA-ban (SAE J2632) és Japánban is. A MISRA-C 2004 javítja a MISRA-1998 hibáit, és pontosítja azt. A szabvány 121 *Mandatory* (kötelező) és 20 Advisory (javasolt) szabályt tartalmaz a C nyelv leszűkítésére. A MISRA C röviden összefoglalva a következő célokat tűzte ki:

- Szintaktikailag helyes, szemantikailag rossz megoldásokra felhívni a figyelmet.
- Tiltani a nem egyértelmű változó típus használatot.
- Szabályozni a precedencia zárójelezéseket.
- Tiltani a nem strukturált programozást eredményező szerkezeteket.

A MISRA közösség a C-n kívül még a C++ programozási nyelvhez is kidolgozott szabályrendszert.

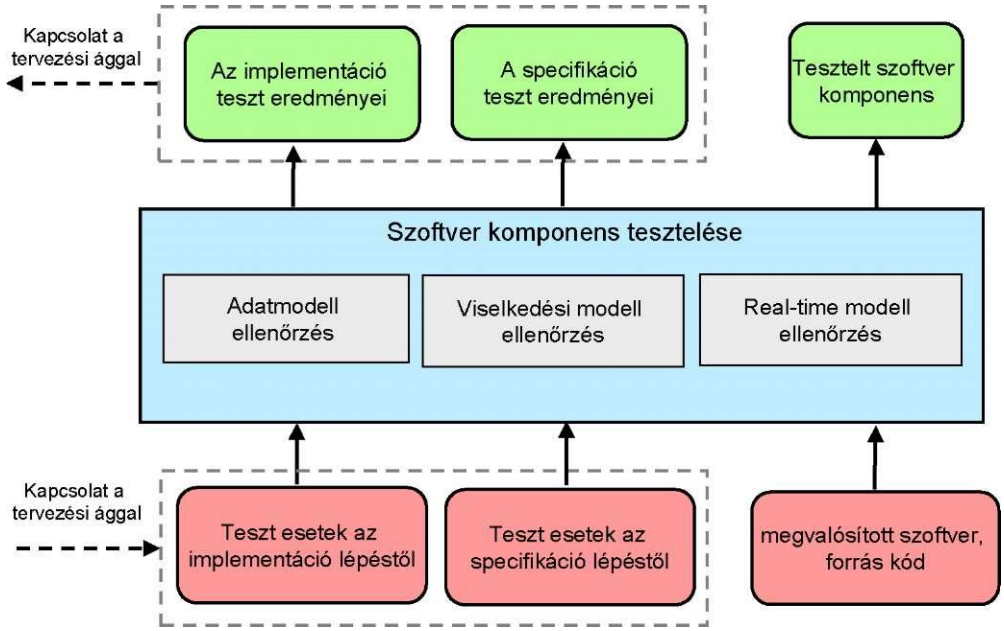
56. **Röviden foglalja össze, hogy a V-modell tesztelési ágánál melyik hierarchia szinten milyen jellegű tesztelési eljárásokat használnak! Mutasson példát minden hierarchia szintre!**

Az egyes V-modell szerint végrehajtott tesztelési lépések előtt az ábra egy gyors áttekintést ad a különböző szoftvertesztelési módszerekről és azok alkalmazási pontjáról az életciklus modellben.



57. **Mutassa be, hogy a *Szoftverkomponensek tesztelése* milyen főbb lépésekből áll!**

A szoftverkomponensek tesztelése tipikusan fehérdoboz módszerekkel történik. Ilyen módszer például a statikus kódanalízis az adatmodell ellenőrzésénél, a vezérlési gráf ellenőrzése a viselkedési modellnél.



58. **Mutassa be az szoftverkomponensek, rendszerek adatmodelljének tesztelésére leggyakrabban alkalmazott módszereket!**

Ez a lépés magában hordozza a tervezés során specifikált, a szoftverkomponens által végrehajtott adatfolyam helyességének ellenőrzését, valamint a komponens interfészeihez tartozó limit értékek és azok átlépésének tesztelését. Talán ennél érdekesebb az implementációt és nem a specifikációt ellenőrző *statikus analízis* rész.

A *statikus analízis* a kód futtatás nélküli kódelemzővel történő ellenőrzését jelenti és elsősorban az úgynevezett *runtime*, vagy futási hibák felderítésére szolgál. *Runtime* hibának nevezünk minden alább felsorolt a program futása közben bekövetkező hibát:

- Aritmetikai hibák: overflow, underflow, divide by zero...
- Pointer aritmetikai hibák
- Tömbtúlindexelések
- Függvény paraméter problémák

Az összes nagyobb cégnél szabály, hogy ezeknek a futási idejű hibáknak a kivédéséhez valamilyen eljárást kell alkalmazni. A MISRA-C az alábbi szabályt hozta ezzel kapcsolatban:

- A Run-time hibák minimalizálására minimum egyet az alábbi eljárások közül használni kell.
    - Statikus kód analizátor
    - Dinamikus kód analizátor
    - Explicit lekódolása a run-time hibák kezelésének

A cégek általában statikus kódanalizátort szoktak használni.

59. **Mutassa be az szoftverkomponensek, rendszerek viselkedési modelljének tesztelésére leggyakrabban alkalmazott módszereket!**

A viselkedési modell ellenőrzése során leggyakrabban alkalmazott tesztek az ún. kódfedettség vizsgálatok, ahol azt ellenőrizik a tesztelők, hogy program minden sora, feltétele, ága stb. végrehajtásra kerül-e (természetesen úgy, hogy a végrehajtás eredménye helyes).

A kódfedési vizsgálatokat az ún. vezérlési gráf alapján létrehozott tesztekkel szokták végezni. A tesztelési irodalom a következő fedési vizsgálatokat ismeri [Majzik István]:

- *Utasítások lefedése:* Minél több utasítást végrehajtani
- *Döntési ágak lefedése:* Elágazások esetén minden irányban továbbmenni
- *Feltételek lefedése:* Feltételes elágazásokban minél több feltétel kombinációt kipróbálni
- *Utak lefedése:* Minél több független utat bejárni a gráfban

A kódfedettségi vizsgálatok elvégzésére alapvetően két lehetőségünk van. Az egyik lehetőség egy pusztán szoftveres mérés a *kódfelműszerezésével.* Ez a megoldás memória és végrehajtási idő overhead-et jelent és a tesztelés után tipikusan nem szedhető ki a műszerezés, mert az megváltoztatná a real-time viselkedést. Ilyen szoftveres kódfedést mérő eszköz elvégzi a kód felműszerezését és a

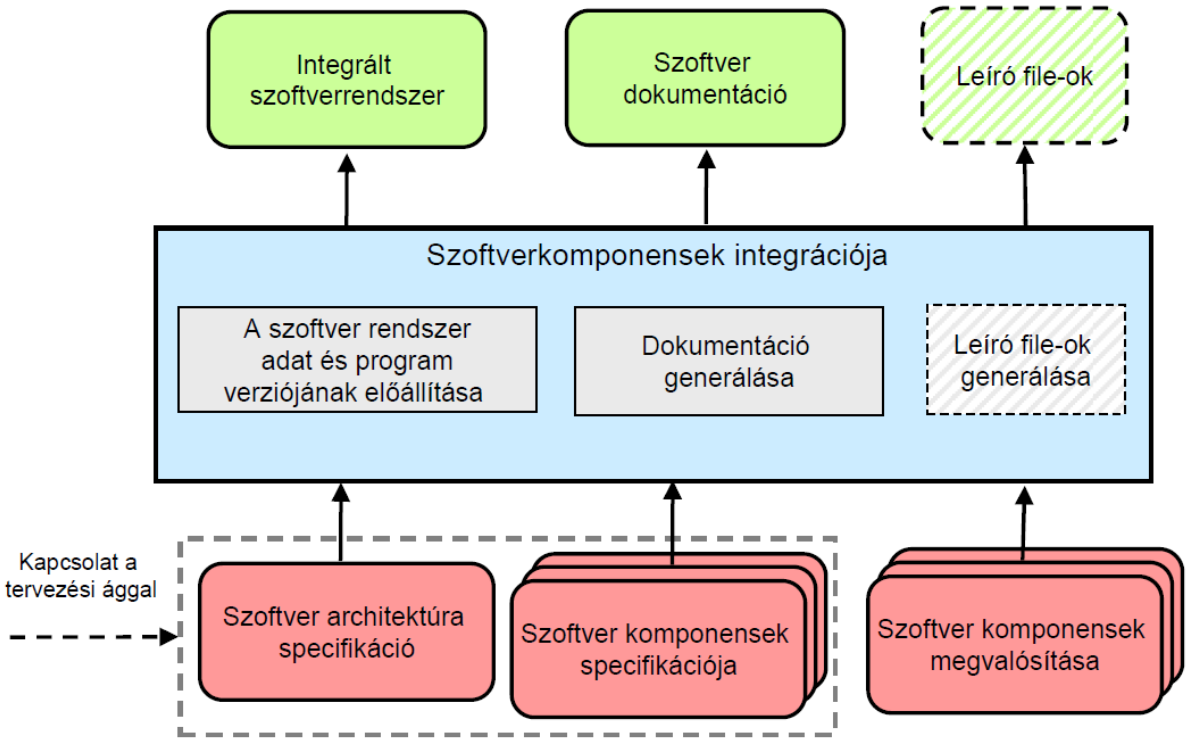
futtatás alatt készít egy logfile-t, amiből meg tudja határozni, hogy az egyes sorok hányszor futottak le.

A kódfedettségi vizsgálatok elvégzésének másik, hatékonyabb módja a hardware-es trace alkalmazása, hiszen ezek nem befolyásolják az adott rendszer működését. A harware-es megfigyelésre alapvetően két lehetőségünk van. Az egyik lehetőség cím és/vagy adatvonal figyelése, ami igen komplex hardware támogatást igényel így drága, további probléma ennél a módszernél, hogy a legtöbb modern mikrovezérlő integrált program és adatmemóriával rendelkezik, így nem lehet hozzáférni a cím és adatvezetékeikhez. A másik gyakorlatban jobban alkalmazott megoldás a beágyazott *trace* modulok használata. A legtöbb modern mikrovezérlőnél ugyanis magába a magba integrálnak olyan *trace* blokkokat, amelyek az ilyen jellegű real-time megfigyelést lehetővé teszik. Ezekre a hardware-es támogatásokra épülve komoly és drága eszközökkel kódfedési vizsgálatok és real-time viselkedést ellenőrző tesztrendszereket lehet felépíteni.

A kódfedettségi vizsgálatokkal kapcsolatban érdemes megjegyezni, hogy még profi tool- okkal és képzett csapattal is igen időigényes dolog. Ezért a bonyolultabb fedési vizsgálatokat még a nagy megbízhatóságú rendszereknél sem írják elő kötelező érvényűre.

60. **Mutassa be röviden, hogy a *Szoftverkomponensek integrációjánál* milyen lépéseket kell végrehajtani.**

Az egyes komponensek tesztelése után a szoftverintegráció lépés következik. Ez gyakorlatilag az első futtatható szoftver release létrehozását jelenti. Ennél a lépésnél néhány esetben az integrált szoftvert már az új hardware-en futtatják, amennyiben annak integrációja és tesztelése már lezajlott, de ugyanúgy jellemző az is, hogy az integrált szoftver még egy korábbi verziójú hardware-en (az iteráció során), vagy fejlesztőkártyán fut.



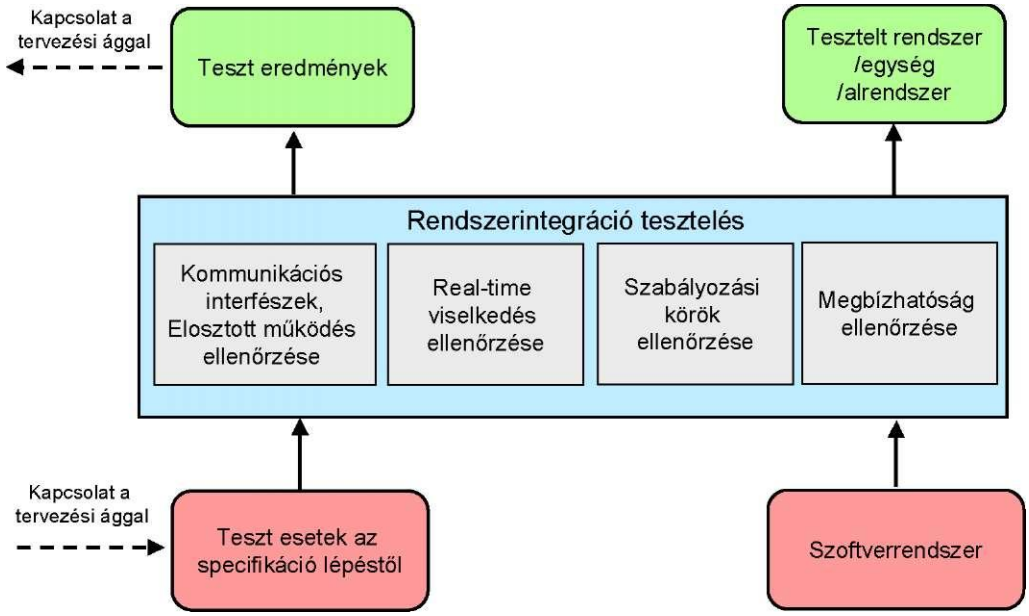
Az integrációs lépés elsődleges célja az integrált *szoftverrendszer előállítása,* de ez a folyamat sokszor kiegészül a *dokumentáció előállításával,* generálásával, illetve opcionálisan a különböző *leíró file-ok előállításával* is. A szoftverrendszer előállítása gyakorlatilag az egyes komponensek kódjának egy projectben való integrálását jelenti.

A dokumentáció generálásra jó példa a számos cégnél alkalmazott és a *coding style guide*- dokban rendszeresen hivatkozott Doxygen használata (a Doxygen használatához a 4.2 fejezetben kaphatunk útmutatót).

A különböző leíró file-ok előállítása már jóval alkalmazásfüggőbb és opcionális folyamat. Ezek a leírófile-ok a szoftver globális változóihoz tartalmaznak adatokat: a változók formátumát, memória címét, fizikai világgal való viszonyát (felbontás, ofszet, mértékegység stb.). A leíró file-ok információit a későbbi szürkedoboz tesztek, illetve kalibrációs lépések alatt használják fel. Ezekben a HIL (Hardware In the Loop) tesztekben file-oban megadott fontosabb szoftver paraméterek online nyomon követhetők, vagy akár módosíthatók kalibrálhatók is.

61. **Mutassa be a rendszerintegráció utáni *integrációs tesztelés* főbb lépéseit!**

Ennél a lépésnél a szoftver és a hardware már mindenképpen integrálódott, és itt ennek az integrációját tesztelik. A teszt célja az, hogy a rendszer külvilággal való kapcsolatát ellenőrizzék: megfelelő külső eseményekre megfelelő válaszokat ad-e.



Ezek a tesztek már tipikusan a szürkedoboz, feketedoboz teszt kategóriába tartoznak, tehát a szoftver forráskódja már egyáltalán nem szükséges hozzá. A legtöbb esetben ezeket a teszteket külön teszt centerekben a fejlesztéstől független csapatok hajtják végre. Egy ilyen tesztelés általában egy speciális tesztkörnyezetet igényel, amely segítségével a tesztelendő modul, vagy modulok környezetét szimulálni lehet. Ezeknél a teszteknél kapnak szerepet az leíró file-ok.

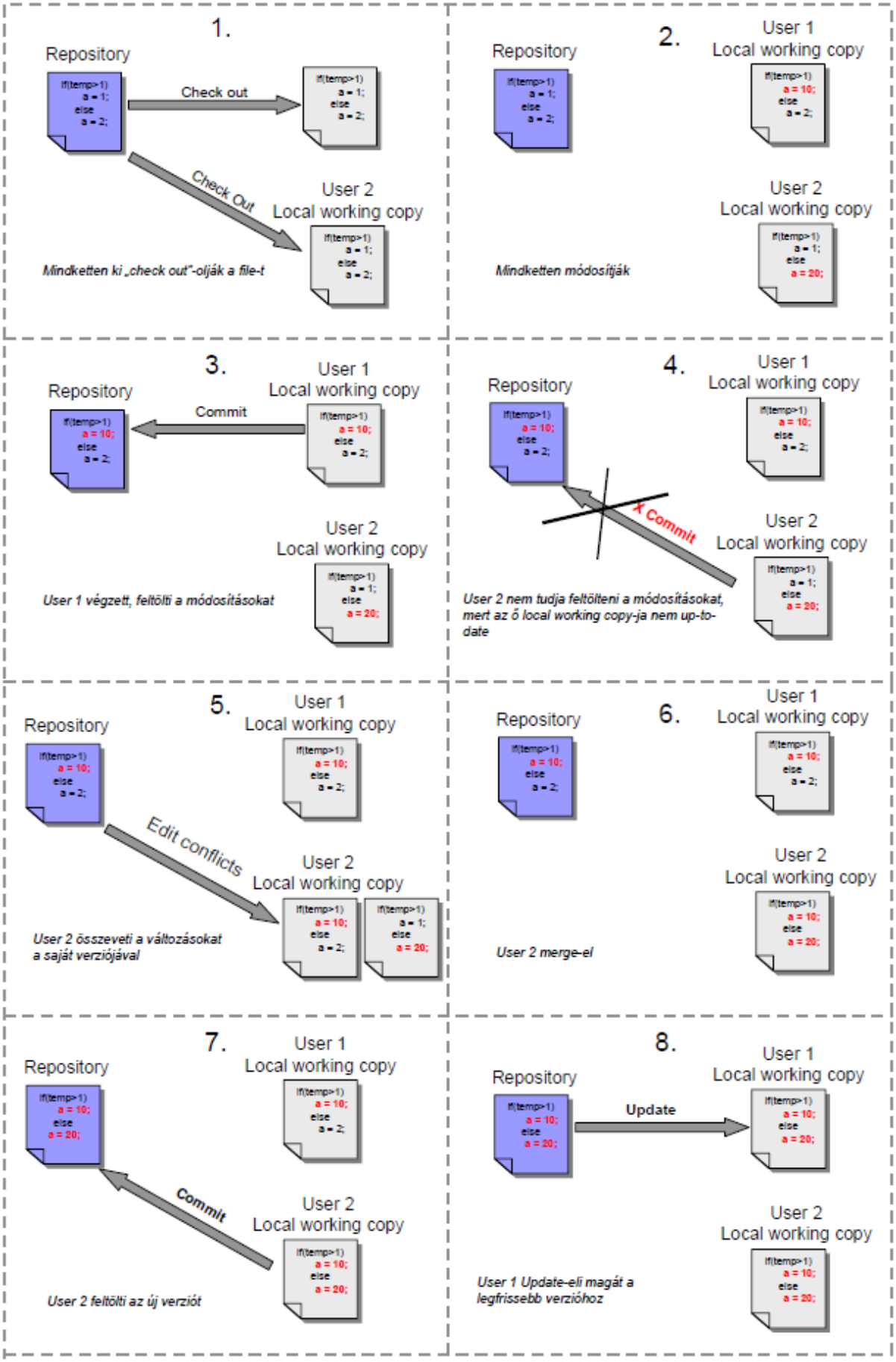
62. **Mi az az ISTQB (International Software Testing Qualifications Board)?**

Egy nagyobb cégnél kiemelten fontos a termék minőségbiztosítása és ennek az egyik kritikus lépése a termék tesztelése. Annak érdekében, hogy a lehető legnagyobb biztonságban érezzék maguk ezek a cégek igyekeznek tanúsítványokat szerezni a tesztelésük minőségéről. Jelenleg nincs olyan elterjedt tanúsítvány, amely magát a tesztelési folyamatot minősítené, viszont létezik egy szervezet az ISTQB (International Software Testing Qualifications Board) amely magukat a tesztelőket minősíti és a tesztelők az általuk letett vizsgákkal bizonyítják azt, hogy képesek az adott folyamat ellátására. A legtöbb nemzetközi cégnél a tesztelők egy jó része rendelkezik ilyen ISTQB vizsgával és adott esetben ez előnyt jelenthet a felvételinél is. Az ISTQB egy nemzetközi szervezet, amelynek Magyarországon létezik alszervezete. Az ISTQB vizsgákra léteznek hivatalos felkészítő tanfolyamok, illetve erre felkészítő könyvek és tesztsorok, a vizsgák angol nyelvűek, de a nemzeti szervezetek hatáskörébe tartoznak, Magyarországon is lehetséges ilyen vizsgát letenni nem túl magas költséggel.

63. **Soroljon fel legalább 5 darab SVN parancsot és ismertesse ezek működését!**

(lásd: 23-as kérdés, Add, Commit, Check out, Update, Merge)

64. **Rajzolja fel egy azonos programrészen dolgozó két fejlesztő konfliktusának és a konfliktus feloldásának folyamatát SVN parancsok segítségével!**



65. **Mi a *doxygen* és milyen problémákra próbál megoldást adni és hogyan?**

Ez egy nyílt forráskódú szoftver, mely leegyszerűsíti és automatizálja a projektdokumentációk készítését. A C++, C#, Java, Objective-C, Fortran és egyéb forrásokban található, speciális formátumú magyarázatok kinyerésével különböző kimenetek előállítására képes, a HTML-től a PDF-ig. Lényege, hogy a dokumentációt a kódba írjuk, ezért elég egyszerűen naprakészen tudjuk tartani. Sőt, az olvasó számára a keresztreferenciák megkönnyítik mind az olvasást, mind a megértést.
Igyekszik feloldani a komment-dokumentáció közötti inkonzisztenciát.
Speciális karakterek segítségével tudjuk megadni, hogy „észrevegye” a Doxygen generátor. Ezeket a szövegeket használja a dokumentumgenerálás során. Képes gráfokat készíteni, ezáltal grafikusan is láthatjuk az egyes részek, függvények, fájlok kapcsolatokat, hogy mi miből származik, mi mire mutat. Kiemeli és csoportosítja a kód részeit, mint például az enumerációkat, változókat, függvényeket vagy a define-okat. Továbbá az egyes dokumentációknál (pl a függvényeknél) leírja funkcióját, paramétereit, visszatérési értékét, valamint referenciáit és azt, hogy hol található a kódban. Így pontokba szedve áttekinthetővé, könnyen megérthetővé válik a kód. Minden egyes fájlra létrehoz egy ilyet, ezek között a kereszthivatkozások segítségével tudunk navigálni.

66. **Soroljon fel legalább 8 *doxygen* parancsot!**

- @file – A file dokumetációjának kezdete
- @brief – rövid ismertető leírás következik
- @author – Szerző tipikusan file elejére
- @version – Verzió megadás
- @date – Dátum megadása
- @warning – Figyelemfelhívás
- @fn – Függvény definítció, csak akkor van rá szükség, hogyha a komment nem közvetlenül a függvény előtt van
- @param – Függvény paraméter
- @return – Függvény visszatérési érték
- @typedef – Tipusdefinició dokumentálása
- @def <name> – Define ra hívja fel a figyelmet
- @arg <word> – Argumentum megadás
- @enum – Felsorolásos típus
- @struct – Struktúra dokumentáció
- @var – változó definició
- @ref – dokumentáción belüli hivatkozás

67. **Adja meg egy tipikus fájl fejlécének kommentezését *doxygen* segítségével!**

```
/** @file misra-c_pl1.c
* @brief MISRA-C demostration on aritmetic operations
* @author Balazs Scherer
* @version 1.0
* @date    2009-04-23
* @warning Only tested with AVRStudio
*/
```

68. **Adja meg egy tipikus függvény fejlécének kommentezését *doxygen* segítségével!**

```
/** @fn void UART1_Init(unsigned long baud_rate, void (*handler)(void));
 *  @brief An inicialisation function to redirect STDIO to UART1
 *  @param baud_rate: Baudrate in bit/sec
 *  @param handler: Callback function for receiving UART characters with IT
 *  @return nothing
*/
```

69. **Mire szolgál az *@addtogroup* parancs és miért fontos a dokumentáció elkészítése során, hogyan jelenik meg a dokumentációban? Mutasson példát rá!**

Csoportosításra jó.

```
/** @addtogroup Az én csoportom
@{
*/

......

......

......
/**
@}
*/
```

```
/** @brief  I2C Init structure definition  */
typedef struct
{
  uint32_t I2C_ClockSpeed;      /*!< Specifies the clock frequency */
  uint16_t I2C_Mode;            /*!< Specifies the I2C mode.
This parameter can be a value of @ref I2C_mode */
} I2C_InitTypeDef;
```

```
/** @defgroup I2C_mode
@{
*/
#define I2C_Mode_MASTER                 1
#define I2C_Mode_SLAVE                  0
/**
@}
*/
```

70. **Mutasson példát arra, amikor a C fordító működésének specifikálatlansága problémát tud okozni!**

A kiválasztott fordító egész osztásokra való reakcióját ellenőrizni és dokumentálni kell. (Például két ISO kompatibilis fordító is mutathat eltérő viselkedést negatív egész osztás esetében)

```
a = (-5/3);
/* a = -1 ahol a maradék -2 vagy
      a = -2 ahol a maradék +1              */
```

A bitmezők tárolási módját ellenőrizni és dokumentálni kell. A bitmezők implementálása a C egyik leghomályosabb területe. Nem specifikált, hogy a tároló területben a bitmezőket melyik oldalról kezdi feltölteni a fordító, illetve, hogy van-e átjárás a byte, szó határokon. Bár alapvetően a bitmezőket csak névvel hivatkozva illik kezelni, de ezeket a viselkedéseket így is célszerű dokumentálni.

Char típust csak karakter tárolásra szabad használni. A *singned* és *unsigned* verziót szabad szám értékre használni, mert a char megvalósítás implementációfüggő, char típusokon csak az =,==,!= műveletek értelmezhetőek.

```
char c = 200; int i = 1000;
printf("i/c = %d\n\r",  i / c ) ;   / *  Az eredmény lehet 5 is és -13 is */
```

71. **Mutasson példát arra, amikor ugyanazt a kódot, ami egy 32 bites vezérlőn hibátlanul futott egy 8 bites vezérlőre lefordítva és azon futtatva a végrehajtásnál hibát okoz, vagy okozhat!**

```
uint16_t u16a = 40000;
uint16_t u16b = 30000;
uint32_t u32x;
U32x = u16a + u16b; /* u32x = 70000 vagy 4464? */
```

A 32 bites kontroller regiszterei 32 bitesek, ezért nem csordul túl az összeadáskor, 8 bites kontrollernél 16 biten fog összeadni, így túlcsordul és a rossz végeredményt már hiába tárolod le az u32x változó 32 bites memória területén.

72. **Mutasson példát arra, amikor ugyanazt a kódot, ami egy 8 bites vezérlőn hibátlanul futott egy 32 bites vezérlőre lefordítva és azon futtatva a végrehajtásnál hibát okoz, vagy okozhat!**

```
uint8_t port = 0x5aU;
uint8_t result_8;
uint16_t result_16;
uint16_t mode;
Result_8 = (~port) >> 4; /* Nem engedélyezett */
```

A 8 bites kontroller azt csinálja amit várunk: negálja a 8 bites változót 01011010 ==> 10100101 majd eltolja jobbra 10100101 ==> 00001010, így a felső 4 bit helyére 0 kerül.
A 32 bites kontroller mivel nem kasztoltunk ezért 32 bites regiszterekkel dolgozik. Negálja a port-ot 32 biten, vagyis a felső 24 bit ami ugyebár csupa nulla, a negálás után csupa egyes lesz és léptetéskor ezek az egyesek csúsznak le az alacsonyabb helyértékekre. A végeredmény persze 8 bites így csak 8

bit tárolódik le a memóriában, de az a nyolc bit az előbbi 00001010 helyett most 11111010 lesz.
Megoldás:

```
result_8 = ((uint8_t)(~port)) >> 4; /* Engedélyezett */
result_16 = ((uint16_t)(~(uint16_t)port)) >> 4; /* Engedélyezett */.
```