Rendszertervezés (Embedded System Design)

VIMIM238

BME MIT: MSc beágyazott információs rendszerek szakirány

Óravázlat 2010

Csak belső használatra

Scherer Balázs

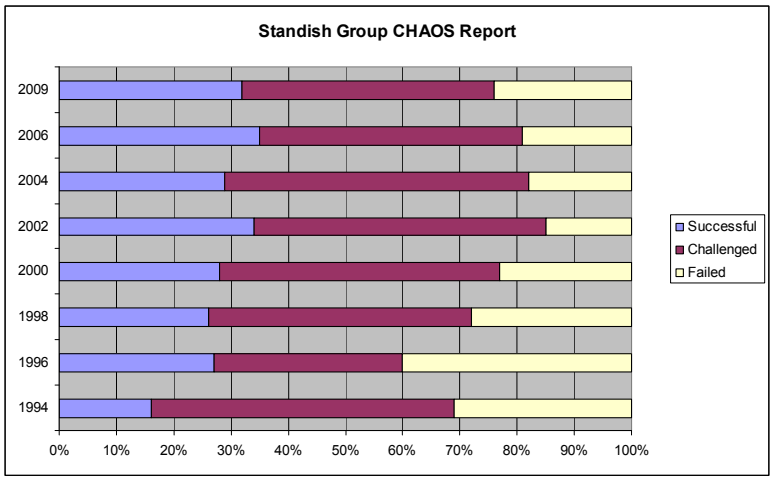
Lektorálta: Csordás Péter
Javítási ajánlások a v2.3-hoz: Szegő Márton

1. Bevezetés

1.1. Miért van szükségünk valamilyen rendszerfejlesztési módszertanra?

Egy módszertan segítségével fejlesztett rendszer, vagy szoftver fejlesztési költsége viszonylag magas kezdő értékről indul, de a fejlesztési munka és a bonyolultság előrehaladásával csak közel lineárisan nő, míg egy módszertan nélkül létrehozott rendszer vagy szoftver költsége bár 0-ról indul, de a bonyolultság növekedésével exponenciálisan nő, ezzel szinte garantálva, hogy egy nagyobb projekt kifut az időből, és túllépi a költségkeretet.

Érdemes megnézni a Standish Group által a szoftver projectekkel kapcsolatban végzett felmérés eredményét 1.1 ábra (Nem csak beágyazott projekteket tartalmaz a felmérés). A *Successful* az időben befejezett sikeres projekteket jelenti. A *Challenged* azokat, amik valamilyen nehézséggel küzdöttek (pl. költség keret- vagy időtúllépés) de végül sikeresen zárultak, a *Failed* pedig a fejlesztés közben abbahagyott projecteket jelzi.



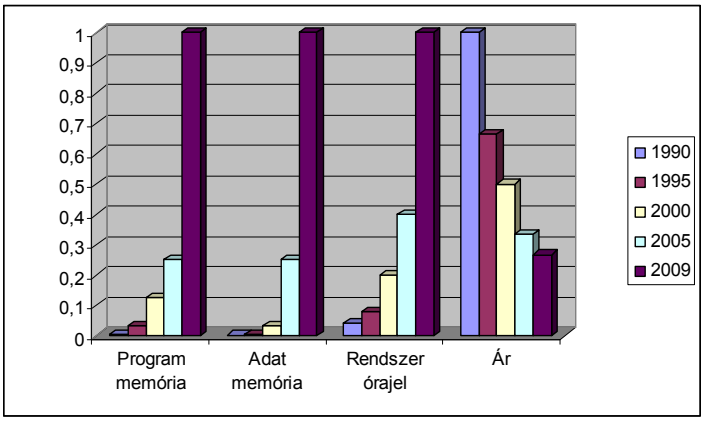
	1994	1996	1998	2000	2002	2004	2006	2009
Successful	16%	27%	26%	28%	34%	29%	35%	32%
Challenged	53%	33%	46%	49%	51%	53%	46%	44%
Failed	31%	40%	28%	23%	15%	18%	19%	24%

***1.1 ábra**. A Standish Group [1] felmérése a szoftver projektek eredményességéről*

Jól látható, hogy az 1994-es évhez képest a 2000-es években már érzékelhető javuláson ment keresztül a szoftverfejlesztői piac. Érdemes megjegyezni, hogy a cégeknél a különböző rendszerfejlesztési modellek elsősorban az 1990-es évek második felétől kezdtek elterjedni. Például az ISO9001 minőségbiztosítási szabvány 1996-ban adták ki, az általunk sokat hivatkozott V-modell első verziója pedig 1997-ben került publikálásra. Ezekre az évekre tehető a *képesség-érettség* modellek (CMM: Capability Maturity Modell) alkalmazásának bevezetése is. Igaz ez annak ellenére, hogy rendszerfejlesztési életciklus egyik meghatározó modelljét a *vízesés modell*-t már 1970-ben publikálták.

Észrevehető ugyanakkor, hogy a 2000-es évekre csak stagnálás jellemző, gyakorlatilag nincs előrelépés. Mi indokolja azt, hogy az elméletileg fokozatosan fejlődő projekt végrehajtási módszertanok és apparátus mellett sem nőtt az eredményes projektek aránya? A válasz

megértéséhez elég, ha megnézzük a beágyazott rendszerek fejlődési ütemét az elmúlt pár évben 1.2 ábra. (hasonló tendencia igaz az asztali számítógépes szoftver projecteknél is).



***1.2 ábra**. A beágyazott vezérlők jellemzőinek változása az 1990-es évektől napjainkig (az y tengely mindig a legmagasabb értékhez vett arányokat mutatja)*

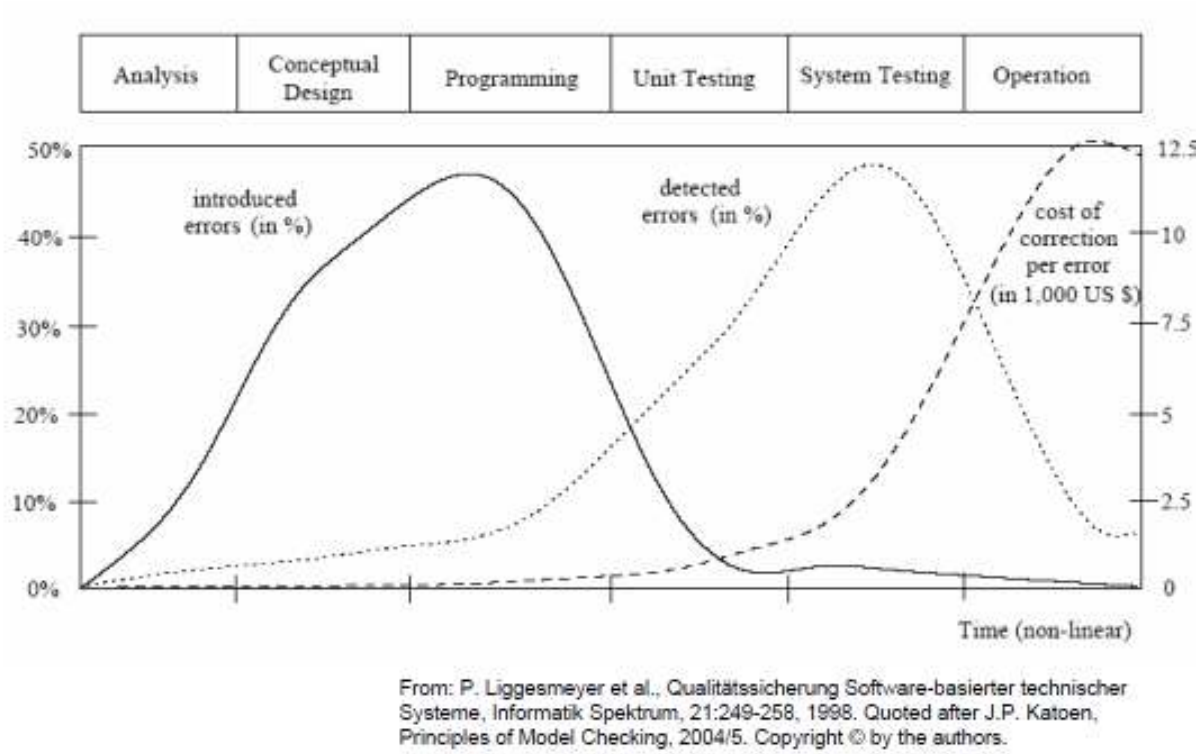
A számszerű adatokat érzékelteti az 1. táblázat, ami bemutatja egy tipikus autóipari ECU-ban (Electronic Control Unit) alkalmazott feldolgozó egység változásait (a bemutatott táblázat nem egy vezérlőtípus részletes eredménye, hanem több ilyen táblázat összefoglalása). [1] [2]

1. táblázat. *Egy tipikus erőátviteli rendszerben alkalmazott autóipari ECU feldolgozó egységének változása az elmúlt 20 évben*

	Adatszélesség	Program memória	Adat memória	Órajel
1985	8 bit	8 kbyte	128 byte	4 MHz
1990	8 bit	64 kbyte	256 byte	8 MHz
1995	32 bit	256 kbyte	2 kbyte	20 MHz
2000	32 bit	512 kbyte	16 kbyte	40 MHz
2005+	32 bit	2+ Mbyte	64+ kbyte	100+ MHz

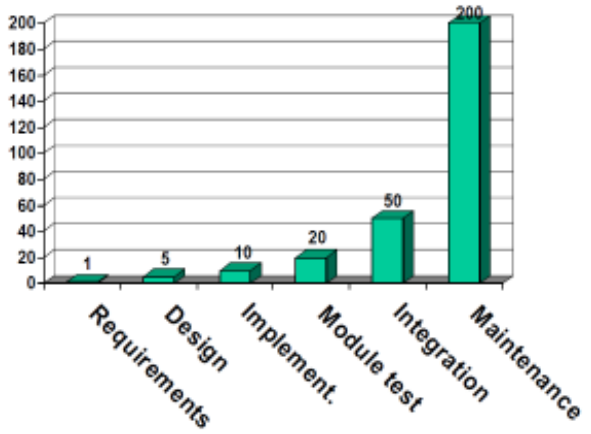
Az 1.2 ábra és az 1. táblázat segítségével tehát könnyen levonhatjuk azt a következtetést, hogy az elmúlt évtizedben bár csak szinten maradt a sikeres projektek aránya, de ezt annak ellenére sikerült elérni, hogy a projectek komplexitása és mérete közben minimum megnégyszereződött. Ez így már nem rossz 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.

De miben is tud segíteni egy rendszertervezési módszertan, és egy rendszer tudatos tervezése? Ha megnézzük a 1.3 ábrát, akkor információt kaphatunk arról, hogy egy project során átlagosan hol vétünk hibákat, hol fedezzük fel őket és mennyi energiába, illetve anyagi ráfordításba kerül azok kijavítása. Az 1.4 ábrán a hibajavítás költségét láthatjuk számszerűsítve.



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.

***1.3 ábra**. Egy projekt során elkövetett hibák eloszlása, a hibák megtalálása, és a javítás költsége [4].*



***1.4 ábra**. A hibajavítás költségének arányai egy másik forrásból [4]*

Az 1.3 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.

A fejlesztési modellek használatának másik nyomós indoka, hogy a mai világban nagyon sok termék több cég kooperációjában készül el. Ezekben az esetekben egyrészt az egyes cégek elvárják a partnertől, hogy igazolni tudják, azt hogy a saját project részük időre és a megfelelő minőségben fog előállni. Másrészt fejlesztés közben is sok esetben szükség van együttműködésre, és ezt a kooperációt nagyban megkönnyíti az azonos fejlesztési módszerek és metódusok használata.

1.2. Minőségi szabványok

Az előző fejezet rávilágított arra, hogy miért lehet fontos egy fejlesztési módszertant használni a rendszereink tervezésénél. Ugyanakkor felmerülhet bennünk az a kérdés, hogy miért a fejlesztési folyamatokkal foglalkozunk és miért nem magával a késztermékkel, aminek a minőségét garantálni akarjuk. Erre az a válasz, hogy bár voltak és vannak rá kísérletek (például az ISO-9126 szabvány), de például egy szoftvernél, mint késztermeknél igen nehézkes megállapítani a minőségét. Fokozottan igaz ez egy beágyazott szoftverre. Ezért egyszerűbb és hatékonyabb, ha a termék – legyen szó hardverről vagy szoftverről – fejlesztésének mikéntjét vizsgáljuk.

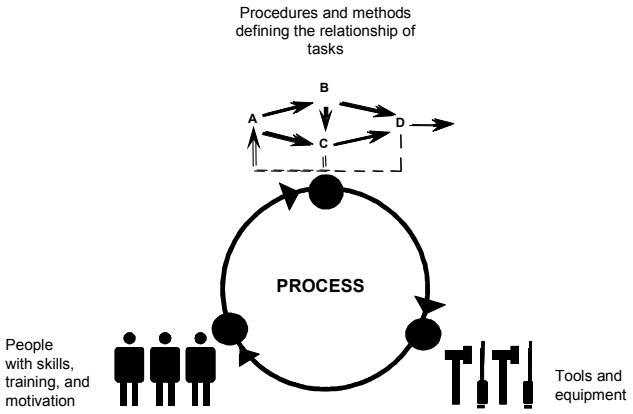
Nézzük meg, hogy magával a fejlesztési folyamat minősítésével milyen nemzetközi szabványok foglalkoznak.

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.
A Capability *Maturity Model® Integration (CMMI)* és a *SPICE* (Software Process Improvement and Capability dEtermination) szűkebb, speciálisan az informatikai/szoftver rendszerek fejlesztésére kidolgozott szabványok, amelyek elsősorban az ilyen tevékenységgel foglalkozó vállalatoknál lettek egyre népszerűbbek az elmúlt években. A legtöbb informatikai vagy szoftver tevékenységgel rendelkező vállalat az ISO9001-es minősítés mellett a CMMI vagy SPICE minősítéssel is rendelkezik.
A két szabvány távolról nézve erős hasonlóságokat tartalmaz (a kettő célja is közel azonos). A SPICE a CMMI alapját adó CMM képességi-érettségi modellekkel együtt a 90-es évek közepén-végén jelent meg, és elsősorban az egyes rendszertervezési folyamatok *képességi szintjeinek* elemzésével foglalkozik. A később megjelenő CMMI for Development egy integrálása a 90-es években megjelent *képességi-érettségi* modelleknek (CMM), és sok fogalmat átvesz a SPICE-ból is.

1.3. Capability Maturity Model Integration (CMMI)

1.3.1. Bevezetés, a CMMI filozófiája

Az 1.3-as fejezetben a CMMI for Development [5],[6] minket érintő részeinek rövid összefoglalása következik.
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.



***1.5 ábra**. Egy project sikerességét befolyásoló tényezők*

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 SPICE használja még ezt a *folytonos* megközelítést).
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 implementálni (A régebbi CMM-ekre volt ez jellemző).
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. Ezekről a későbbiekben lesz szó.

1.3.2. A lépcsőzetes megközelítés érettségi szintjei

A CMMI lépcsőzetes megközelítése az egész szervezetre vonatkozóan 5 érettségi szintet specifikál.

1. szint: ***Initial (kezdeti)***

2. szint: ***Managed (menedzselt/irányított)***

3. szint: ***Defined (meghatározott)***

4. szint: ***Quantitatively Managed (mennyiségileg menedzselt/irányított)***

5. szint: **Optimizing** ***(optimalizáló)***

Ezek jelentése a következő:

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.

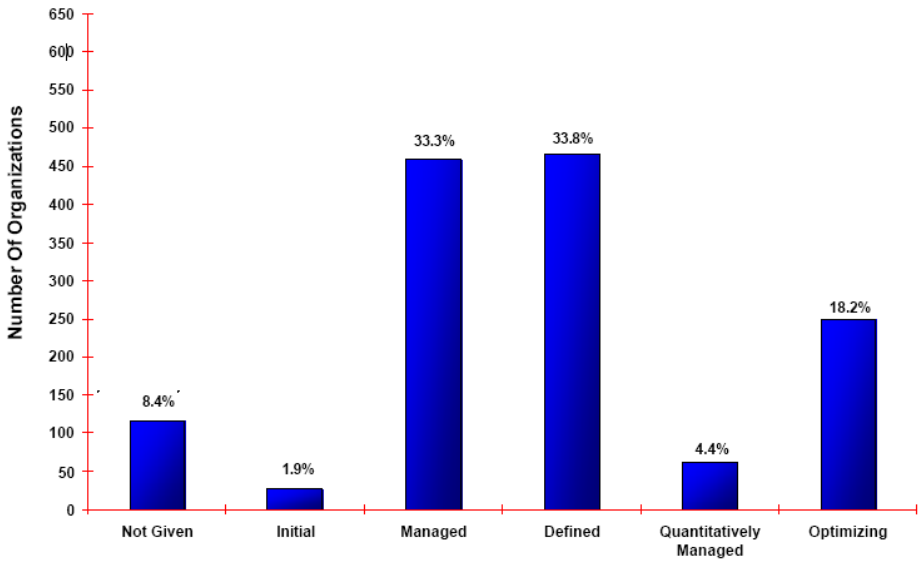
4. Quantitatively Managed (mennyiségileg menedzselt/irányított)

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.

Érdemes megnézni, hogy egy 2006-os statisztika szerint a világon minősítést kapott vállalatok hány százaléka tartozik az egyes kategóriákba (1.6. ábra).



***1.6 ábra**. Az egyes érettségi szintek eloszlása egy 2006-os felmérés alapján.*

Érdemes megjegyezni, hogy a szervezetek méretének növekedésével párhuzamosan nő az *Optimalizáló szint* elérésének aránya is. Míg az 50-100 főt foglalkoztató szervezeteknél 10% körüli az *Optimalizáló szint* aránya addig az 1000 fő feletti szervezeteknél ez 40% fölött van. Szintén az érettségi szintekhez tartozik, hogy távolról sem olyan könnyű ezeket elérni. Statisztikák szerint az egyes érettségi szintek közötti fejlődés minimum 1 évet, de sokszor 2 évet is igénybe vesz.

1.3.3. A folytonos megközelítés képességi szintjei

A CMMI folytonos megközelítése az egyes folyamatokra vonatkozóan 6 *képességi* szintet specifikál. Ezek a következők:

0. szint: ***Incomplete (nem teljes, befejezetlen)***

1. szint: ***Perfomed (végrehajtott)***

2. szint: ***Managed (menedzselt/irányított)***

3. szint: ***Defined (meghatározott)***

4. szint: ***Quantitatively Managed (mennyiségileg menedzselt/irányított)***

5. szint: **Optimizing** ***(optimalizáló)***

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.

1.3.4. A CMMI folyamatai és a kapcsolat a lépcsős és a folytonos megközelítés között

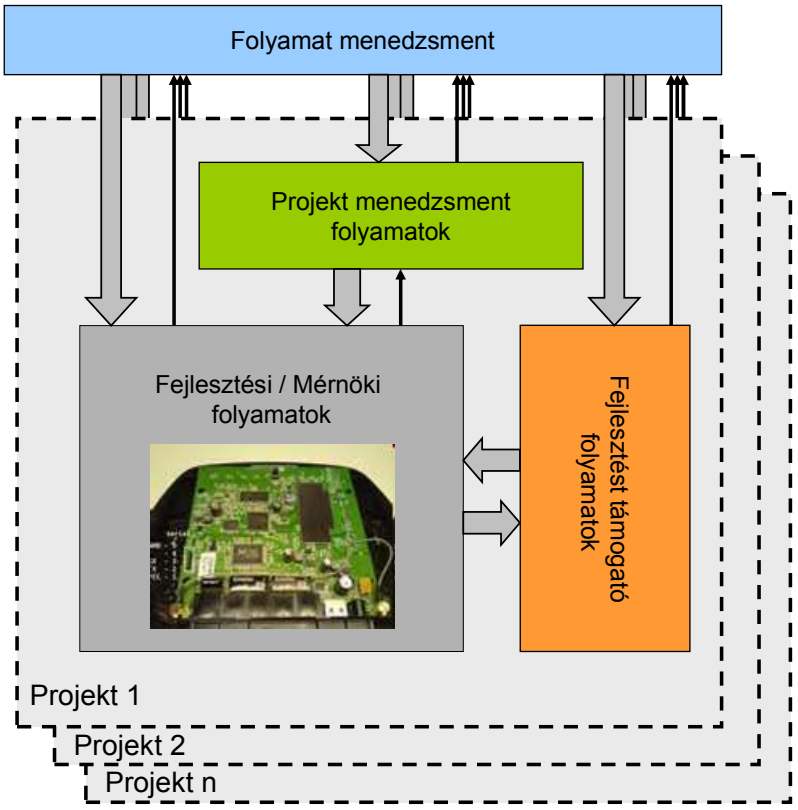
Az előző két alfejezetben arról beszéltünk, hogy hogyan értelmezzük egy szervezet *érettségét*, és a szervezet folyamatainak *képességét*, de nem említettük meg, hogy a CMMI szerinti szemléletben egy rendszertervezési projekt milyen folyamatokból áll. Ez a fejezet ismerteti a CMMI által definiált folyamatokat, és bemutatja, hogy az egyes *érettségi* szintekhez milyen folyamatok alkalmazása kötelező.

A CMMI for Development 22 folyamatot azonosít, ezek a következő nagy csoportokba oszthatóak:

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

Ezeket az 1.7 ábra próbálja meg csoportosítani és logikailag összekapcsolni (Az ábra nem minden esetre teljesen helytálló).

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



***1.7 ábra**. A rendszertervezés CMMI szerinti folyamat osztályai*

A folyamat csoportok bemutatása után következzen az egyes folyamatok felsorolása, majd értelmezése. A folyamatok felsorolására használt *2. táblázat* egyben tartalmazza azt is, hogy az egyes CMMI-ban definiált érettségi szintekhez a szervezeteknek milyen folyamatokat kell implementálniuk, illetve ezeket milyen *képességi szinten* kell megvalósítani.

A tárgy során természetesen nem ismertetjük az összes folyamatot részletesen, hanem elsősorban csak a mérnöki fejlesztési folyamatokra koncentrálunk, kiegészítve azokat néhány a mérnöki életben megkerülhetetlen támogató folyamattal.

A folyamat neve	*A folyamat típusa*	*Érettségi szint Maturity Level*	*CL1*	*CL2*	*CL3*	*CL4*	*CL5*
Requirements Management (Követelménymenedzsment)	Mérnöki / fejlesztési	2	**Érettségi szint 2**				
Project Planning (Projecttervezés)	Projekt menedzsment	2					
Project Monitoring and Control (Projectkövetés és vezérlés)	Projekt menedzsment	2					
Supplier Agreement Management (Beszállítói megállapodás menedzsment)	Projekt menedzsment	2					
Measurement and Analysis (Mérés és Analízis)	Fejlesztést támogató	2					
Process and Product Quality Assurance (Folyamat és termék minőség biztosítás)	Fejlesztést támogató	2					
Configuration Management (Konfigurációmenedzsment)	Fejlesztést támogató	2					
Requirements Development (Követelményfejlesztés)	Mérnöki / fejlesztési	3	**Érettségi szint 3**				
Technical Solution (Műszaki megoldás)	Mérnöki / fejlesztési	3					
Product Integration (Termék Integráció)	Mérnöki / fejlesztési	3					
Verification (Verifikáció)	Mérnöki / fejlesztési	3					
Validation (Validáció)	Mérnöki / fejlesztési	3					
Organizational Process Focus (Szervezeti szintű folyamatszemlélet)	Folyamat menedzsment	3					
Organizational Process Definition +IPPD* (Szervezeti folyamatok meghatározása)	Folyamat menedzsment	3					
Organizational Training (Szervezeti szintű képzés)	Folyamat menedzsment	3					
Integrated Project Management +IPPD* (Integrált projektmenedzsment)	Projekt menedzsment	3					
Risk Management (Kockázatmenedzsment)	Projekt menedzsment	3					
Decision Analysis and Resolution (Döntés elemzés és döntés hozatal)	Fejlesztést támogató	3					
Organizational Process Performance (Szervezeti szintű folyamatmenedzsment)	Folyamat menedzsment	4	**Érettségi szint 4**				
Quantitative Project Management (Mennyiségi projektmenedzsment)	Folyamat menedzsment	4					
Organizational Innovation and Deployment (Szervezeti szintű innováció és közzététel)	Folyamat menedzsment	5	**Érettségi szint 5**				
Causal Analysis and Resolution (Oksági elemzés és megoldás)	Fejlesztést támogató	5					

*IPPD: Integrált folyamat és termék menedzsment

2. táblázat. A CMMI folyamatai és az érettségi szintek kapcsolata

1.3.5. A CMMI folyamatainak rövid ismertetése

Aláhúzva a későbbiekben picit részletesebben ismertetett folyamatok.

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.

Project Monitoring and Control (Projectkövetés és vezérlés) ML2

A projectfolyamatok végzésének ellenőrzése, eltérés esetén korrekció, hibajavítás. Feladatai többek között a költség és az ütemterv összevetése a tervekben szereplőkkel a meghatározott mérföldköveknél. Javításra általában csak jelentős eltérés esetén van szükség.

Supplier Agreement Management (Beszállítói megállapodás menedzsment) ML2

Általában csak azokban az esetekben szükséges, ha valamilyen hardware komponens is átadásra kerül, de alkalmazhatják a fejlesztésnél használt termékek beszerzésére is. A folyamat célja a termék típusok és a beszállítók kiválasztása, megállapodás a beszállítókkal valamint a beszerzett alkatrészek ellenőrzése és beleintegrálása a termékbe.

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.

Process and Product Quality Assurance (Folyamat és termék minőség biztosítás) ML2

Cél az objektív betekintés biztosítása a folyamatokba és a munkatermékekbe. Ezek által lehetőség van az egyes végrehajtott folyamatokat, termékeket és szolgáltatásokat értékelni. Tipikusan valamilyen projekttől független minőségbiztosítási csoport végzi a szervezeten belül.

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.

Requirements Development (Követelmény fejlesztés) ML3

Termék és a termékkomponens követelményeinek létrehozása. Célja az ügyféllel együttműködve a követelmények összegyűjtése, elemzése, jóváhagyása. A követelmények nem csak magára a termékre vonatkozhatnak, hanem annak előállítási módjára, az alkalmazott

metódusokra is. A folyamat egyik fő célja, hogy ezt a követelmény -gyűjtést, -elemzést és az ügyfélnek való megfelelést fokozatosan finomítsa, fejlessze.

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.

Organizational Process Focus (Szervezeti szintű folyamatszemlélet) ML3

Célja a szervezet folyamatainak feltérképezése és szervezeti szintű fejlesztése. A folyamatok magukba foglalják a szervezet szabványos folyamatait, és az ezekből testre szabott konkrét projectekre jellemző folyamatokat is. A folyamatok javítása figyelembe veszi a mérések eredményeit. Ezért a tevékenységért tipikusan egy külön csoport felel, akik a fejlesztésbe bevonják az adott folyamatok végrehajtóit is, és elkészítik a folyamatfejlesztési terveket.

Organizational Process Definition (Szervezeti folyamatok meghatározása) ML3

A szervezeti folyamattapasztalatok összegyűjtése, a széles körben használható részek közzététele és karbantartása a célja. Segítséget nyújt a folyamatfejlesztéshez. Gyakorlatilag az így előálló adatbázis tartalmazza az összes tapasztalatot arról, hogy a szervezetnél milyen módszerek szerint alkalmazzák, készítik el az egyes folyamatokat. Útmutatót ad arra vonatkozva is, hogy ezeket az általános tapasztalatokat hogyan szabják testre az egyes projekteknél.

Organizational Training (Szervezeti szintű képzés) ML3

Célja a dolgozók szakértelmének és tudásának fejlesztése. Felméri az igényeket, megszervezi a tréningeket, begyűjti a visszajelzéseket.

Integrated Project Management (Integrált project menedzsment) ML3

Célja az egyes projectek létrehozása, és a projectben közreműködők kiválasztása. Meghatározza a project folyamatait a szervezeti adatbázis alapján. Meghatározza a project költség és időkeretét, valamint ütemtervét.

Risk Management (Kockázatmenedzsment) ML3

Célja a potenciális problémák azonosítása, mielőtt azok gondot okoznának. Az egyes kockázatok azonosítása után azokra kezelési stratégiát kell meghatározni, azt be kell vonni a termék életciklusába. A külső és a belső kockázatokat és azok hatását az ütemtervre és a költségre is mérlegelni kell.

Decision Analysis and Resolution (Döntéselemzés és döntés hozatal) ML3

Célja az egyes döntési helyzetekhez formális módszer nyújtása. Létrehozza az általános döntési kritériumokat a kiértékeléshez. A folyamat feladatai közé tartozik az alternatív megoldások azonosítása, valamint az egyes alternatívák kiértékelése. Alkalmazása elsősorban

műszaki döntéseknél jellemző. Ilyen döntések a beszállítók, az újra felhasználható modulok, a fejlesztőkörnyezetek kiválasztása.

Organizational Process Performance (Szervezeti szintű folyamatteljesítmény) ML4

Az egyes folyamatokhoz tartozó teljesítményjellemzők meghatározása, gyűjtése, statisztikai értékelése. Az egyes projectekre becslés adása a meglévő szervezeti statisztikából.

Quantitative Project Management (Mennyiségi project menedzsment) ML4

A project mennyiségi céljainak meghatározása. A project egyes szakaszaihoz mérőszámok rendelése, a valóság és a tervezett mérőszámok alakulásának követése. A statisztikák begyűjtése és elemzése, az eltérések okainak vizsgálata.

Organizational Innovation and Deployment (Szervezeti szintű innováció és közzététel) ML5

Összegyűjti a továbbfejlesztési javaslatokat és elemzi azokat. A hasznos javaslatok kiszűrése és prototípus bevezetése. A sikeres prototípus után a változtatások közzététele, és a továbbfejlesztés menedzselése.

Causal Analysis and Resolution (Oksági elemzés és megoldás) ML5

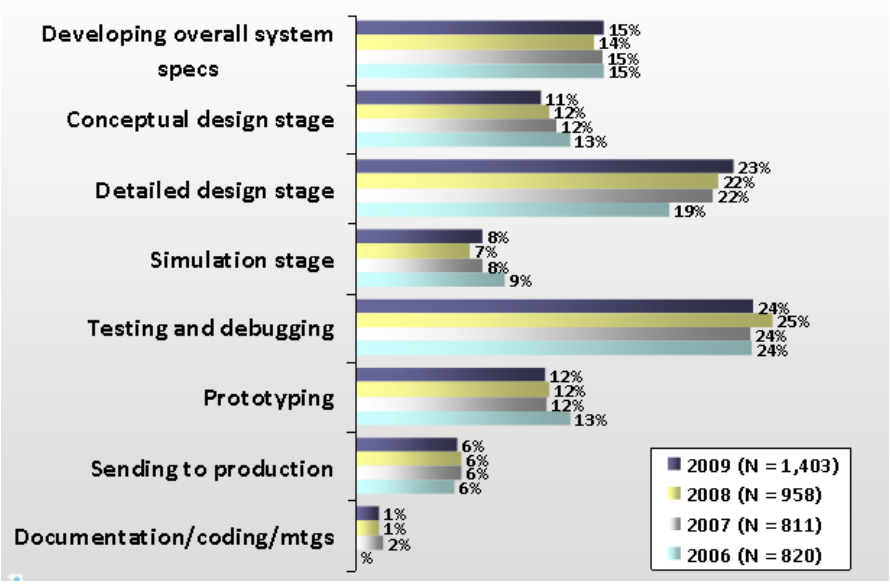
A hibaadatok begyűjtése és elemzése, az okok feltárása. Annak ellenőrzése, hogy az adott hiba csak az adott projectre jellemző, vagy általános probléma. A feltárt okok megszüntetése.

1.3.6. A CMMI modell auditálása a SCAMPI

A SCAMPI (Standard CMMI Appraisal Method for Process Improvement) felméréseknek három típusát különböztetjük meg: A, B, C. Az A típusú, formális SCAMPI felméréseket a hivatalos, SEI által elfogadott vezető auditorok végezhetik. Ez a legformálisabb módszer és csak e módszer alapján szabad érettségi vagy képességi szintet megállapítani. A B típusú felmérést leginkább az A-t megelőzően szoták alkalmazni előzetes feméréként. Ekkor azt mérik fel, hogy mire van még szükség ahhoz, hogy egy SCAMPI A típusú felmérésen is minden követelménynek megfeleljen a szervezet. A C típusú felmérések bevezető jellegűek, melyek során azt mérik fel, hogy mi lehet egy reális cél az adott szervezet számára, mely folyamatait fejlessze. Rendszerint egy C típusú (gap-analysis)-el indul a folyamatfejlesztés.

1.3.7. Egy beágyazott project fejlesztése során az egyes feladatokra fordított idő

A 1.3.5 fejezetben bemutatott folyamatok száma és leírása után az az érzésünk támadhat, hogy az eddigi tanulmányok során elsajátított részek igencsak kis hányadát teszik ki a felsoroltaknak. A kérdés az, hogy a valóságban egy project végrehajtásánál melyik folyamat mekkora súllyal szerepel. Erre a legjobb választ talán az Embedded Market Study 2009-ben [] készített statisztikája nyújtja (1.8 ábra)



***1.8 ábra**. A Embedded Market Study 2009 statisztikája az egyes project tevékenységek erőforrás igényéről*

Ebből a statisztikából kitűnik, hogy a tényleges programozás a fejlesztés csak igen kis hányadát jelenti egy project elkészítésének, és bármilyen fájdalmas is a többi 99%-98% munkát is mérnökök végzik. (A statisztikában a nem mérnöki feladatok nincsenek benne).

2. A fejlesztést támogató és menedzsment folyamatok

Ebben a fejezetben röviden összefoglalásra kerülnek az általunk használni kívánt fejlesztést támogató és menedzsment folyamatok.

2.1. Project menedzsment

A *project menedzsment* egyik legfontosabb feladata, hogy a project időbeli lefutását megtervezze. Azonosítsa az egyes végrehajtandó feladatokat, felmérje azok időbeli és erőforrás költségét, allokálja az egyes feladatok végrehajtásához szükséges emberi és tárgyi erőforrásokat. Felmérje a project futása alatt párhuzamosan végezhető feladatokat, azonosítsa az időkritikus feladatsorokat, ágakat. A project tervezés során fontos olyan mérföldkövek megállapítása, amely pontokon a project állása, haladása felmérhető. Ilyen mérföldköveknél belső, vagy külső ellenőrzéseket (*audit*) szoktak tartani. A project tervezés foglalkozik még a project céljának és termékeinek a kijelölésével is, valamint a project kockázatainak azonosításával is. A tervezés legfontosabb lépései összefoglalva:

- 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
- Projektkövetés a terv alapján
  - Projecttervezési paraméterek követése
  - Kockázatok követése
  - Adatmenedzsment követése
  - Mérföldkő ellenőrzések.
- Javítás
  - Problémák elemzése
  - Javító intézkedések megtétele és menedzselése

A projekt időtervének elkészítésére a legtöbb helyen valamilyen CAD programot használnak, ilyen például a Microsoft Project, valamint egy ingyenesen letölthető projekttervező program az Openproj [2.1].
Ezek a projekttervezők tipikusan a feladatok és azok időigényének, költségének bevitelét támogatják. Ezekből az adatokból tudnak valamilyen szemléletes, átfogó megjelenítést létrehozni.

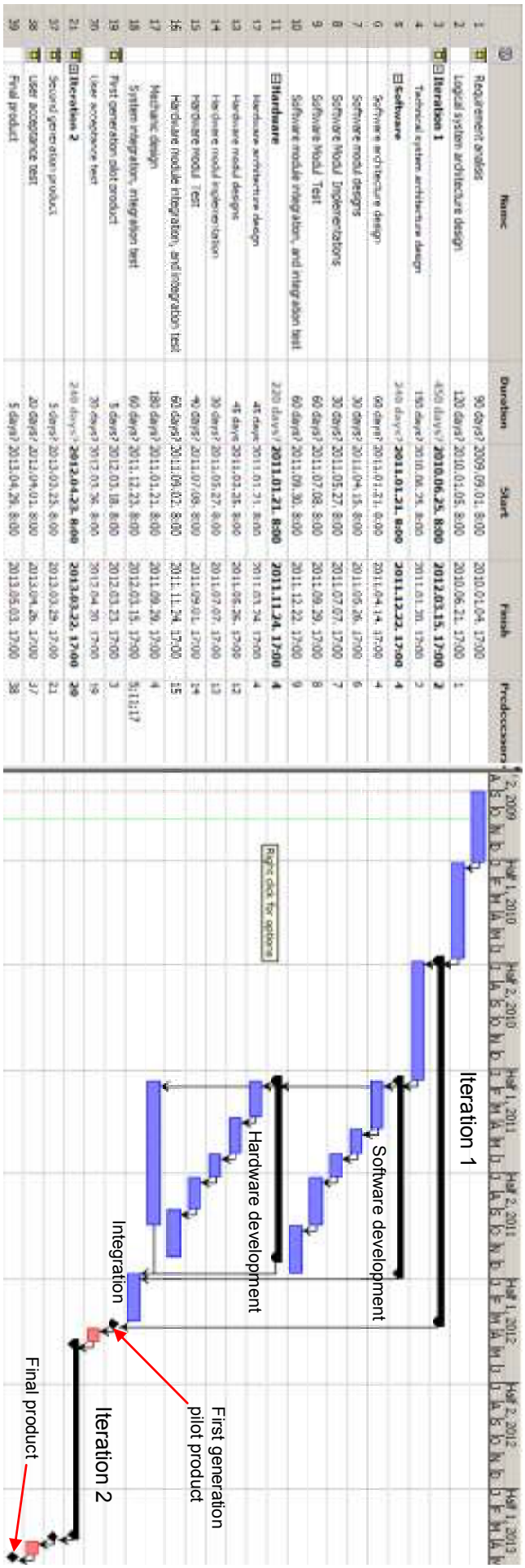
2.1.1. A Gantt diagramm

A feladatok egymásutániságának és egymásra épüléseinek megjelenítésére legtöbbet használt eszköz talán az ún. *Gantt diagramm*, amelyet a legtöbb CAD rendszer támogat. A másik igen elterjedt megjelenítés a *Pert chart*. Általában egymás mellett léteznek ezek a megjelenítési módok.

A *Gantt diagrammot* 1910-1915 között fejlesztette ki Henry L. Gantt [2.2]. 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

A 2.1 ábrán egy tipikus projekt folyamatait és annak időbeli lefolyását látjuk Gantt diagrammon ábrázolva.



***2.1 ábra**. Egy projekt folyamatai és Gantt diagrammja*

2.1.2. Constructive Cost Model

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 az 1981-ben Barry Boehm által publikált **COCOMO (Constructive Cost Model) [2.3]**. Hogy értsük ezeknek a modelleknek a működését, nézzük meg az egyszerűsített **COCOMO** modell működését:

Definíciók:

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.

Software project	a	b	c	d
Normál	2,4	1,05	2,5	0,38
Embedded	3,6	1,2	2,5	0,32

3. táblázat *COCOMO konstansok*

Mintapélda:
Példa 10,000 sor esetében (nagyjából helytálló egy ajtó vezérlőre).

Beágyazott

$$Eff = 3.6(10)^{1.2} \approx 57 \text{ [ember hónap]}$$
$$Dt = 2.5\ (57)^{0.32} \approx 9 \text{ [hónap]}$$
$$P = 57/9 = 6.3 \text{ [ember]}$$

Nem beágyazott esetre

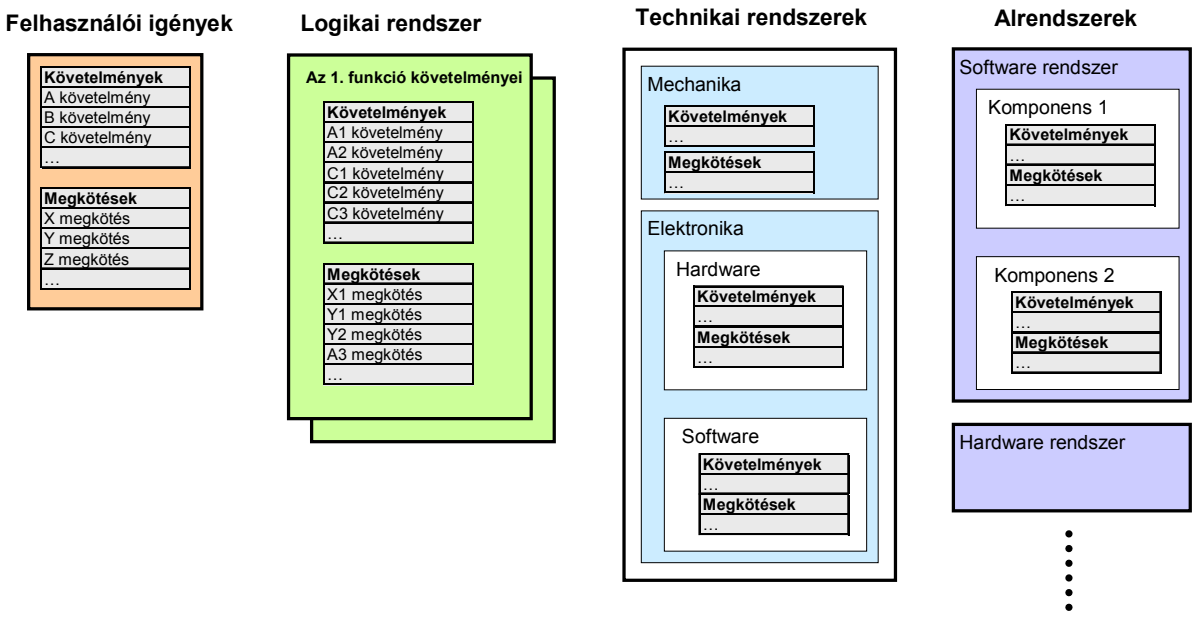
$$Eff = 2.4(10)^{1.05} \approx 27 \text{ [ember hónap]}$$
$$Dt = 2.5\ (27)^{0.38} \approx 9 \text{ [hónap]}$$
$$P = 27/9 = 3 \text{ [ember]}$$

A fenti számokból érződik, hogy a COCOMO egy nagyon elnagyolt becslést ad (beágyazott rendszerek esetében egyébként nagyságrendileg ilyen számokkal számolnak a fejlesztő cégek is). A modellnek léteznek már sokkal finomabb verziói is, például a COCOMO2, ahol rengeteg plusz információt figyelembe vesznek. Ilyen például az újra felhasznált illetve külső helyről származó kódsorok száma, e kódok integrálási nehézségei.

Gyakorlati tanácsként egy projekt időtervezéshez érdemes ismerni C. Northcote Parkinson törvényét, amely szerint: *„Egy munka mindig annyira terjed ki, hogy kitöltse az elvégzésére felhasználható időt”*. Ezt az ember magán is gyakran tapasztalja, hát még másokon (Érdemes elolvasni Parkinson más megállapításait, ha meg akarjuk ismerni a nagy szervezetek működésének emberi tényezőit).

2.2. Requirement Management (Követelmény menedzsment)

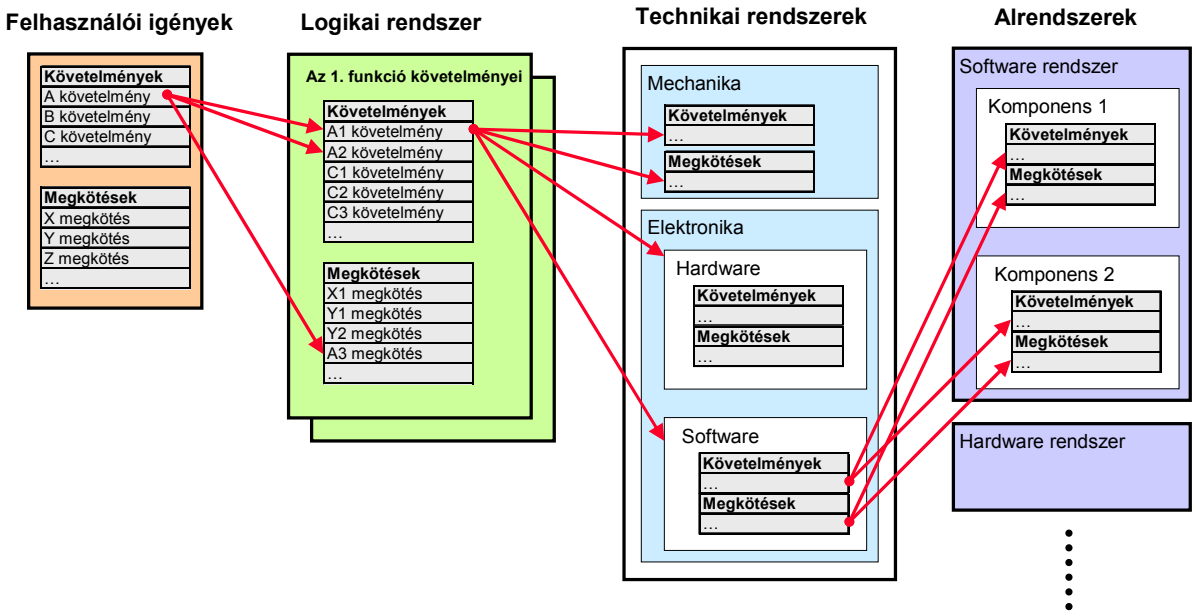
A követelmény menedzsment folyamat célja a projekt által előállított termékre vonatkozó követelmények elemzése, nyilvántartása és azok inkonzisztenciáinak kiszűrésére. A nem megvalósítható követelményeket fel kell fedni, és a megrendelővel egyeztetni kell a további sorsukról. A követelmények megadásának és gyűjtésének formalizálására sokféle kísérlet történt már. Ezek közül talán a legutolsó a SysML *requirement diagramm*-ja. A gyakorlatban azonban a legtöbb cégnél továbbra is egyszerű listaként, például Excel táblázatokban tartják számon a követelményeket. Az egyes alrendszerek, modulok követelményeinek nyilvántartása a tervezés minden fázisában fontos. (2.2 ábra).



***2.2 ábra**. Követelmények és azok beépülése a tervezésbe*

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.



2.3 ábra *Felhasználói igények és azok megvalósulásának követése*

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® [2.4].

Egy beágyazott rendszeres projectben alapvetően a következő követelményeket szokták figyelembe venni (Példaként megadjuk egy autó ajtó vezérlésére vonatkozó követelményeket):

- ***User Interface requirements (követelmények a felhasználói interfésszel szemben)***
  - Billenő kapcsoló, négyállású joy-stick stb.
- ***Funcionality requirement (követelmények a működéssel szemben)***
  - A gépkocsi minden ajtónál a saját vezérlés + a vezetőnél vezérelhető legyen az összes ajtó
- ***Open-Loop, close Loop requirements (szabályozási követelmények)***
  - Ablakemelőnél akadás érzékelése és reakció.
- ***Real time requirement (valósidejüségi követelmények)***
  - Reakció 0,2 másodpercen belül
- ***Reliability requirements (megbízhatósági követelmények)***
  - Működjön minimum 10 évig a -25 - +80 C tartományban.
- ***Safety requirements (megbízhatósági követelmények)***
  - A kocsi indulásakor a központi zár záródjon be, Ha leszedik az akkumulátor sarut, akkor újrainduláskor ne nyíljon ki a központi zár.
- ***Instalation space, weight, current draw requirements (elfoglalt terület, súly, fogyasztás követelmények)***
- ***Scalability and variant requirements (skálázhatóság, variáns)***
  - Például egy egyszerűbb verzióban ne lehessen a vezető oldaláról mozgatni a többiek ablakát, ne legyen childlock funkció.

- ***Quality Requirements (minőség követelmények)***
- ***Cost, effort, time to market requirements (költség, fejlesztési munka, piacrakerülési idő követelményei)***

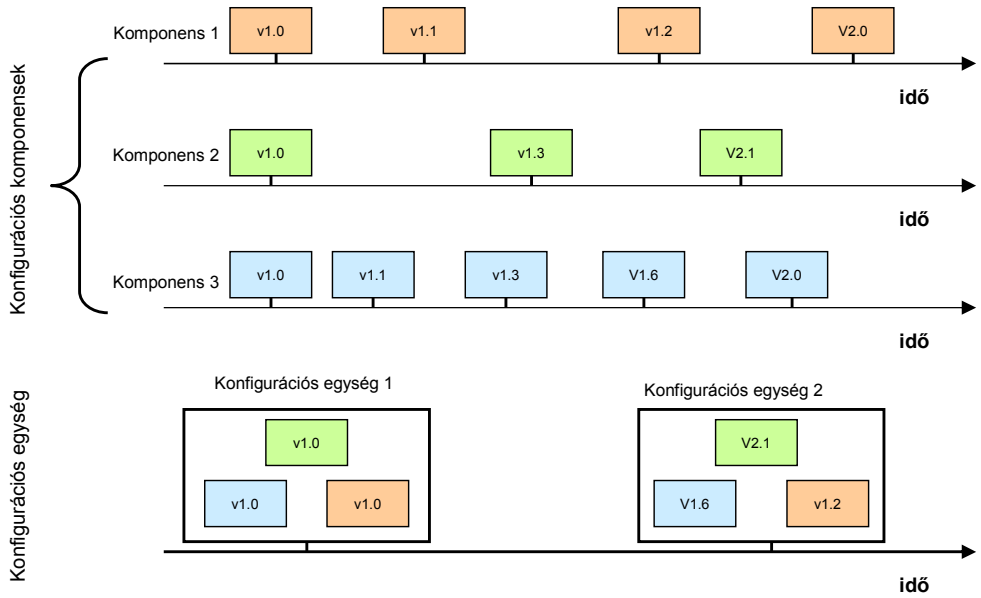
Amennyiben egy cég rendelkezik a CMMI-ban specifikált *require development* folyamattal, úgy az látja el a *requirement management* folyamatot inputtal.

2.3. Configuration Management (Konfigurációmenedzsment)

2.3.1. A konfigurációmenedzsmentben célja

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.

A konfigurációmenedzsment egy hierarchikus folyamat. Megkülönbözetünk konfigurációs *komponenseket* és konfigurációs *egységeket 2.4 ábra*.



2.4 ábra *A konfigurációs komponenseket és konfigurációs egységek viszonya*

2.3.2. A konfigurációmenedzsment folyamat lépései

A konfigurációmenedzsment általában az alábbi lépésekből áll:

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 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.

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.

- 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.

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. Tipikus ezzel kapcsolatos feladatok:

- A konfigurációs komponensek *Revision history*-ja
- A változásokhoz tartozó *log*-ok karbantartása
- A konfigurációs komponens állapotának nyilvántartása
- A *release*-ek és a fejlesztési utak közötti különbségek nyilvántartása
- Konfigurációs auditok támogatása

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.

2.3.3. A konfigurációmenedzsment és a verziókövetés viszonya

A konfiguráció menedzsmentnek egyik legfontosabb eszköze a verziókövetés, ugyanakkor azt is fontos megjegyezni, hogy a konfigurációmenedzsment jóval több egyszerű verziókövetésnél.

Miért szükséges a verziómenedzsment?

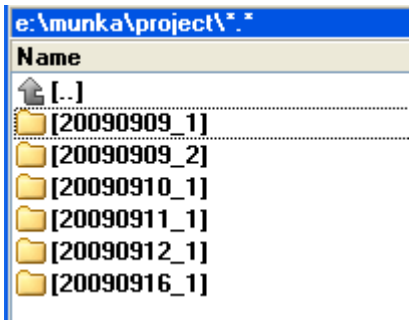
Aki valaha foglalkozott fejlesztéssel annak a következő probléma valószínűleg igen ismerős:

1. Az eddig futó alkalmazáshoz új kódrészletet rakunk.
2. Az alkalmazás lefagy.
3. Visszaállítjuk a módosításokat.
4. Az alkalmazás még mindig fagy…

Az ilyen „szívások” rengeteg időt el tudnak rabolni, tehát érdemes valamilyen megoldást találni arra, hogy adott esetben egy régebbi verzióhoz fájdalommentesen vissza tudjunk lépni a fejlesztés során. Ezt hívjuk verziókövetésnek.

A Triviális verziókövetés, avagy mire jó a Total Commander

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 (2.5 ábra).



2.5 ábra *Total Commander alapú verziókövetés*

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:

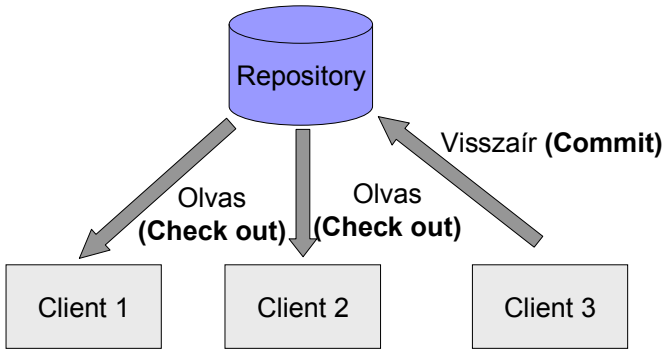
1. Változtatunk a működő alkalmazáson
2. De valaki más is beleír egy picit
3. Az alkalmazás lefagy

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

A verziókövető rendszerek alapfogalmai

A verziómenedzsment alapja nem más, mint egy adott project összes változásának nyilvántartása. Egy verziómenedzsment rendszer nyilvántart minden egyes file-on végrehajtott változtatást, valamint a könyvtárstruktúrát érintő minden változást. A felhasználónak lehetősége van megtekinteni a project vagy egy file állapotát egy adott pillanatban, megtudni, hogy ki, mit és mikor változtatott az adott projecten. A verziókövető rendszerek beépített támogatással rendelkeznek a verziók mellé adott *change log*-ok nyilvántartásához.
A verzió követés alapfogalmai (2.6 ábra):

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



2.6 ábra *A verziókövető rendszerek alapfogalmai*

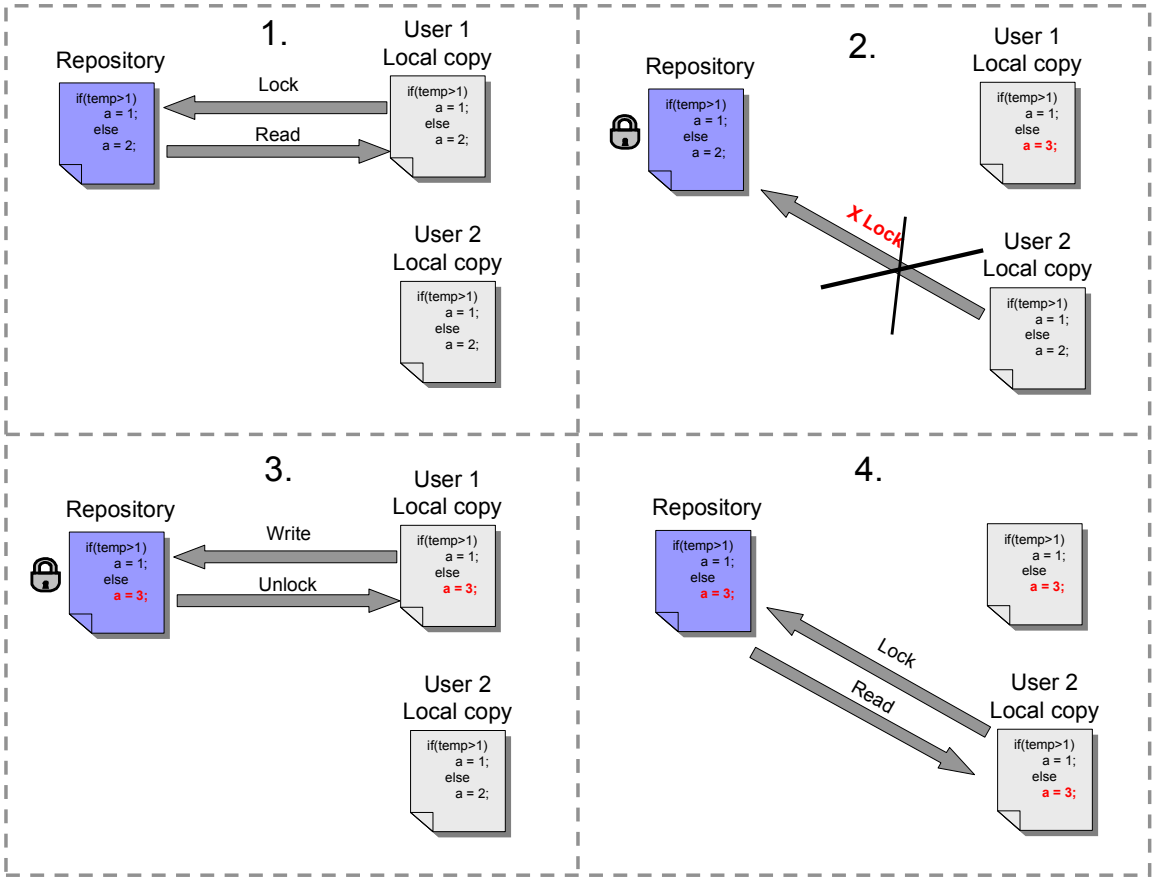
A verziómenedzsment rendszerekkel kapcsolatban felmerülő első kérdés általában, hogy hogyan támogatja a felhasználók együttműködését, úgy, hogy azok ne lépjenek egymás lábára? Ilyen stratégia nélkül könnyen előfordulhat, hogy egy file-t vagy projectet egyszerre többen módosítanak, majd felülírják egymás módosításait (a módosítások nem tűnnek el, de nem is kerülnek bele az új verzióba).

Lock-Modify-Unlock megközelítés

Az első ilyen együttműködési mechanizmus az ún. Lock-Modify-Unlock megközelítés volt.

Ennek működése a következő (2.7 ábra):

- Módosítás előtt le kell lock-olni egy file-t.
- Tehát egyszerre csak egy ember tudja módosítani a file-t. Olvasni tudja más is.



2.7 ábra *A Lock-Modify-Unlock megközelítés*

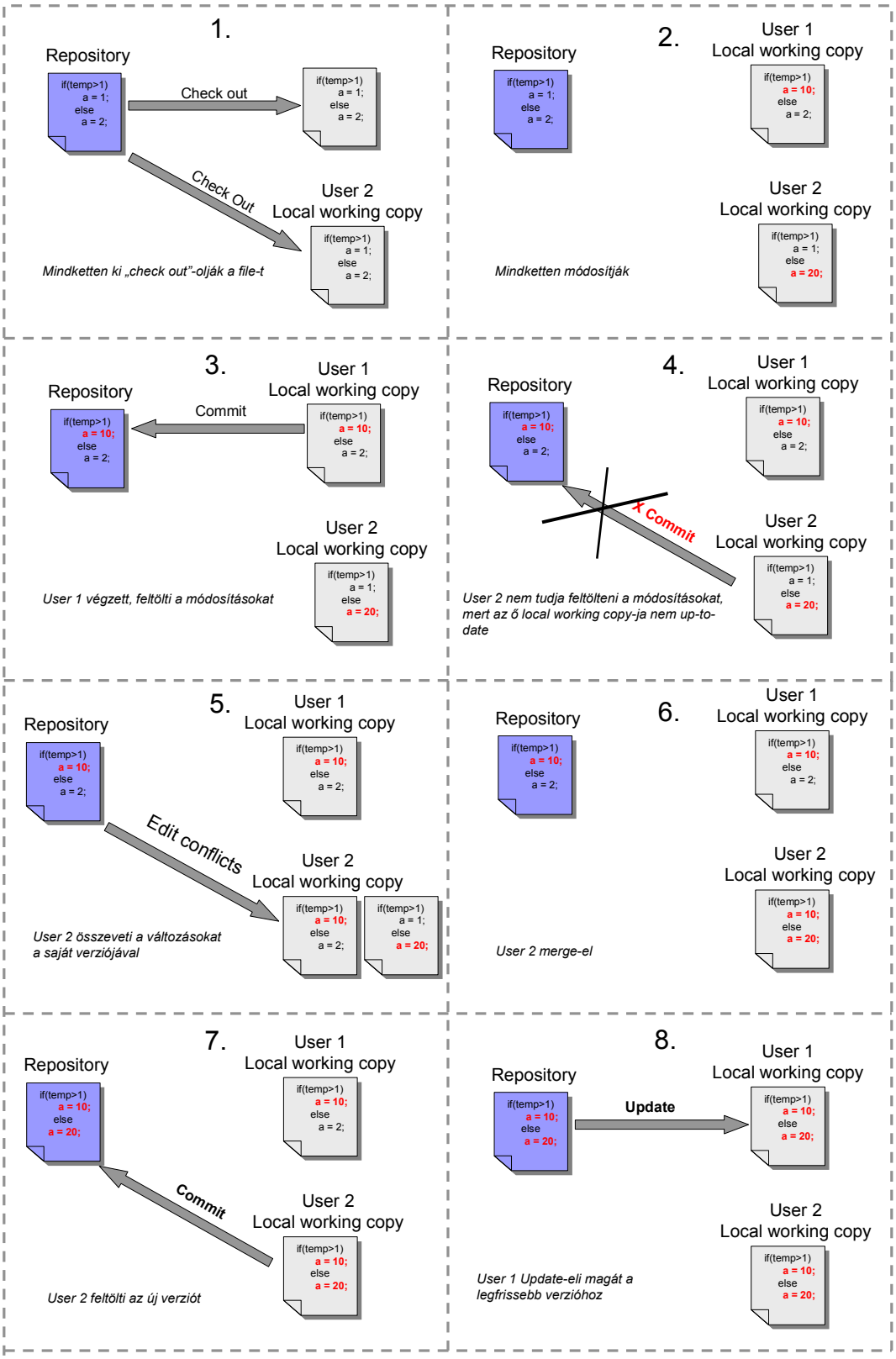
A Lock-Modify-Unlock megközelítésnek a következő problémái vannak:

- 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 Copy–Modify–Merge megközelítés

A Copy–Modify–Merge megközelítést (2.8 ábra) használja a legtöbb manapság is forgalomban lévő verziókövető rendszer (CVS, Subversion stb.) A megközelítés jellemzői a következőek:

- 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, tehát fuzionálással 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.



2.8 ábra *A Lock-Modify-Unlock megközelítés*

A 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.

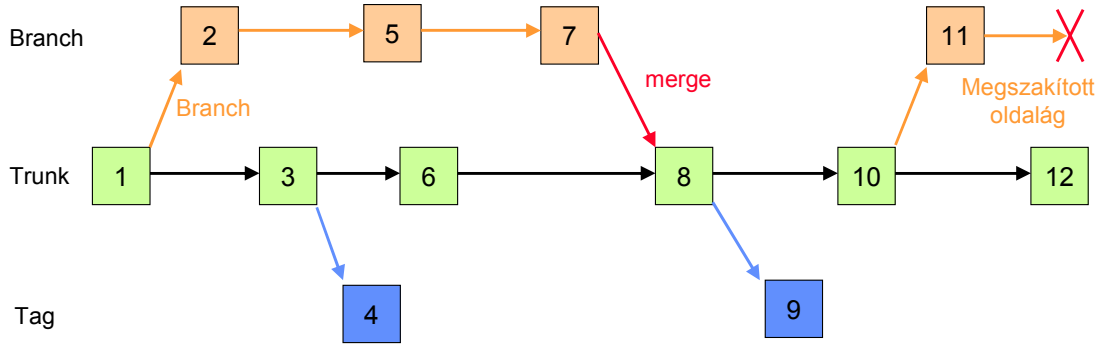
Nagyon fontos azt hangsúlyozni, hogy a verziókövető rendszer nem helyettesíti az emberek közötti kommunikációt. Csak segít a tisztánlátásban és a munkafolyamatok követésében. A jó kollegiális viszonyra mindenképpen szükség van, anélkül semmi sem működik.

Ez a Copy–Modify–Merge megközelítés ugyanakkor nem használható minden fejlesztésben résztvevő termékre, dokumentációra. 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.)

A Törzs ág, Cimkézett ágak, és Elágazások vagy Trunks, Tags és Branches

Egy komplex project verziókövetése során nem csak az egyes file-ok verzióit, hanem az egész project konfigurációját nyomon kell követni (2.9 ábra). 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.



2.9 ábra *Egy project verziókövetési életciklusa*

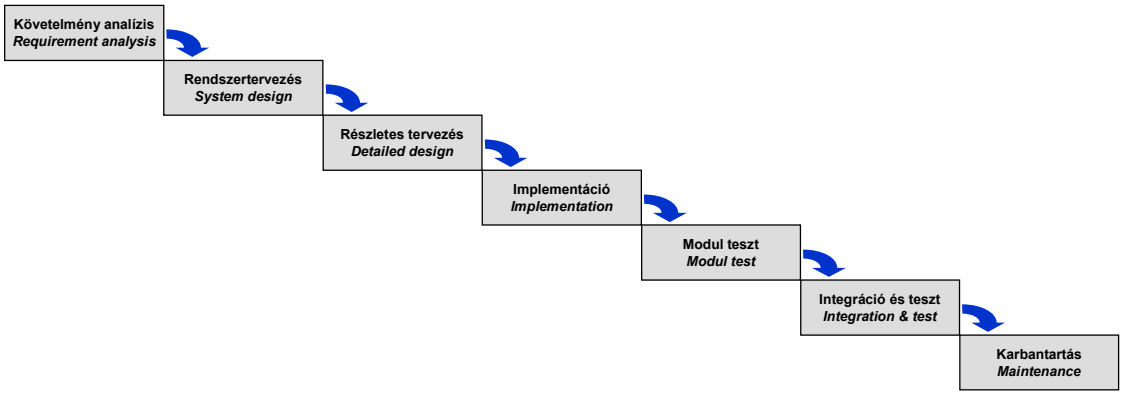
3. A rendszertervezés fejlesztési folyamatai

3.1. A fejlesztési folyamatok életciklus modelljei

A műszaki világban számos fejlesztési modell terjedt el az évek során. Ez a fejezet röviden átveszi a mérnöki folyamatok végrehajtására használt legelterjedtebb életciklus modelleket. Mielőtt azonban ismertetnénk ezeket a modelleket fontos megjegyezni, hogy a bemutatott modellek mindegyike csak ***útmutató információkat*** tartalmaz. Az, hogy hogyan hajtanak végre egy ilyen fejlesztési modellt mindig az ***adott cégtől függ.*** Nem véletlenül tárgyalja szinte mindegyik ilyen jellegű módszertan az adott modell ***testreszabását***. Tehát, ne lepődjünk meg, hogyha máshol egy picit eltérően nevezik a lépéseket, vagy nem teljesen az általunk bemutatott lépéseket hajtják végre.

3.1.1. A Vízesés modell

A rendszertervezés egyik első fejlesztési modellje az ún. *vízesés modell* [3.1] volt, amelyet 1970-ben publikáltak (3.1 ábra).

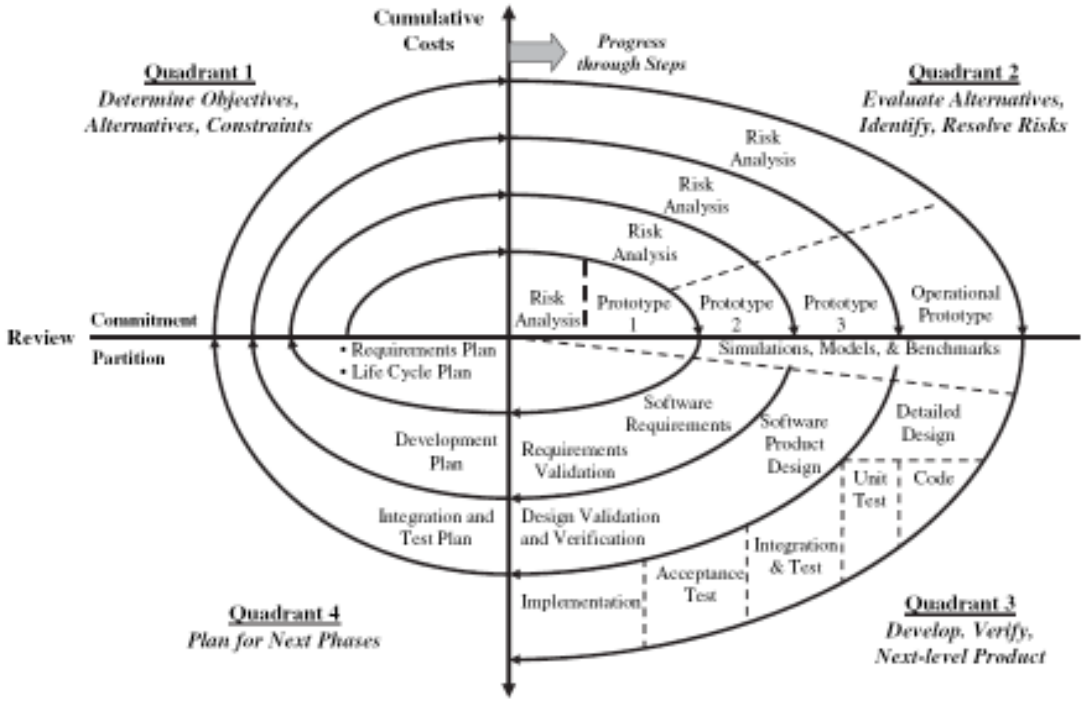


3.1 ábra *A vízesés modell egyik lehetséges formája*

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.

3.1.2. A Spirál modell

A *spiral modelt* [3.2] Boehm publikálta 1986-ban. 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.



3.2 ábra *A Spirál modell*

A Spirál modell fázisai (negyedei) (3.2 ábra):

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)

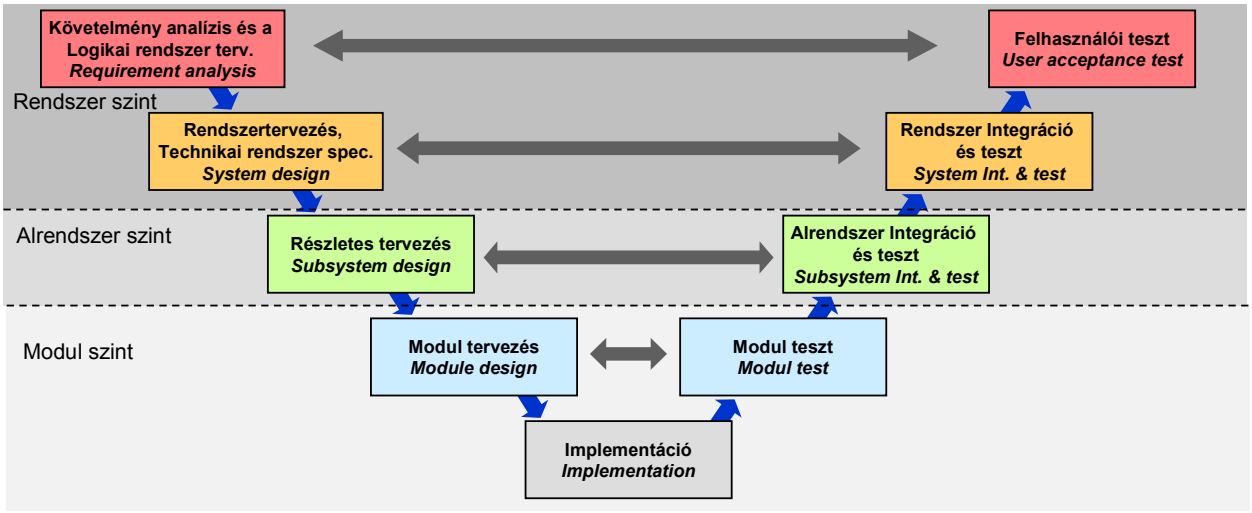
A következő fázisnak megfelelő termék kiválasztása, és a következő fázis előkészítése. Ha az előző prototípus már teljesen működő verzió volt, akkor egy vízesés modellszerű fejlesztés befejezése.

Next Phases (A következő fázis)

A következő lépés és annak ütemének megtervezése.

3.1.3. A V-modell

Az 1997-ben megjelent *V-modell* [3.3] (3.3 ábra) 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 legtöbb beágyazott rendszereket fejlesztő cég ezt a modellt alkalmazza.



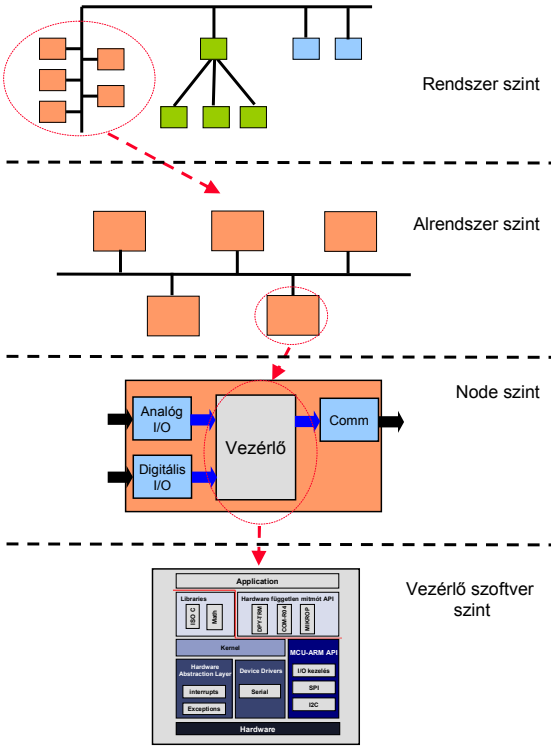
***3.3 ábra** A V-modell*

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).

Mielőtt elkezdenénk a V-modell lépéseinek részleteivel foglalkozni, ismerkedjünk meg annak valóságban való használati módjával.

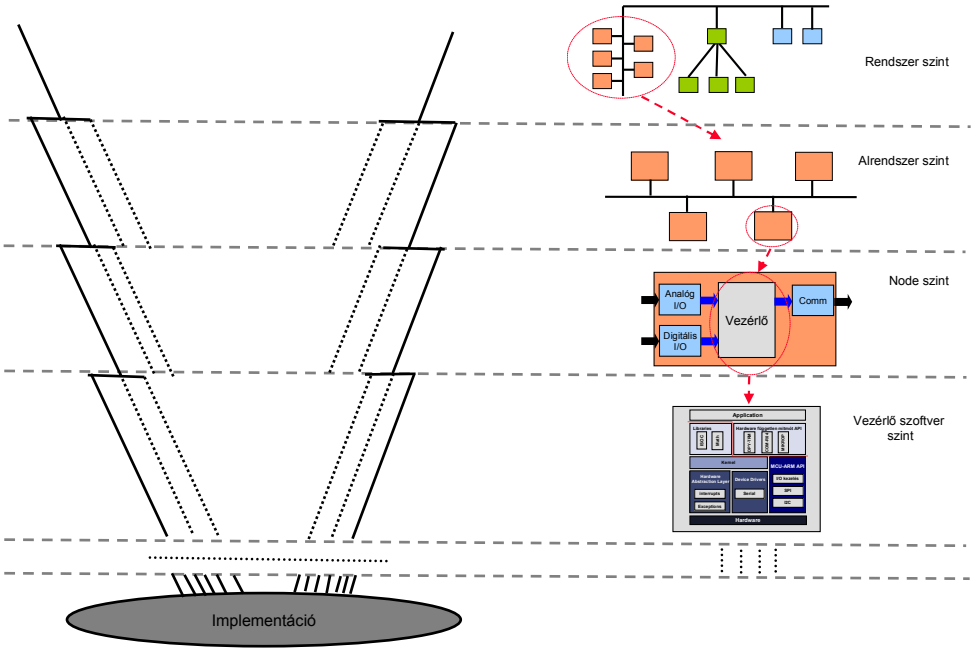
Hierarchia szintek

Minden rendszer tervezésénél az első feladat a rendszer hierarchikus absztrakciós szintekre való bontása (3.4 ábra). Ezek végigkísérik a tervezés folyamatát, és az egyes lépések az absztrakciós szintekhez tartozó tervezési folyamatokat fogják megadni.



3.4 ábra *Rendszertervezési absztrakciós szintek (nem teljes)*

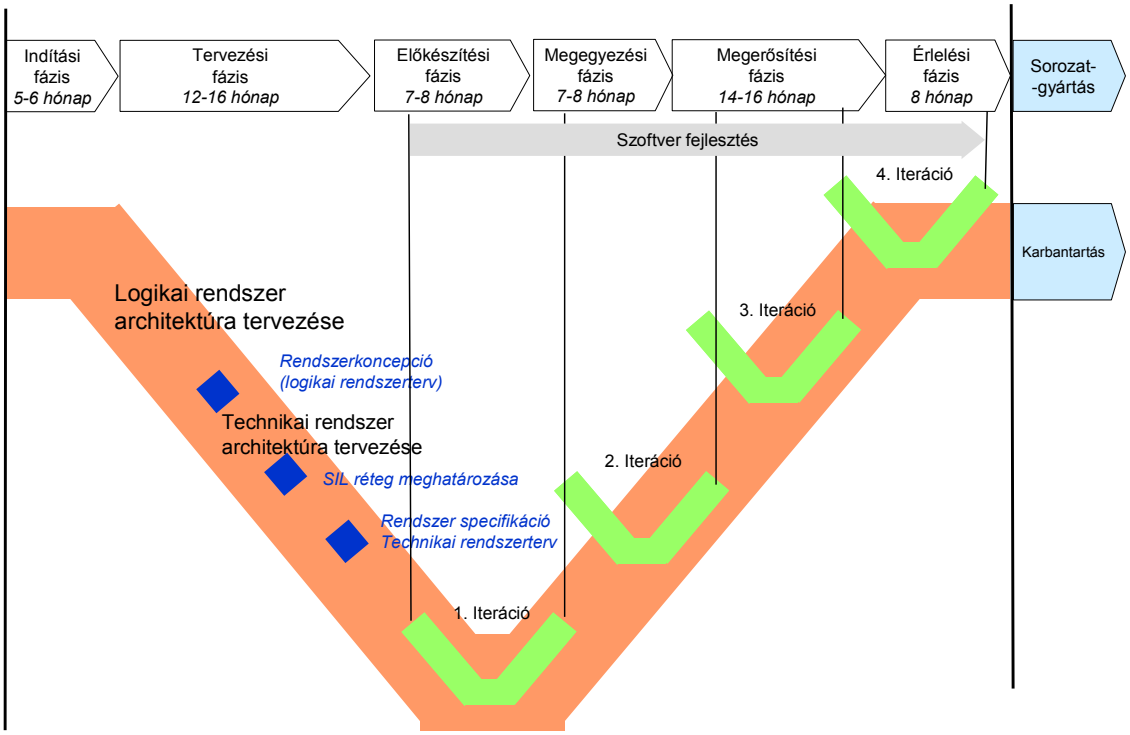
Az absztrakciós szintekre való bontás után érezhető, hogy nincs olyan fejlesztési modell, ami elágazás nélkül végig tudná követni ezt a sok szintet. Azért, hogy a gyakorlatban ezeket, a modelleket használni lehessen, a legtöbb esetben a fejlesztési modell az egyes szinteken külön ágakra, külön feladatokra osztódik szét. Ezeknél a fa jellegű elágazásoknál történik meg tipikusan a feladatok alvállalkozóknak, fejlesztési csoportoknak való kiadása is (3.5 ábra).



3.5 ábra *A rendszertervezési folyamat feladatokra bontása a tervezési szakaszban és összeintegrálódása a tesztelési/integrálási szakaszban*

***Például**: Egy autógyár rendszerszinten megtervezi egy gépkocsi elektronikáját. Az egyes belső csoportjainak kiadja az alrendszerek, mint erőátvitel, komfortelektronika, stb. részletes tervezését. Az egyes csoportok specifikálják node-ok megvalósítását, majd kiadják azokat alvállalkozóknak. Miután az alvállalkozók megcsinálták és tesztelték a node-okat, a csoportok integrálják azokat alrendszerekké és tesztelik azokat. Végső lépésként pedig a rendszertervező csoport összeállítja az alrendszerekből a teljes rendszert és teszteli azt.*

A (3.4 ábra), (3.5 ábra) segítségével bemutattuk, hogy a fejlesztési folyamat milyen hierarchia szintekre osztható és hogyan bomlik részfolyamatokra. 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. Ezt a modern fejlesztési modellek már figyelembe veszik. Például a V-modell XT már tartalmaz *Iterációk tervezése* lépést (ez nem integrális része az alap V-modellnek). A (3.6 ábra) egy tipikus V-modell alkalmazási gyakorlatot mutat be.



3.6 ábra Egy tipikus projectfejlesztés a V-modell alapján

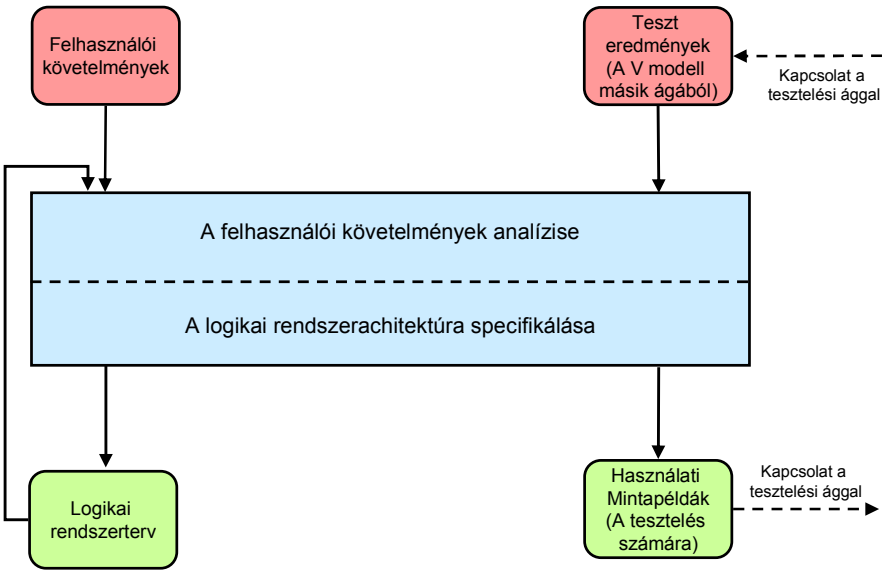
3.2. A V-modell tervezési lépései

Ez a fejezet a V-modell tipikus fejlesztési lépéseit tekinti át. Az egyes lépéseket úgy határozzák meg, hogy mindig valamilyen jól megfogalmazható végtermék álljon elő a lépés végén. Ezeket a végtermékeket a szakzsargonban ***artifact***-nek szokták nevezni.

3.2.1. Követelményanalízis, és a Logikai rendszerterv elkészítése

A Követelményanalízis, és a Logikai rendszerterv lépés (3.7 ábra) angol neve: Analysis of system reqirements and specification of logical system architecture.

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. Sokszor előfordul ugyanis, hogy amit a felhasználó követelménynek gondol, az a valóságban nem az, és neki egészen másra van szüksége. Például egy tipikus rossz felhasználói követelmény, hogy 24bites AD-t használjunk. Ez a valóságban nem feltétlenül jelenti azt, hogy a felhasználónak valóban 24-bites AD-ra van szüksége (általában nincs is tisztában azzal, hogy ez mit jelent pontosan), hanem azt jelenti, hogy valamilyen értéket ő egy adott pontossággal akar mérni. Az már egy műszaki kérdés, hogy ez hogy megvalósítható (jelkondicionálás + 16 bites AD stb.), de nem követelmény.



3.7 ábra A felhasználói követelmények analizálása és a logikai rendszer architektúra megtervezése

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. A legtöbb esetben az UML *Use Case, Class, Interaction* diagrammjait használják erre a célra (Az UML diagramokat az előző féléves *Beágyazott rendszerek szoftvertechnológiája* ***vimim150*** tárgyban oktatták).

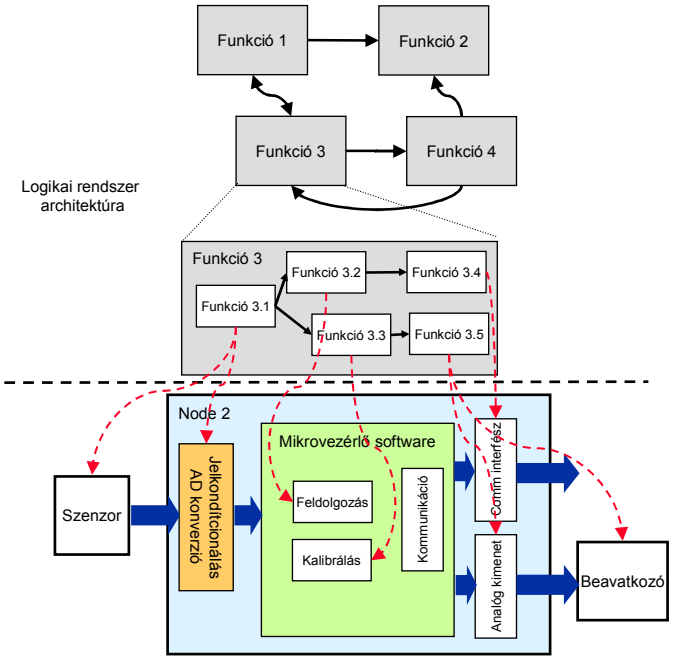
Fontos megjegyezni a lépéssel kapcsolatban, hogy a ***logikai rendszer architektúra*** mellett egy ***használati példák*** (*use cases)* artifact-et is létrehoz. Ez azért kiemelten fontos, mert ezek a felhasználási példák teremtenek kapcsolatot később a tesztelési ággal. Ez a dokumentáció fogja a *rendszer* és *végfelhasználói* tesztek alapjait jelenteni.

A 3.7 ábrán megfigyelhető az iterációra való felkészülés is. 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.

3.2.2. A logikai rendszer architektúra elemzése, a technikai rendszer architektúra specifikációja

A logikai rendszer architektúra elemzése, a technikai rendszer architektúra specifikációja lépés angol neve: ***Analysis of logical system architecture and specification of technical system architecture.***

A logikai rendszerarchitektúrából kiindulva a technikai rendszer architektúra hozza létre a rendszer technikai vázát. A lépés feladata, hogy az egyes logikai funkciókat elossza a tényleges fizikai modulok között (3.8 ábra).

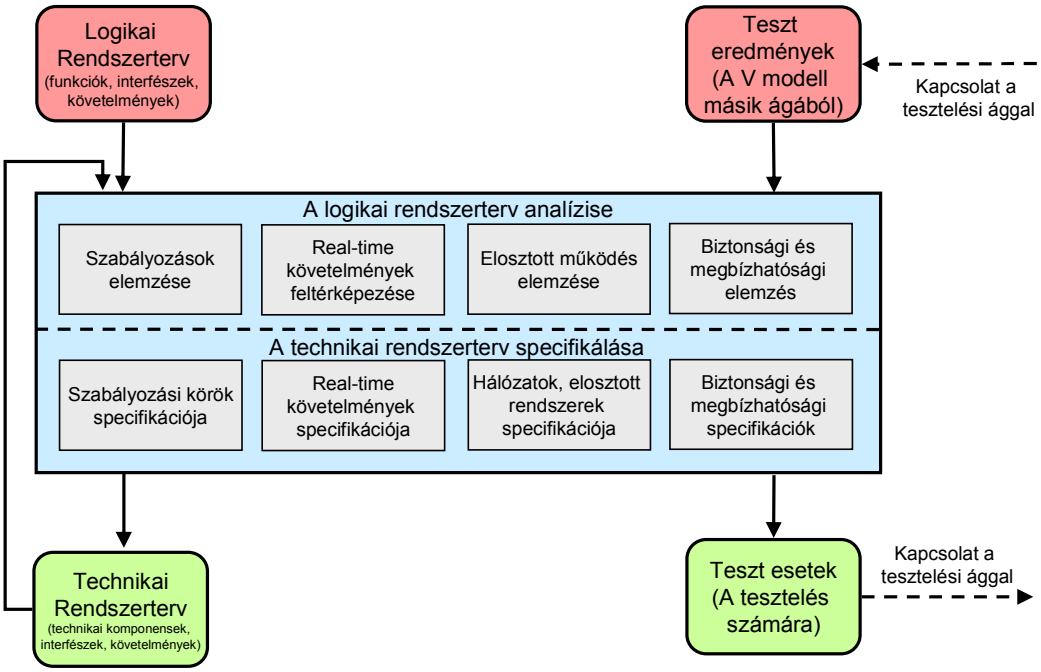


3.8 ábra A logikai rendszer architektúra leképeződése a technikai rendszer architektúrává

Ebben a lépésben történik meg a technikai alrendszerek további finomítása és alcsoportokra bontása, valamint a szoftver és hardver folyamatok különválasztása és specifikálása is.

Fontos megjegyezni, hogy amikor a logikai rendszerarchitektúrát leképezzük a technikai rendszerarchitektúrává, akkor egy logikai funkció többnyire nem egy fizikai egységgé fog leképződni. Általában egy fizikai egység több logikai funkciót valósít meg, vagy fordítva. Tehát a leképezés tipikusan nem 1:1, éppen ezért is különböztetik meg a két lépést.

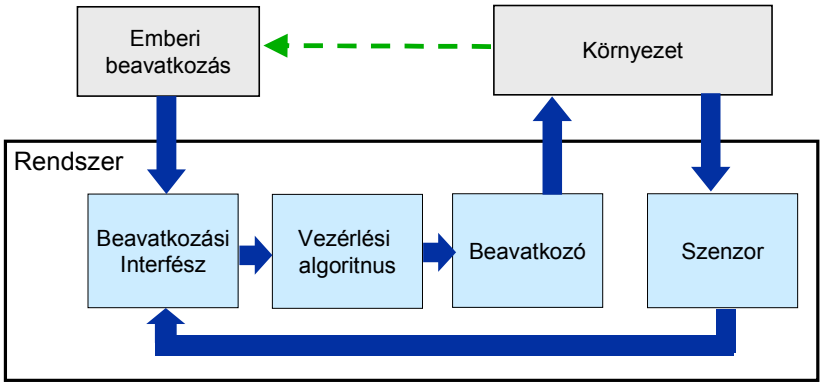
Arra, hogy a logikai rendszerarchitektúrából hogyan kell technikai rendszerarchitektúrát létrehozni, nem lehet minden körülmények között érvényes útmutatót adni, hiszen minden rendszer más, ezért a leképezés módszerei különbözőek lehetnek. Az alábbiakban egy igen széleskörűen, elsősorban komplett rendszereknél használható leképezési lépéssorozatot mutatunk be (3.9 ábra).



3.9 ábra A logikai rendszerarchitektúra elemzése és a technikai rendszerarchitektúra specifikálása

A szabályozási körök elemzése és specifikációja

Amikor egy technikai rendszerarchitektúrát specifikálunk, akkor a legtöbb esetben először is felmérjük a rendszerben található szabályozási köröket. Legyenek azok nyílt, vagy zárt hurkúak (3.10 ábra). 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.



3.10 ábra Egy általános szabályozási/vezérlési kör blokkvázlata --algoritmus

Sok esetben, ahol a szabályozási kör komplex, mint például egy motorvezérlő elektronikánál a viselkedést már ebben a stádiumban analizálják valamilyen modellező eszköz, mint például Matlab Simulink segítségével (adott esetben – ha ezt a megbízhatósági elemzés is indokolja – prototípus készítést is végeznek).

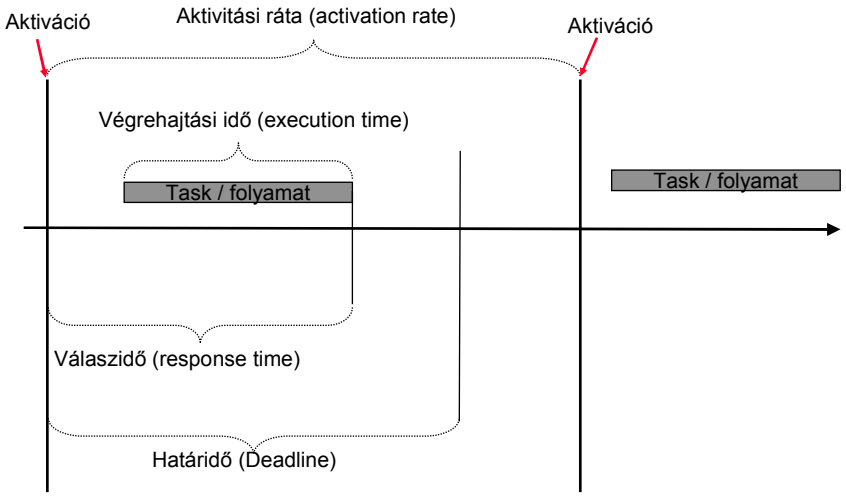
Már ezen a ponton elkezdődik a tényleges hardver figyelembe vétele, hiszen a digitális rendszerek tulajdonságukból adódóan mindenképpen diszkrét idősorokon alapuló szabályozást valósítanak meg. Figyelembe kell venni a mikrovezérlők véges feldolgozási sebességét, beleértve a kisebb számábrázolási felbontásból eredő kerekítési hibákat, az AD átalakítók késleltetését, nemlinearitását stb.

Ez a lépés egyben előkészíti a következő, real-time követelményeket elemző lépést.

Real-time követelmények feltérképezése és specifikációja

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)* (3.11 ábra).



3.11 ábra Realtime követelmények paraméterei

Azt, hogy az egyes folyamatokra hogyan kell real-time karakterisztikát megállapítani az előző féléves: *Valós idejű és biztonságkritikus rendszerek* ***(VIMIM151)*** tárgyból elsajátíthattuk.

Elosztott működés elemzése, az elosztott rendszerek és hálózatok specifikálása

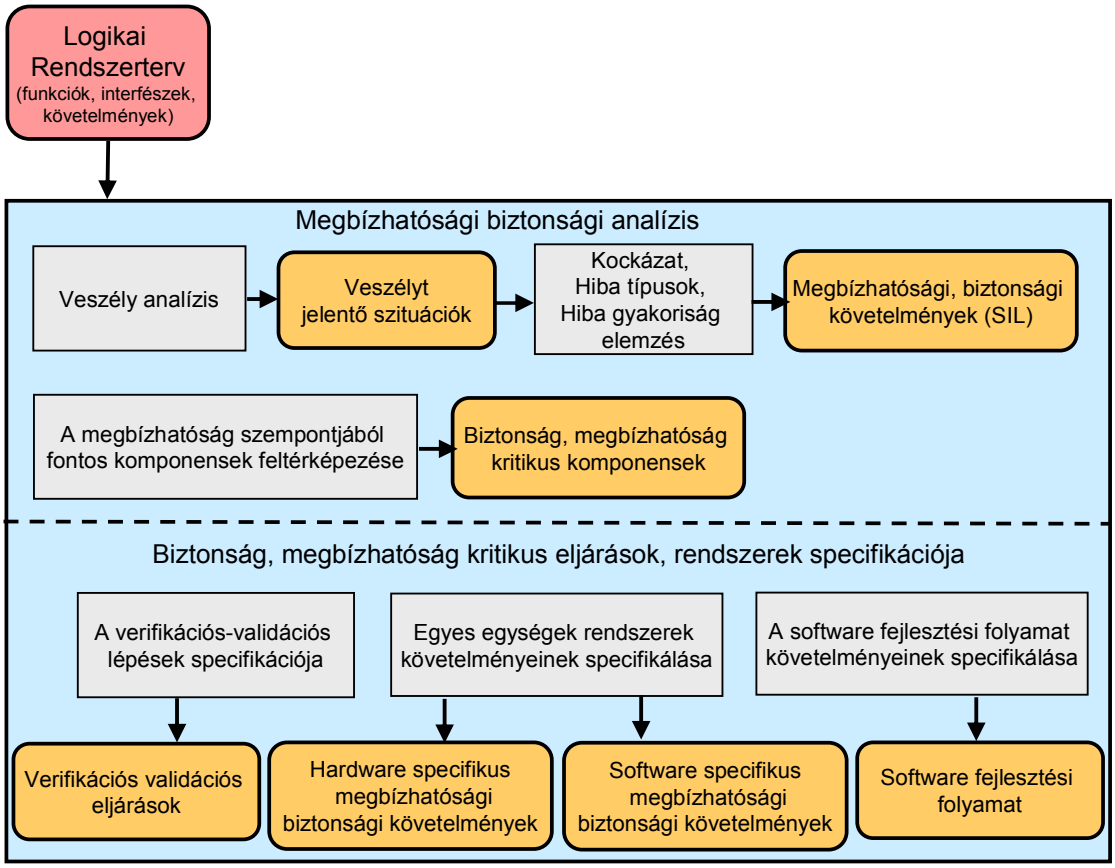
A real-time időzítések és a szabályozási követelmények ismeretében a rendszer funkciói szétosztásra kerülnek az egyes node-ok, egységek között. Tehát ténylegesen specifikáljuk, hogy melyik funkciót melyik hardver egység és hol hajtja végre. Ahhoz, hogy ezt megtegyük először elemezni kell, hogy mely műveletek 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.

A megbízhatóság és biztonság szempontjából fontos részek azonosítása

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.

Ez a biztonsági és megbízhatósági analízis a 3.12 ábrán bemutatott lépésekből áll.



3.12 ábra A biztonságossági és megbízhatósági analízis lépései

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.

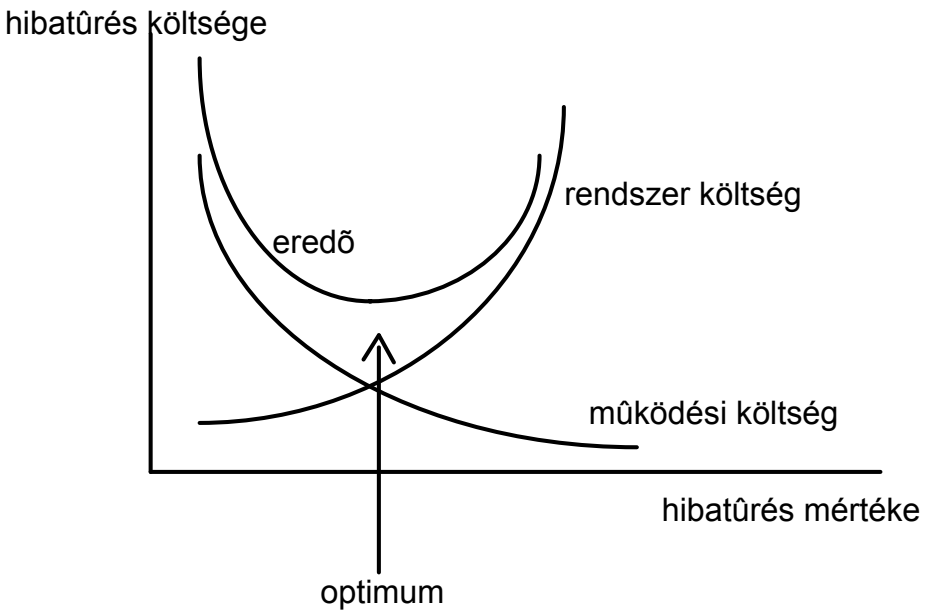
Risk (kockázat): Valaminek a kockázatát nem lehet egzakt módon megadni, a kockázat megadásra a legelfogadottabb mód, hogy a baleset valószínűségét és az ebben az esetben bekövetkező kár mértékének a szorzatát vesszük.

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.

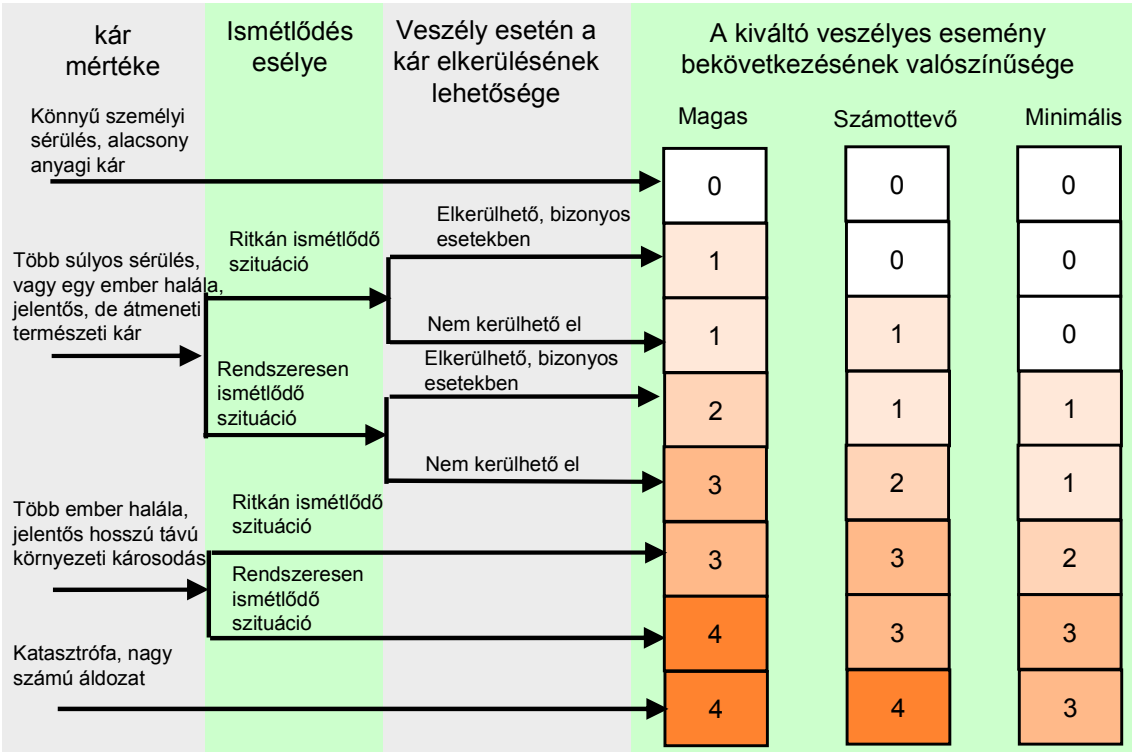
Limit Risk (kockázati határ): A kockázati határt szintén nem lehet egzakt módon meghatározni (függ a hatályos törvényektől, piaci pozíciótól stb.), a legjobb definíció rá, hogy egy olyan optimum, ahol a rendszer költsége (redundáns vezérlő stb.) és a rendszer működésének költsége (kártérítési perek, visszahívások) együttesen a minimumot adják 3.13 ábra.



3.13 ábra A limit-risk megtalálása

Természetesen ezt az optimumkeresési folyamatot különböző szabványok támogatják. Például az IEC61508-as szabvány által specifikált ***SIL (Safety Integrity Level)*** rétegek egy ilyen besorolást nyújtanak (érdemes megjegyezni, hogy ezt a nemzetközi IEC 61508-as szabványt csak 1998-ban dolgozták ki).

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 módját a 3.14-es ábra mutatja be.



3.14 ábra A SIL réteg meghatározásának egy módja

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. (Ezek itt nem kerülnek ismertetésre, mert a múlt féléves tárgyaknál számtalan ilyen példával találkozhattunk. Az egyes komponensek, mint processzorok, diódák meghibásodási tényezőit a gyártóktól lehet megszerezni). 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. A 3.15-ös ábrán láthatjuk, hogy 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.

Példaként 3.15 ábra bemutatja egy cég fejlesztési szabályait a SIL réteg besorolás függvényében. A cég csak SIL3-ig terjedő rendszerekkel foglalkozik (SIL4 már csak a repülőkben és atomerőművekben használt rendszerekre jellemző).

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	+	++	++	++

Jelmagyarázat:
"-": tilos
"0": nem szükséges
"+": ajánlott
"++": kötelező

3.15 ábra. A tervezés egyes munkafolyamatai a SIL réteg függvényében (példa)

A technikai rendszer architektúra tervezésénél figyelembe kell venni azokat a megkötéseket is, amelyeket vagy a törvény ír elő, vagy az alkalmazott gyártási technológia szab meg.

3.2.3. Szoftver követelmények elemzése és a szoftver architektúra megtervezése

A Szoftver követelmények elemzése és a szoftver architektúra megtervezése lépés angol neve: ***Analysis of software requirements and specification of software architecture.***

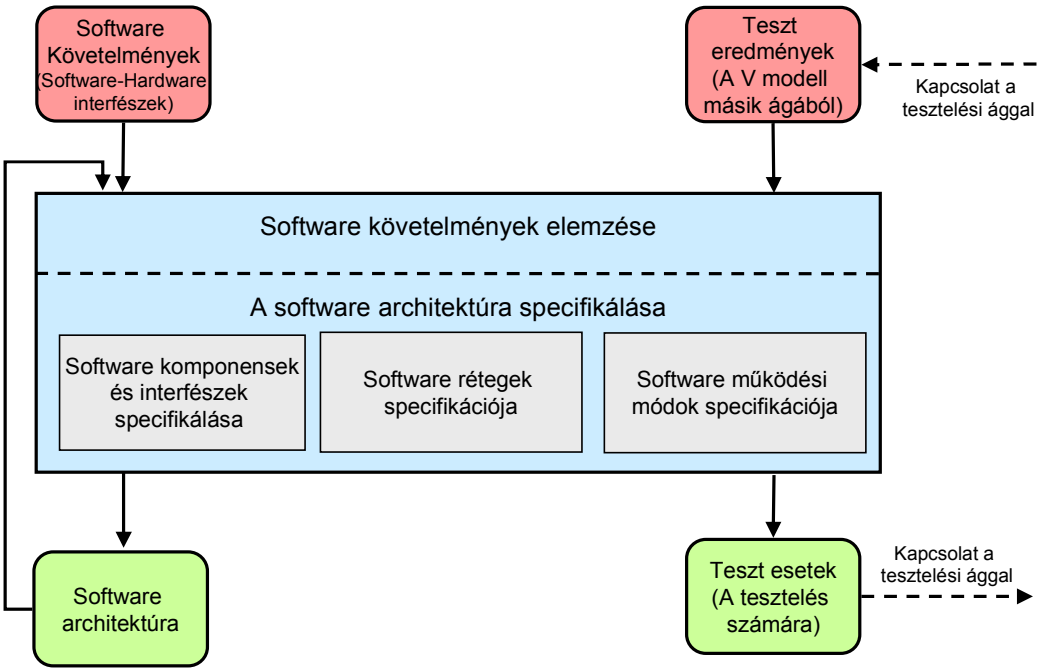
A *technikai rendszer architektúra* megtervezése után a fejlesztési folyamat szétválik hardware, software, mechanika irányba (3.5 ábra). Mi itt részletesebben a szoftver fejlesztési ággal foglalkozunk, a tárgy későbbi része tér ki a hardver ágra.

A *technikai rendszer architektúra* megtervezésénél előálltak a szoftverre vonatkozó követelmények. Ez a szoftver szempontjából annyit, tesz, hogy az egyes feladatok hozzá lettek rendelve a mikrovezérlőkhöz és meghatároztuk az egyes feladatok kommunikációs interfészét is. A szoftver architektúra tervezése lépés ezek után specifikálja azokat a szoftver funkciókat, amelyek a vezérlőkön belül a bementi adatokból előállítják a kimeneti adatokat.

A szoftver architektúra tervezése az alábbi részfeladatokból áll:

- 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.

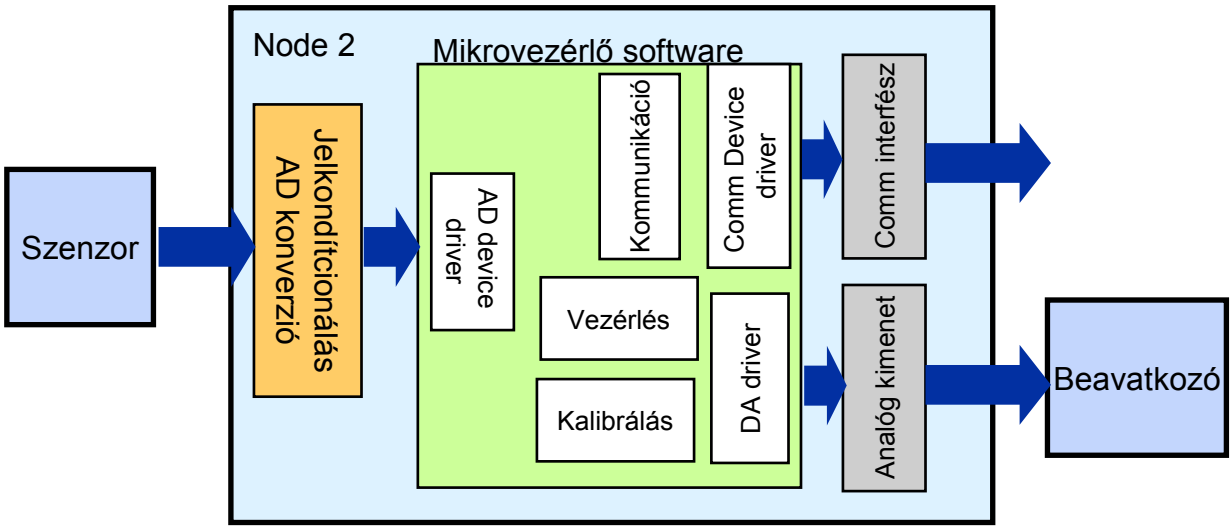
Magát a tervezési folyamatot a 3.16. ábra szemlélteti.



3.16 ábra A software követelmények analízise, a software architektúra megtervezése

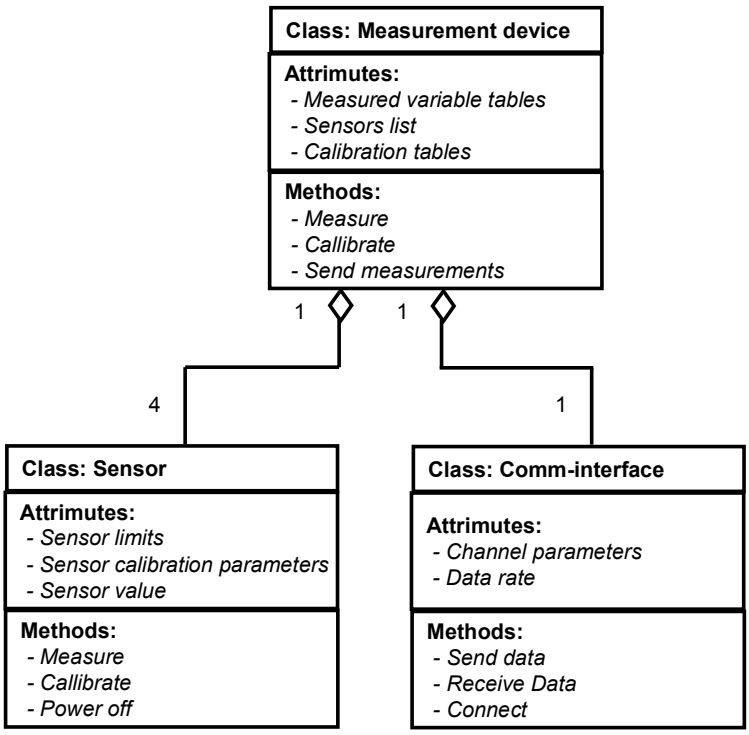
Szoftver komponensek és azok interfészeinek definíciója

A szoftver architektúra tervezésének első lépése, hogy azonosítjuk a rendszer szoftver komponenseit és azok interfészeit. Ennek a lépésnek az első munkaszakasza a szoftver rendszer határainak pontos definiálása (3.17 ábra). Mi az, amit a szoftver rendszer témakörébe tartozik, pontosan mit fog megvalósítani a szoftver? Ez gyakorlatilag nem más, mint a szoftver követelmények dokumentációban található specifikációjának elemzése.



3.17 ábra Egy vezérlő szoftver határainak megállapítása és a belső szoftver funkciók azonosítása (nagy része direktben adódik az előző lépésekből)

A szoftver komponensek meghatározásánál, és az interfészek specifikálásánál legtöbbször UML alapú formalizált leírást alkalmaznak. Ennél a lépésnél elsősorban a *Class diagramm* használható. A 3.18 ábra egy egyszerű példát mutat erre. Az UML formalizmusokat és UML alapú tervezést itt bővebben nem tárgyaljuk, hiszen a múlt féléves *„Beágyazott rendszerek szoftvertechnológiája" (VIMIM150)* tárgyból ezzel részletesen foglalkoztak.

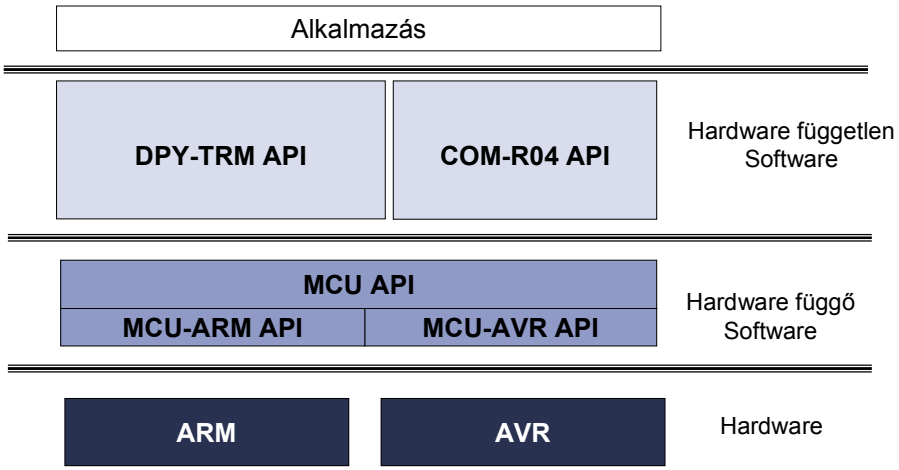


3.18 ábra Egy mérőeszköz Class diagram tervének részlete

Szoftver rétegek meghatározása

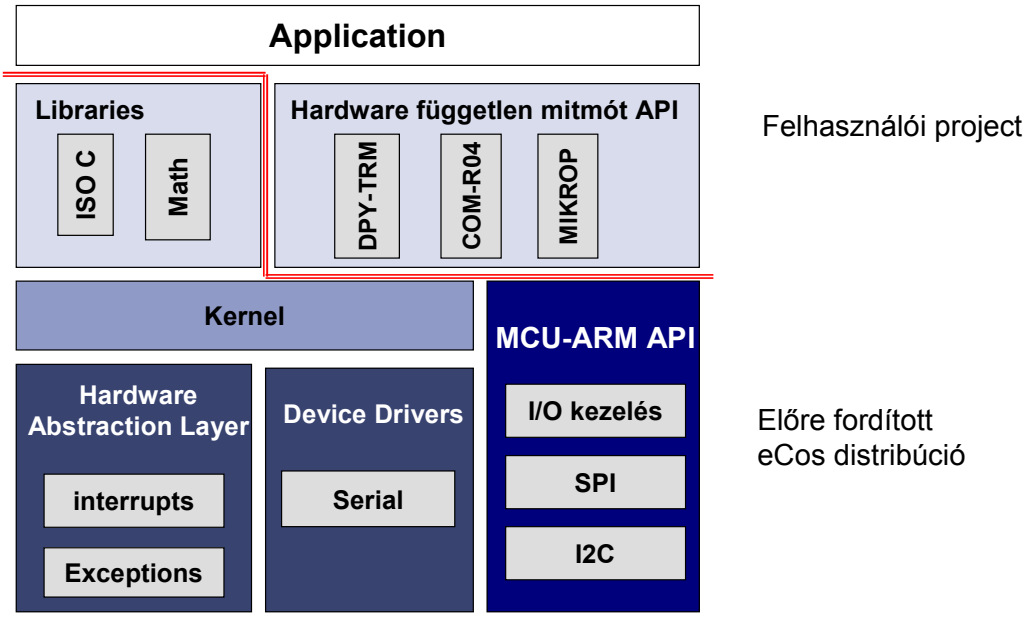
A szoftver funkciókat a legtöbb esetben rétegekre bontják. Ezek a rétegek tipikusan a *hardware abstraction layer*, *kernel / device drivers*, *application support layer* és az *application layer*. A kommunikációs protokolloktól eltérően a szoftver rétegek általában nem az ún. *linear-order layer model* 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.

Egy ilyen nagyon egyszerű szoftver réteg hierarchia a *mitmót* API is (*3.19 ábra*). Ennél jól látható, hogy az *MCU-API*-k által nyújtott közös felülettel oldható meg, hogy mind a 32 bites ARM, mind a 8 bites AVR platformon működjenek a kijelző és a rádió kezelői függvényei.



3.19 ábra A mitmót API-k hierarchikus szerkezete

Egy általános beágyazott rendszer szoftver rétegeinek a bemutatásához jó példa az előzőekben már megismert *eCos* operációsrendszerrel összeintegrált *mitmót api* (*3.20 ábra)*. Az ábrán látható a legtöbb beágyazott rendszerben alkalmazott réteg megnevezési konvenció is.



3.20 ábra Egy általános beágyazott rendszer szoftverrétegei (Az ARM-es mitmót kártya operációs rendszerrel)

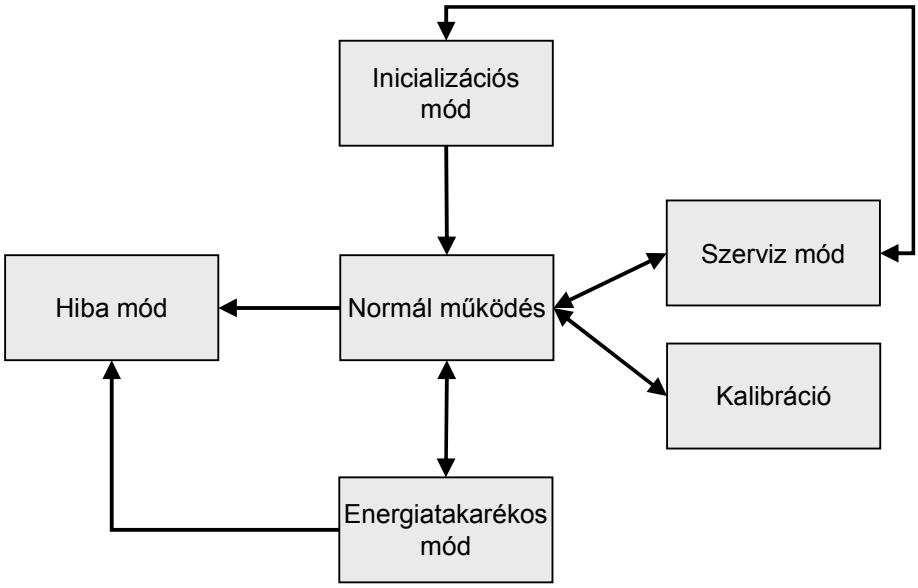
A szoftver működési módok meghatározása

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.

Például egy rádiós hálózati eszköznek általában nincs megbízhatóságkritikus csökkentett módja. Egy autóipari vezérlőnek nem feltétlenül van energiatakarékos módja (egyre többször van), de mindig van csökkentett (*limp home*) módja.

Érdemes külön is kiemelni a normál működés közben nem látható, ezért a rutintalan tervező által sokszor „kifelejtett” működési módokat, mint például a *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.



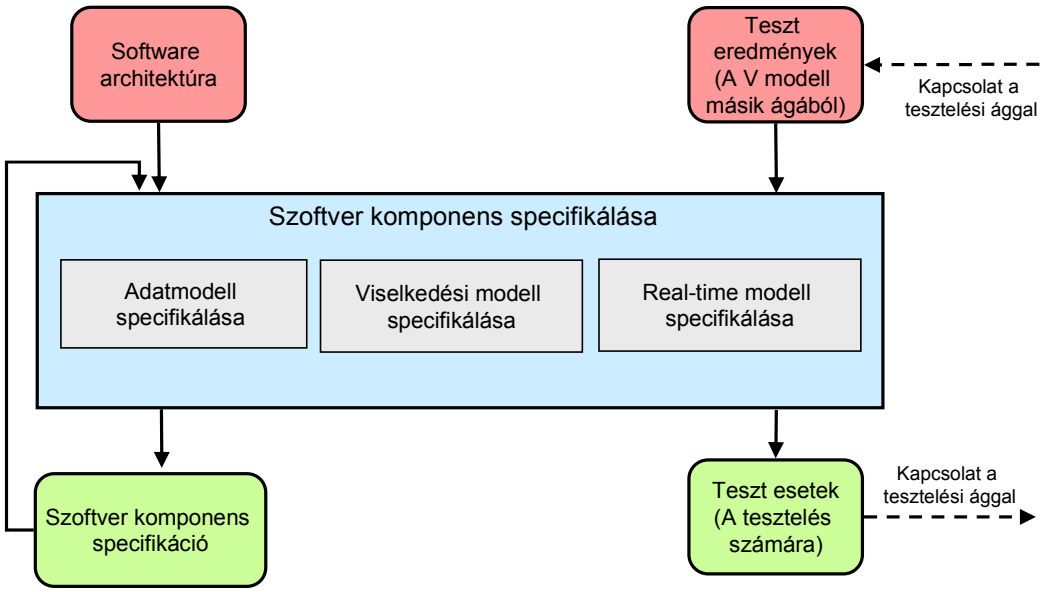
3.21 ábra Egy beágyazott rendszer tipikus működési módjai és azok kapcsolatai

3.2.4. Szoftver komponensek specifikálása

A Szoftver komponensek specifikálása lépés angol neve: ***Specification of software components.***

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.



3.22 ábra A szoftverkomponensek specifikációjának folyamata

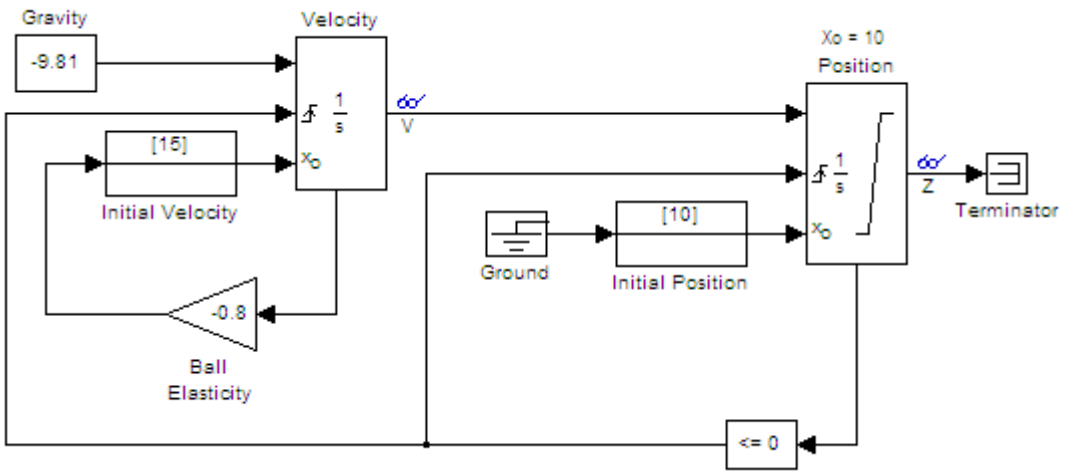
Az adatmodell meghatározása

A szoftver komponensek specifikációjának első lépése az adatmodell megadása. Az adatmodell tartalmazza, hogy az adott komponensnek milyen bemenetei és kimenetei vannak, és a bemenetekből hogyan állítódik elő a kimenet.

Ennek első lépése, hogy elvonatkoztatunk az adatok tényleges hardver implementációjától, és meghatározzuk, az adatok absztrakt ábrázolásának módját:

- Skalár értékek
- Vektorok, egydimenziós tömbök
- Mátrixok stb..

A következő lépés, az adatokon végzett műveletek specifikációja. Erre sokszor alkalmaznak valamilyen grafikus programozási nyelvet. Ilyen például az ASCET, Matlab Simulink, LabVIEW stb (3.23 ábra). Erre többnyire azért van szükség, mert ezek az adatmodellek a legtöbb esetben valamilyen szabályozástechnikai részhez, vagy más, igen speciális ismereteket ún. *domain specifikus* tudást igénylő részhez kötődnek. Ezekhez általában speciális tudású, nem feltétlenül villamosmérnök fejlesztőket alkalmaznak, akik nem járatosak a beágyazott rendszerek fejlesztésében, viszont a *Simulink*, *ASCET*, stb. *domain specifikus* nyelvekben otthonosan mozognak. Tehát ezek a *domain expert*-ek elkészítik és szimulálják a funkciót (pl.: szabályozás) és azt utána vagy egy programozó lekódolja, vagy automatikus kódgenerátorokkal kódot generálnak belőle a vezérlő számára. A félreértések elkerülése végett: általában nem az egész rendszer szoftverét generálják, hanem annak valamilyen speciális tudást igénylő részét, és utána azt integrálják össze a többi, általában kézzel írt kóddal (ez a módszer egyre szélesebb körben kezd elterjedni).



3.23 ábra Példa egy szabályozás adatfolyam ábrázolására Simulinkban

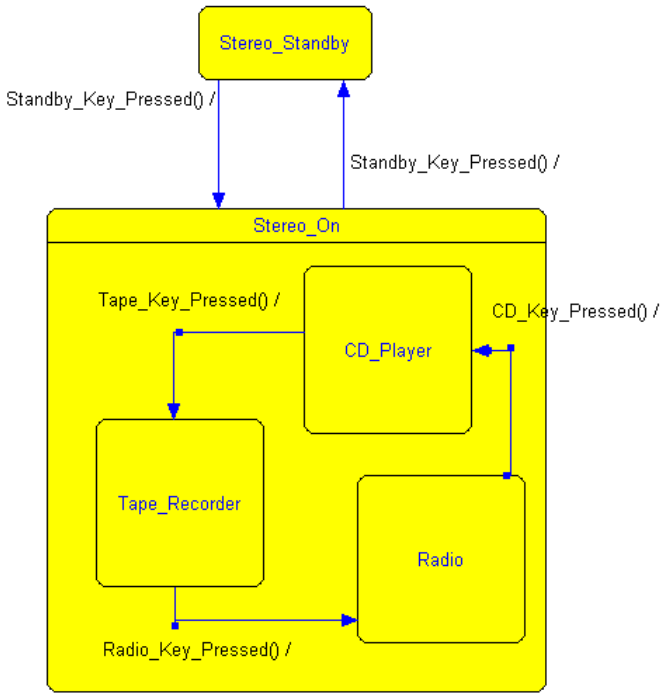
Az adat és a viselkedési modellt és azok verzióit néhány esetben élesen szét is választják külön adat és vezérlő szoftver verziókat létrehozva. Ezt a szétválasztást tipikusan csak azokra a részekre alkalmazzák, amelyek kizárólag az adatok megváltoztatásával új rendszer verziót képesek létrehozni. Ilyen 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).

A viselkedési modell meghatározása

Bár az adatfolyam gráfok egyszerűen ábrázolhatóak, mégsem hordoznak magukban minden információt a szoftver komponens viselkedéséről. Például, az adatfolyam nem írja le, hogy minek a hatására hajtódnak végre az egyes feldolgozó lépések, illetve meddig tartanak azok. Az ilyen leírások tipikusan a szoftver viselkedési folyamához tartoznak (figyelem, a LabVIEW például keveri a kétféle megjelenítést, és a Simulinkban is van rá mód, hogy az adatfolyam megjelenítés mellé a State flow-val viselkedési leírást is létrehozzunk.)

A vezérlési folyam sokszor valamilyen folyamatábrával, vagy struktrogrammal kerül leírásra. Ezeknek tartalmaznia kell a végrehajtás sorrendjét, az esetleges elágazásokat, az ismétléseket, iterációkat, illetve a többi komponenssel való kapcsolattartást.

A legelterjedtebb leírási mód az állapotgépes (FSM – Finite State Machine) leírás, legtöbbször az UML *State digram* formátumában, vagy ahhoz hasonló módon megadva. Elsősorban annak köszönhető az FSM alapú leírás elterjedése, hogy a modern eszközökkel az így létrejött leírásból közvetlenül kód generálható. Így a modellben formálisan verifikált működés egy bizonyítottan jól megírt kódgeneráló segítségével a valóságban is helyes működést tud garantálni. Számtalan ilyen állapot diagramból kódot generáló eszköz létezik a piacon, például IAR VISUAL State-je (3.24 ábra).



3.24 ábra IAR Visual State-ből létrehozott állapotgép.

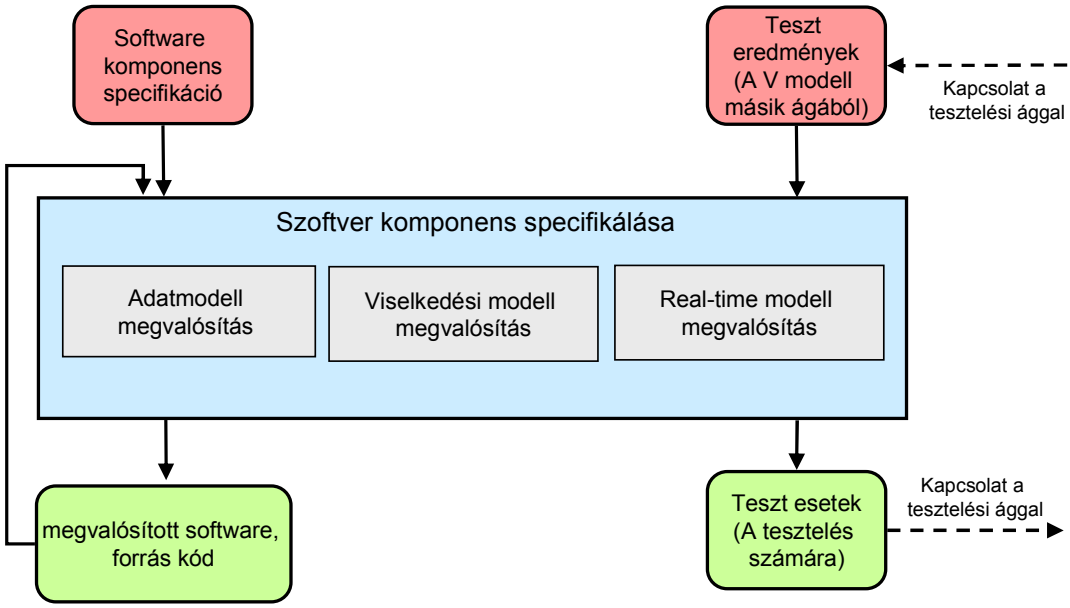
Az real-time modell specifikálása

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.

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). A *DMA*-t itt részletesen nem mutatjuk be, mert az szerepelt a múlt féléves: *„Valós idejű és biztonságkritikus rendszerek” (vimim151)* tárgyban.

3.2.5. A szoftver komponensek implementálása

Az implementálási folyamat során az előzőekben megtervezett adat és viselkedési modellt valósítják meg (3.25 ábra).



3.25 ábra A szoftver komponensek implementálása

Hardver limitációk

A megvalósítási folyamatnál az egyik fontos megkötést jelentek a hardver által szabott korlátok. A hardver költség a legtöbb iparágban (például az autóiparban) 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.

3.2.6. A megfelelő implementációs környezet kiválasztása

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 (3.26 ábra). Ezeknek a szabályoknak az alapjaival célszerű bővebben megismerkedni, mert szinte minden cégnél alkalmaznak ilyeneket.

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	+	+	+	++

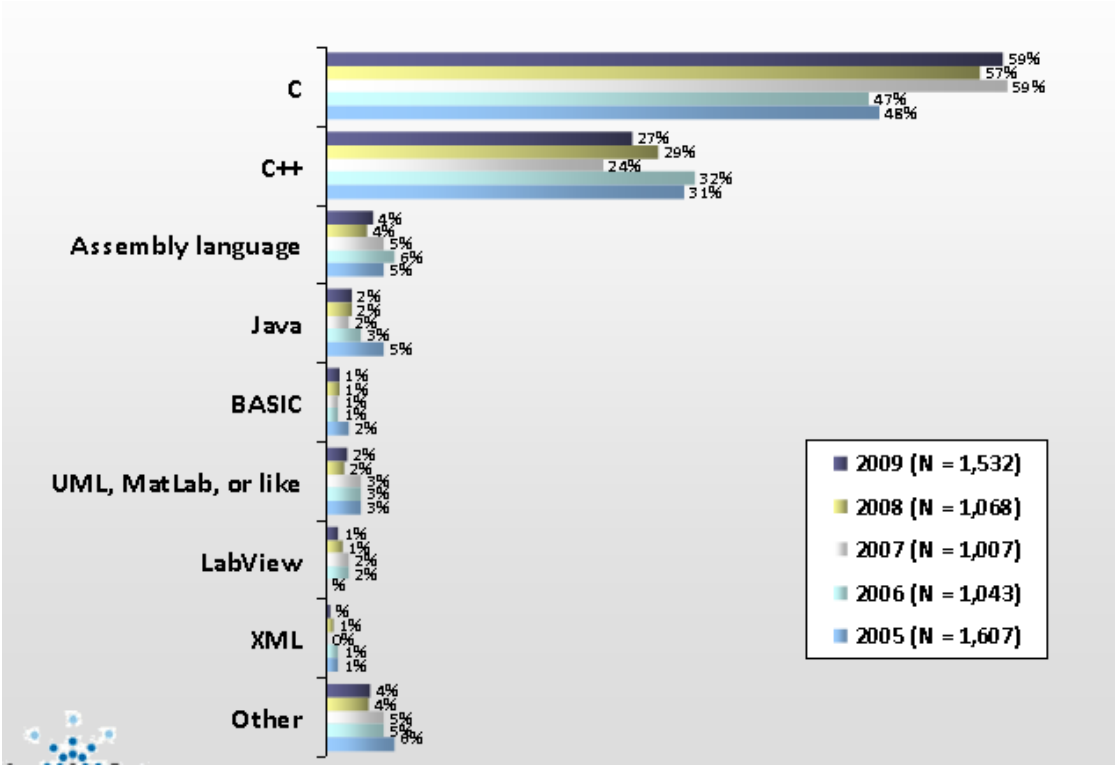
Jelmagyarázat:
"-": tilos
"0": nem szükséges
"+": ajánlott
"++": kötelező

3.26 ábra Minta az implementációs szabályokra a SIL réteg függvényében.

A zölddel kijelölt megkötésekről, mint a dinamikus változók kerülése, pointerek korlátozott használata stb. már sokat hallhattunk. Ami érdekesebb az a narancssárga csoport, ami – ahogy láthatjuk – a megbízhatóságossági szinttől függetlenül kötelező, de kevesebb szó esett erről a területről az eddigi tanulmányok során. Ebben a fejezetben ezeket a szabályokat vesszük át részletesebben.

Megfelelő programozási nyelv használata

A beágyazott rendszereknél a mai napig a C programozási nyelv a legelterjedtebb, és várhatóan még egy jó darabig ez így is lesz. Ezt támasztja alá az *Embedded Market Study 2009* felmérés eredménye is (3.27. ábra). Bár azt gondolhatnánk, hogy ezek után elég azt mondanunk, hogy C-t kell használni. Figyelembe kell azonban venni, hogy a C-nek is több kicsit eltérő változata létezik, és az egyes fordítók sem teljesen egyértelműen kezelik ezeket a különbségeket. Tehát a nyelv kiválasztásán túl a legtöbb cégnél pontosan rögzítik a nyelv verzióját is (például ISO/IEC 9899:1999), és ellenőrzik a fordítók adott verzióval való kompatibilitását is.



3.27 ábra Az Embedded market study 2009 felmérése a beágyazott rendszeres fejlesztéseknél használt programozási nyelvekről

Coding style guide és elnevezési szabályok használata

A fejlesztés során a legtöbb cégnél használnak ún. *Coding,* vagy *Programming Style Guide*-okat. Ezek többnyire cég specifikusak. Az interneten számos ilyen *Style Guide*-ot találhatunk.

Ezek a következő területeket fedik le: (Mindegyik területhez egy-egy mintát is mutatunk, de tartsuk szeme előtt, hogy ez nagyon szervezet-, cégfüggő.)

File struktúra és file elnevezési konvenció

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.

Program file-ok szerkezete

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.

A kommentek szerkezete

Minden cégnél alkalmaznak kommentezési stílus szabályokat. Nagyon sok helyen használják a gyakorlaton is bemutatott ***Doxygen*** kommentből dokumentációt generáló programot és annak kommentezési stílusát. *Fontos megjegyezni, hogy szinte mindenhol az angol nyelvű kommentezést és változó elnevezést követelik meg, nem engedélyeznek más nyelvet.*

Deklarációs és elnevezési szabályok

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_nWheels` : member of a global namespace, integer
- `m_nWheels` : member of a structure/class, integer
- `m_wheels` : member of a structure/class
- `s_wheels` : static member of a class
- `_wheels` : local variable

A *Hungarian notation* mellett és ellen is szólnak érvek, lévén gyakorlatilag redundáns információt tartalmaz, de az kétségtelen, hogy formalizálja a változó elnevezést, és már önmagában emiatt sok helyen használják.

A nyelvi készlet korlátozása

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.

MISRA C (Motor Industry Software Reliability Association)

Az első MISRA C szabályverziót 1998-ban hozták létre. 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.

CERT C Secure Coding Standard

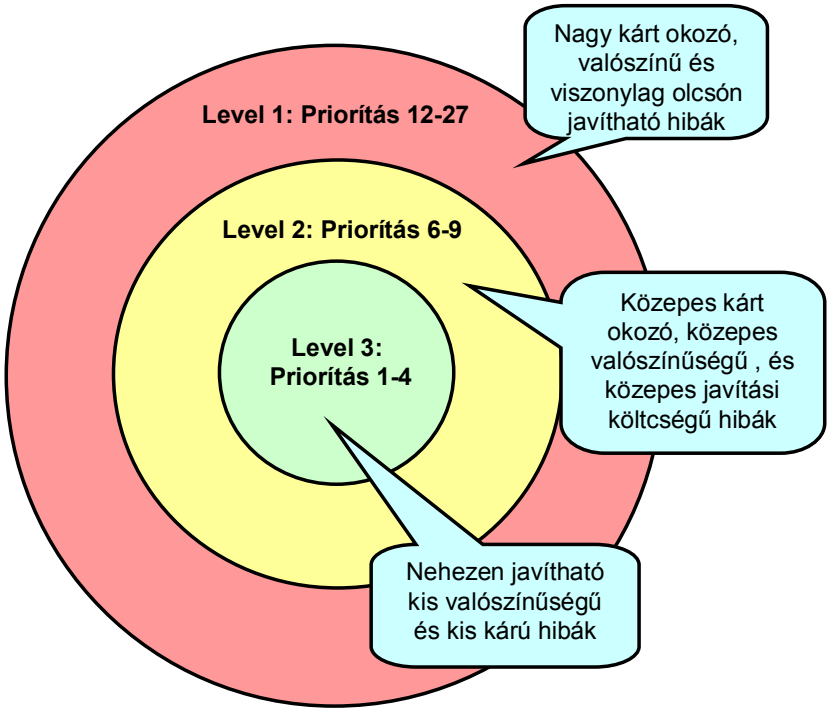
A CERT: Carnegie Mellon University Software Engieenring Institute, amely tagja a SEI-nek (Software Engieenering Institute). Az intézet által kidolgozott *Secure coding* szabványok a következő nyelvekhez jöttek létre: C, C++, JAVA.

A Cert Secure C prioritási osztályokba sorolja a szabályait. Az osztályokat három alap szempont szerint határozzák meg:

- **Severity:** A szabály be nem tartása által kiváltott hiba kár mértéke.
  - Low (alacsony)
  - Medium (közepes)
  - High (magas)
- **Likehood:** Mekkora a valószínűsége, hogy a szabály be nem tartása esetén a hiba jelentkezik.
  - Unlikely (valószínűtlen)
  - Probable (lehetséges)
  - Likely (valószínű)
- **Remediation cost:** Milyen költséges hogy a szabályt betartsuk.

- High (manual detection and correction)
- Medium (automatic detection, manual correction)
- Low (automatic detection and correction)

A prioritási osztályokat ezeknek a kategóriáknak a vizsgálatával három szintbe sorolták a Level1, Level2, Level3-ba (3.28 ábra).



3.28 ábra A CERT Secure C priorításai és a szintek közötti összefüggések

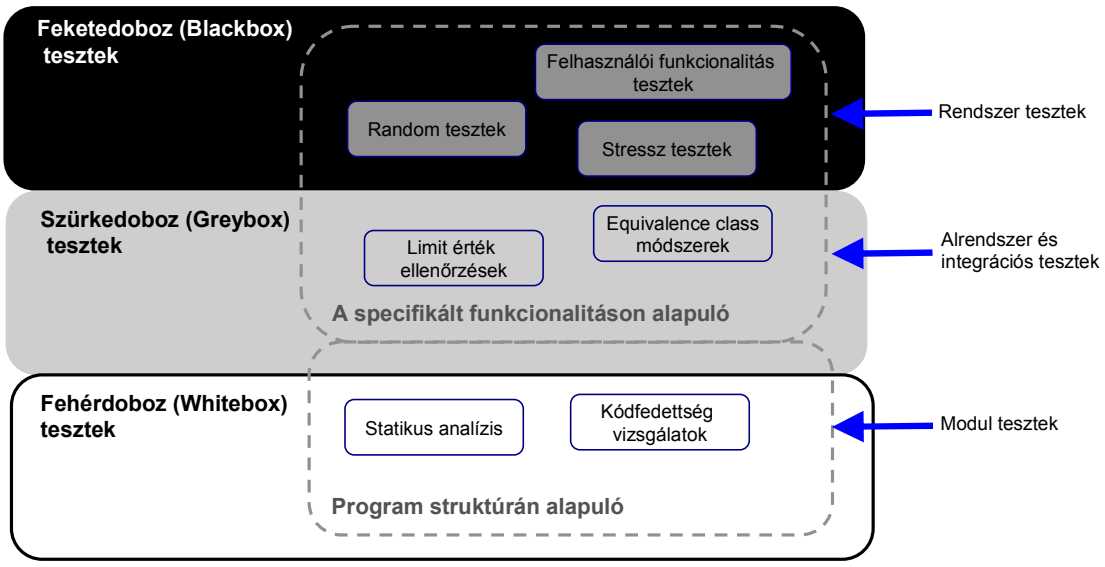
A CERT C szabályok az Interneten szabadon hozzáférhetőek, Mindegyik szabálynál mellékelik annak prioritását is.

01. Preprocessor (PRE)
02. Declarations and Initialization (DCL)
03. Expressions (EXP)
04. Integers (INT)
05. Floating Point (FLP)
06. Arrays (ARR)
07. Characters and Strings (STR)
08. Memory Management (MEM)

3.3. A V-modell tesztelési ága

Ez a fejezet a V-modell tesztelési ágának az áttekintését szolgálja. A fejezet szándékosan rövid és áttekintő jellegű, mert ezzel a tématerülettel Majzik István a *„Valós idejű és biztonságkritikus rendszerek”* tárgyban már részletesen foglalkozott.

Az egyes V-modell szerint végrehajtott tesztelési lépések előtt a 3.29 ábra egy gyors áttekintést ad a különböző szoftvertesztelési módszerekről és azok alkalmazási pontjáról az életciklus modellben.

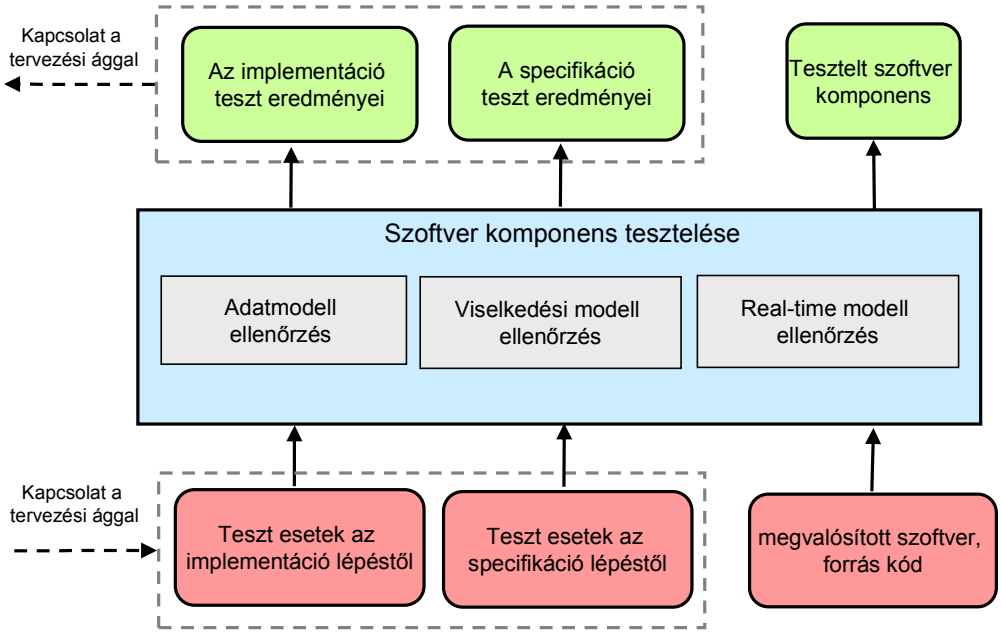


3.29 ábra A tesztelési módszerek áttekintése

Az egyes V-modell lépéseknél gyakorlatilag ezeket a módszereket fogjuk picit részletesebben áttekinteni.

3.3.1. Szoftverkomponensek tesztelése

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



3.30 ábra Szoftverkomponens tesztelése

Adatmodell ellenőrzés

A szoftver komponensek tesztelésénél az első lépés az adatmodell ellenőrzése. 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. A 3.32 ábra bemutatja, hogy ezt az analízist függetlenül a megkívánt megbízhatóságtól végrehajtják.

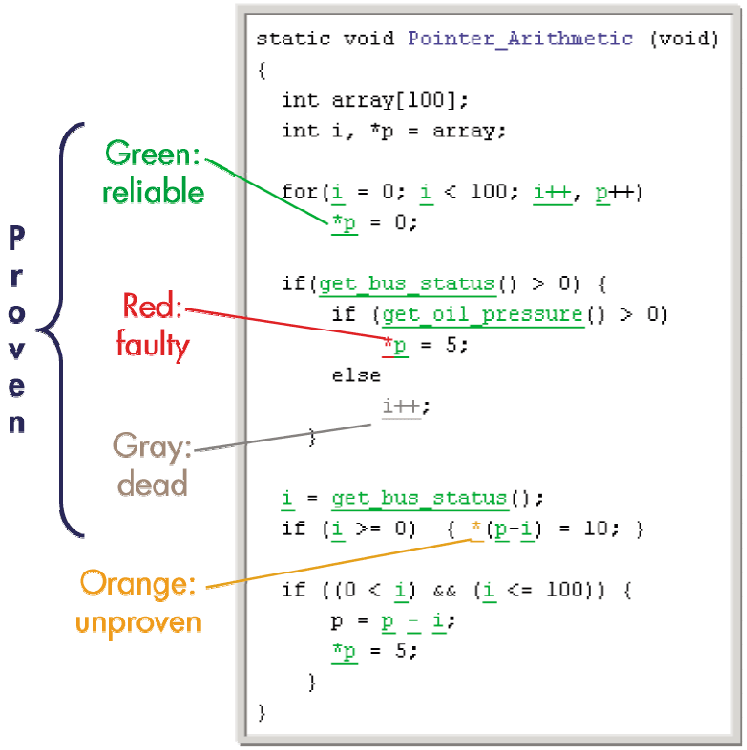
Tevékenység	SIL0	SIL1	SIL2	SIL3
Kódig guidleine-nak való megfelelőség ellenőrzése	++	++	++	++
Statikus kódanalízis	++	++	++	++

Jelmagyarázat:
"-": tilos
"0": nem szükséges
"+": ajánlott
"++": kötelező

3.32 ábra Statikus kódanalízis szükségessége a SIL réteg függvényében

Egy ilyen széles körben elterjed statikus kódanalizátor például a Polyspace. A Polyspace a MathWorks cég terméke és a C/C++ valamint Ada nyelvhez nyújt statikus kódanalízis szolgáltatást. Beágyazott rendszerek esetében külön kedvelt az a funkciója, hogy a MISRA-C szabályok ellenőrzésére is képes.

A Polyspace az általa ellenőrzött kódrészletben az egyes változókat különböző színkódokkal látja el annak függvényében, hogy azok hordoznak-e valamilyen potenciális veszélyt (3.33. ábra).



3.33 ábra Polyspace elemzés

A színkódok jelentése a következő:

- **Zöld**: bizonyítottan helyes minden körülmények között
- **Piros**: bizonyítottan rossz minden körülmények között
- **Szürke**: bizonyítottan elérhetetlen kód
- **Narancs**: bizonyos körülmények között lehet rossz

A változók ilyen statikus elemzése úgy valósul meg, hogy a Polyspace a program összes változójának lehetséges értékeit egy absztrakt interpretációnak nevezett módszer segítségével követi. A Polyspace a következő hibákat képes a *statikus analízissel* felderíteni:

- Osztott váltózók kezelése (multi taszkból adódó problémák)
- Tömbindexelési hibák detektálása
- Hibás pointer hivatkozások
- Inicializálatlan változó olvasása
- Hibához vezető aritmetikai eljárások (nullával való osztás, gyökvonás negatív számból)
- Float vagy Integer változók alul, felülcsordulása
- Illegális típuskonverziók
- Elérhetetlen kód
- Végtelen ciklusok

Polyspace a MISRA-C megfelelőséget a 102 szabályra teljesen 20 szabályra részletesen képes ellenőrizni (A MISRA szabályok egy jó része a környezetre vonatkozik, amelyek nem

ellenőrizhetőek egy ilyen analízis során). Az MISRA szabályok ellenőrzését egyesével be lehet engedélyezni, vagy tiltani a programban.

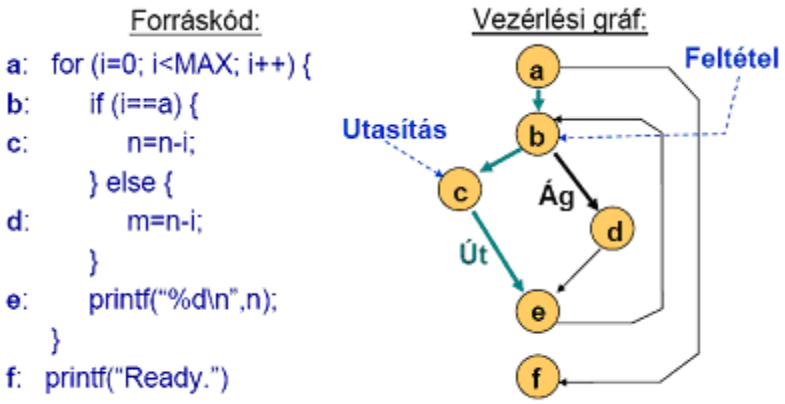
A Polyspace, mint az összes statikus elemző igen költséges program, így a gyakorlatok során nem volt alkalom bemutatni. Ugyanakkor érdemes megjegyezni, hogy egy másik statikus analizátornak a Gimpel-nek létezik egy tesztoldala, ahova rövid kódrészleteket lehet beilleszteni ellenőrzésre ezzel demonstrálva a tool képességeit. Ezt az oldalt érdemes lehet felkeresni: http://www.gimpel-online.com/OnlineTesting.html

A viselkedési modell ellenőrzése: Kódfedési vizsgálatok

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



3.34 ábra Példa egy kódrészlethez tartozó vezérlési gráfra

A különböző fedettségi vizsgálatok kicsit részletesebben:

- **Utasítás lefedettség:**
  - Tesztelés során végrehajtott utasítások száma / Összes utasítás száma
  - Nem szigorú:
    - Utasítások kihagyási feltételeit nem veszi figyelembe
    - pl. k=0; if (a>0) k=1; m=1/k; lefedése esetén az a=0 eset vizsgálata kimarad

- **Döntési ág lefedettség:**
  - Tesztelés során végrehajtott döntési ágak száma / Összes lehetséges döntési ág száma

- Nem szigorú:
  if (a>0 && (safe(c) || safe(b)) start();
  else stop();
  A fordító optimalizálása miatt a safe(b) hívás be sem következik, ha safe(c) igaz.

- **Feltétel lefedettség:**
  - Feltételekben a tesztelt bemeneti kombinációk száma / Feltételek összes bemeneti kombinációinak száma
  - Aránylag bonyolult tesztelés a sok lehetséges kombináció beállítása miatt.

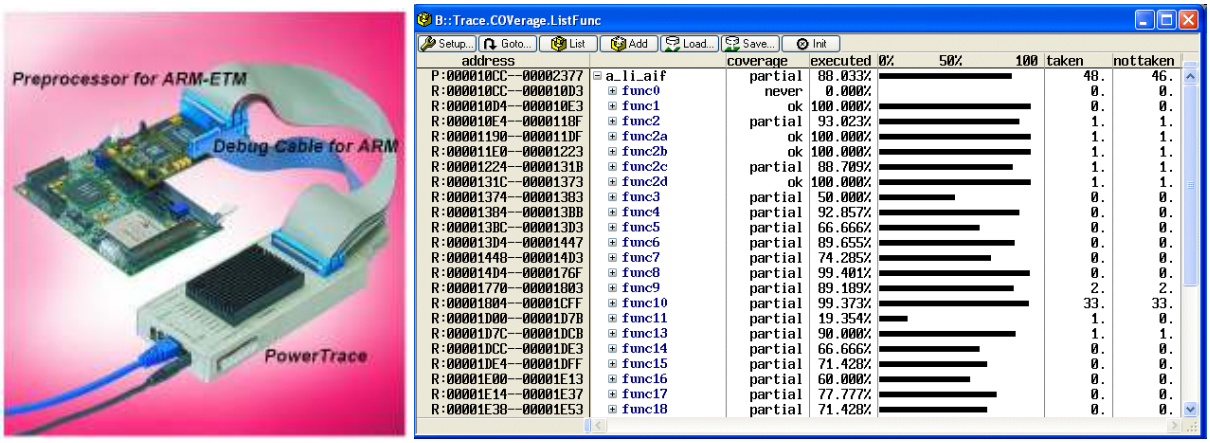
- **Út lefedettség:**
  - Bejárt független utak száma
  - Összes független út száma
  - Független utak: van olyan pont vagy él, ami egy másik útban nincs meg
  - Jellemzés: A gráf ciklomatikus komplexitása (CK): A független utak maximális száma.
    CK(G)=E-N+2, ahol
    E: élek száma, N: pontok száma a G vezérlési gráfban
  - 100% út lefedettség együtt jár:
    - 100% utasítás lefedettség
    - 100% ág lefedettség

A kódfedettségi vizsgálatok elvégzésére alapvetően két lehetőségünk van. Az egyik lehetőség egy pusztán szoftveres mérés a *kód felműszerezés*ével. 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 például a GCC toolchainhez tartozó GCOV, ami elvégzi a kód felműszerezését és a futtatás alatt készít egy logfile-t, amiből meg tudja határozni, hogy az egyes sorok hányszor futottak le. Bár a GCOV alapvetően nem beágyazott rendszerekre tervezett (logfile-okat hoz létre), de kis átalakítással azoknál is használható.

```
#include <stdio.h>
int main (void)
          {
     1    int i;
    10    for (i = 1; i < 10; i++)
            {
     9        if (i % 3 == 0)
     3          printf ("%d is divisible by 3\n", i);
     9        if (i % 11 == 0)
######          printf ("%d is divisible by 11\n", i);
     9      }
     1    return 0;
     1  }
```

3.35 ábra Példa a GCOV eredményekre: a bal oldalt látható szám mutatja, hogy hányszor futott le az adott kódsor.

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. Ilyen *trace* blokk például az ARM magoknál az *Embedded Trace macrocell.* Ezekre a hardware-es támogatásokra épülve komoly és drága eszközökkel, mint a *Lauterbach* trace kit-jei kódfedési vizsgálatok és real-time viselkedést ellenőrző tesztrendszereket lehet felépíteni.



3.35 ábra Hardware-esen támogatott kódfedettség mérés: Lauterbach trace eszközök

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 (3.36 ábra).

Tevékenység	SIL0	SIL1	SIL2	SIL3
Utasítás fedésvizsgálat	+	+	++	++
Döntési ág fedésvizsgálat	+	+	+	++
Feltétel lefedési vizsgálat	+	+	+	+
Utak fedési vizsgálata	+	+	+	+

Jelmagyarázat:
"-": tilos
"0": nem szükséges
"+": ajánlott
"++": kötelező

3.36 ábra Ajánlás a kódfedési vizsgálatok elvégzéséhez

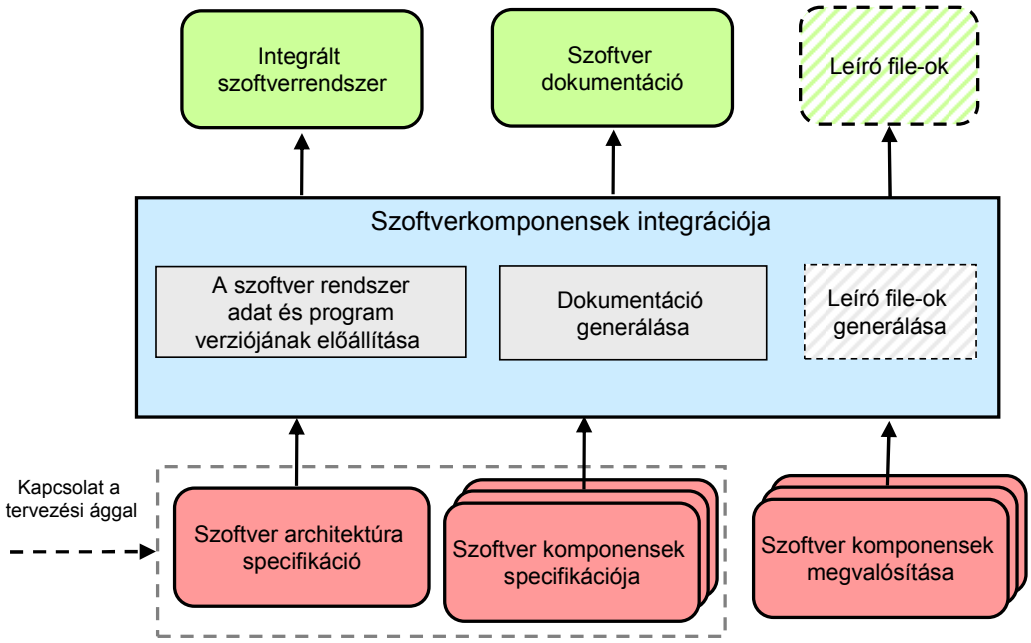
A real-time modell ellenőrzése

A real-time modell ellenőrzésénél az egyes szoftver részek tényleges futási idejét szokták lemérni. Ez jó pár esetben történhet a *trace* toolok segítségével. Ami itt, és egy lépéssel később a szoftver rendszer integrálásánál pluszt szokott jelenteni az az RTOS nyomon követése és felműszerezése. A legtöbb beágyazott operációs rendszer ugyanis rendelkezik

olyan *trace makrókkal,* amelyek segítségével a pontos működése és ezáltal a rendszer állapotának a követése ellenőrizhető (lásd FreeRTOS trace hook-ok).

3.3.2. Szoftverkomponensek integrációja

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.



3.37 ábra. A szoftverkomponensek integrációja lépés

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.

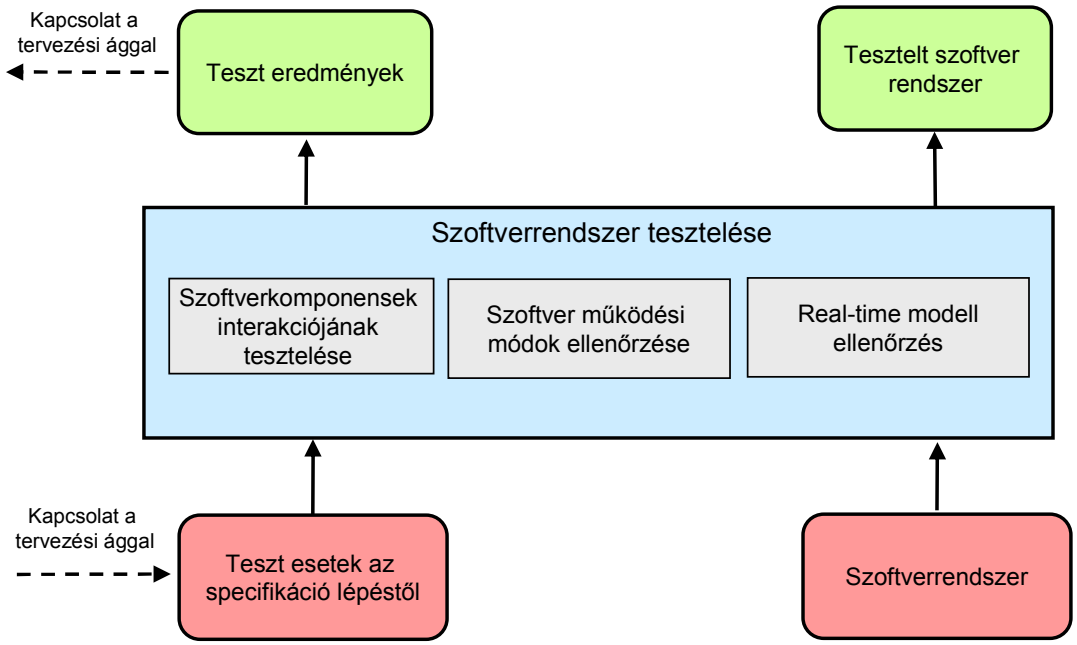
A dokumentáció generálásra jó példa a számos cégnél alkalmazott és a *coding style guide*-dokban rendszeresen hivatkozott Doxygen használata (a Doxygen használatához a 4.2 fejezetben kaphatunk útmutatót).

A különböző leíró file-ok előállítása már jóval alkalmazásfüggőbb és opcionális folyamat. Ilyen leíró file-ok lehetnek például az autóiparban rendszeresen alkalmazott ASAM-MCD2 file-ok (Association for Standardisation of Automation and Measuring Systems - Measurement and Calibration Data Exchange Format). 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 olyan eszközökkel, mint például az INCA, vagy CANape a leíró file-oban megadott fontosabb szoftver paraméterek online nyomon követhetők, vagy akár módosíthatók kalibrálhatók is.

3.3.3. A szoftverrendszer tesztelése

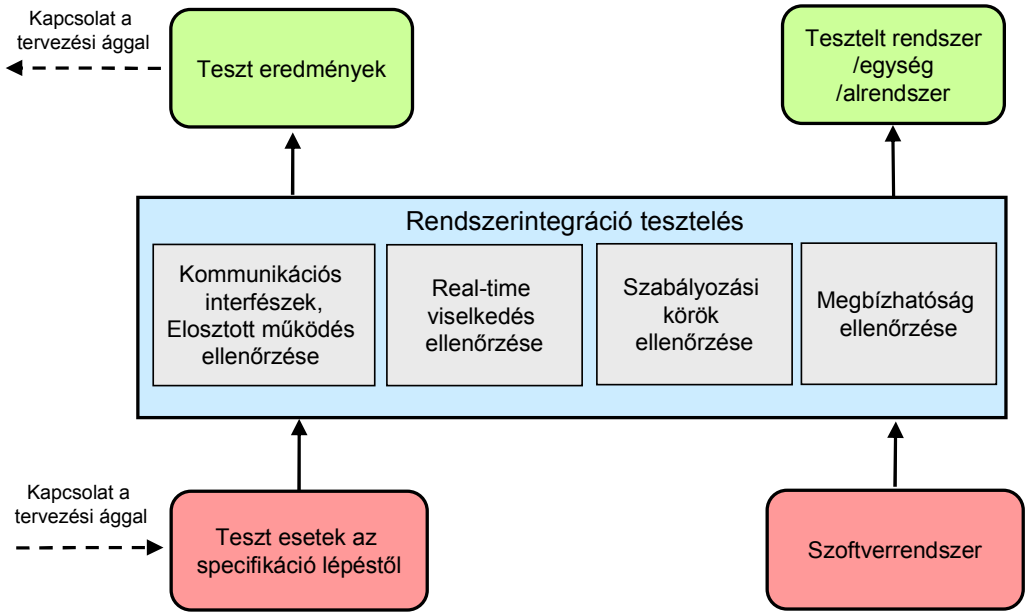
A szoftverrendszer tesztje tartalmazhatják azokat a módszereket, amelyeket a komponenstesztelésnél bemutattunk, például kódfedettség vizsgálatokat, és tipikusan ezeket egészítik ki különböző szürkedoboz alapú tesztekkel, amik például egy *SIL (Software Int he Loop)* tesztnél az egyes komponensek együttműködését ellenőrzik.



3.38 ábra. A szoftverrendszer tesztelése lépés

3.3.4. Rendszerintegráció és integráció tesztelése

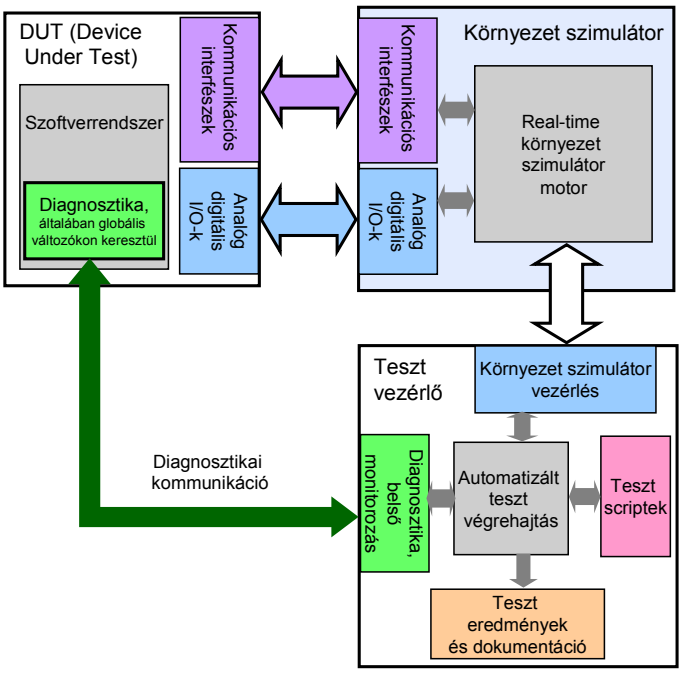
Ennél a lépésnél a szoftver és a hardware már mindenképpen integrálódott, és itt ennek az integrációját tesztelik. A teszt célja az, hogy a rendszer külvilággal való kapcsolatát ellenőrizzék: megfelelő külső eseményekre megfelelő válaszokat ad-e.



3.39 ábra. A rendszerintegráció tesztelése lépés

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.

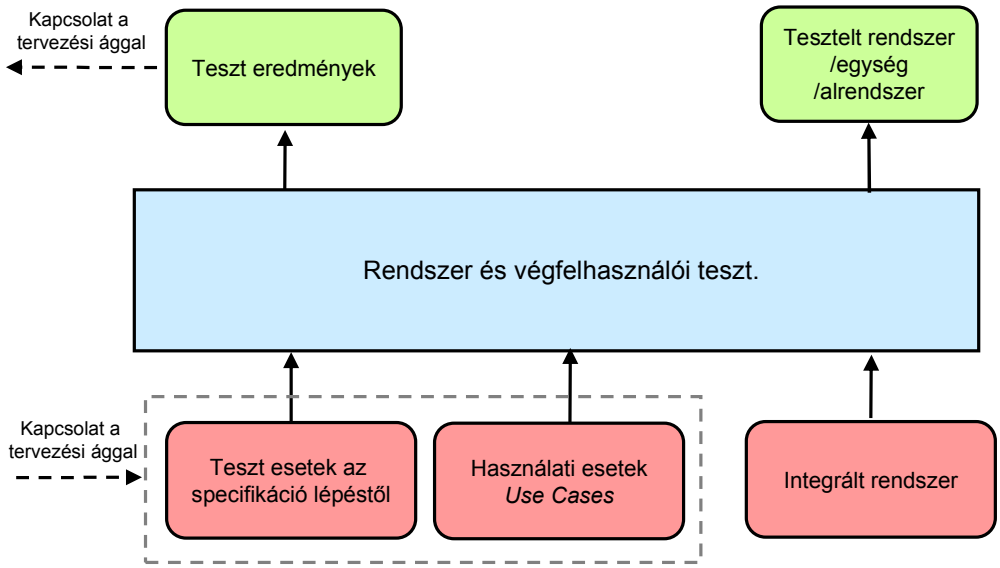


3.40 ábra. Tipikus feketedoboz, szürkedoboz tesztkörnyezet

Ezeknél a teszteknél kapnak szerepet az leíró file-ok, amelyek a 3.40. ábrán bemutatott és Zöld színnel jelölt diagnosztikai kommunikáció alapját adják.

3.3.5. Végfelhasználói teszt

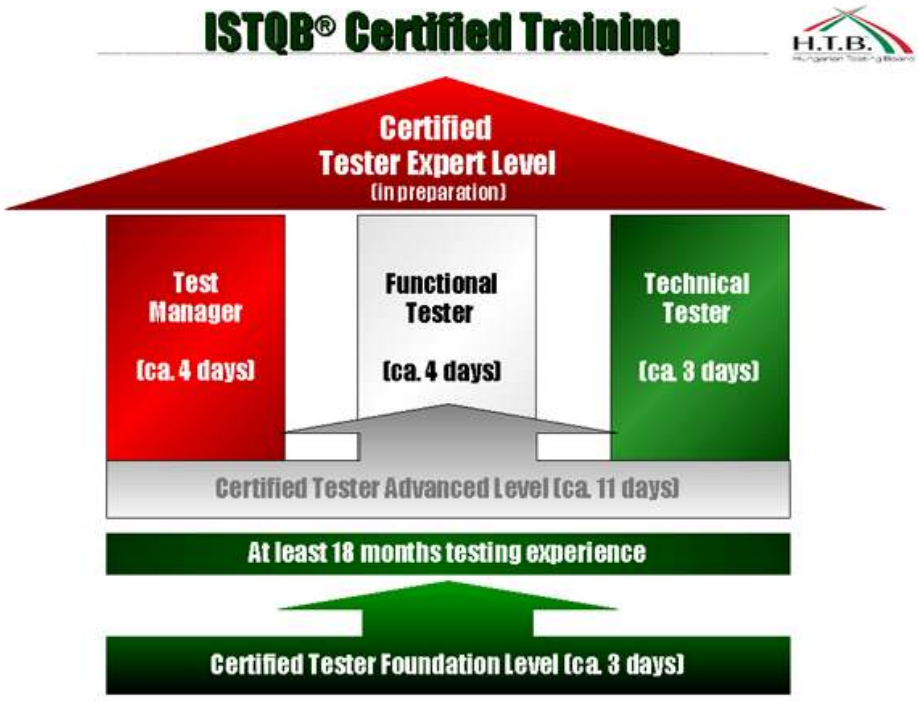
A tesztelés utolsó lépése az ún. végfelhasználói teszt, ahol a vevő ellenőrzi, hogy számára megfelelő termék jött e létre. Ezek a tesztek tipikusan az *Logikai rendszer architektúra* tervezésénél létrehozott használati eset leírásokon alapul *(Use Cases)*.



3.41 ábra. A rendszer- és végfelhasználói-teszt lépés

3.3.6. A tesztelők minősítése

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.



3.42 ábra. Hungarian Testing Board által kiadott minősítések összefoglalása.

Az ISTQB által minősített vizsgák nem egy fokozatúak a Hungarian Testing Board által kiadott minősítések összefoglalását a 3.42 ábrán követhető.

4. Gyakorlati anyagok

Ebben a részben az egyes gyakorlati foglalkozásokon tárgyalt anyagrészek kerülnek ismertetésre.

4.1. Verziókövetés SVN-el

4.2. Dokumentációgenerálás kommentből Doxygen-el

4.3. C kódolási szabályok: MISRA-C, CERT Secure C

Ebben a fejezetben a C nyelvi készletének korlátozása legelterjedtebben alkalmazott két szabálykészletet tekintjük át röviden néhány példa segítségével. Ez a két szabálykészlet a MISRA-C, és a CERT Secure C szabálycsomag.

Mielőtt elkezdenénk a szabálykészletek áttekintését fontos hangsúlyozni, hogy a példák és ez az egész fejezet is csak egy ízelítő szemelvény. Mindkét szabálykészlet önmagában meghaladja a 100 oldalt.

4.3.1. MISRA-C szabályok

A szabvány jelenlegi verziója a MISRA-C 2004 121 *Mandatory* (kötelező) és 20 Advisory (javasolt) szabályt tartalmaz a C nyelv leszűkítésére. Ezek a szabályok a következő nagyobb témacsoportokba lettek rendezve:

Environment (környezettel kapcsolatos előírások)
Ezek a megkötések elsősorban a fordítóval és a C nyelv verziójával kapcsolatosak. Például a MISRA-C megköti, hogy: az összes C kód az ISO9899:1990-es szabvány definícióit kell, hogy kövesse. Többek között szabályozásra kerül az is, hogy egy projectben több nyelv, vagy compiler csak akkor használható, ha a létrejövő object kód közös standardon alapul. Ez elsőre szőrszálhasogatásnak tűnik (egyébként a MISRA-C tele van ilyenekkel), de gondoljunk csak bele, hogy például a függvényhívásoknál az egyes argumentumok átadásánál a stack használatának módja lehet ilyen compiler függő dolog, ami igen kínos eredményekkel járhat.

Language extension (nyelvi kiterjesztések)
Ez blokk a kód kommentezésével és az esetleges assembly betétekkel foglalkozik. Például:

- Definiálja, hogy amikor szükséges assembly nyelvű betéteket használni, akkor ezeket jól egységbe zárva makróként kell megvalósítani:

```
#define NOP asm("NOP")
```

  Ez azért is fontos, mert portolásnál, vagy más fordító használatánál ezeket a definíciókat könnyebb cserélni, mint az összes assembly betétet átírni.

- Specifikálásra kerül az is, hogy csak a `/* ... */` típusú kommenteket lehet használni (Az gyakran használt `//` stílusú komment nem része a C-nek csak a C++-nak ebből néha adódhatnak problémák).

- Kódsorok nem lehetnek kikommentezve. Ez az egyik leggyakoribb hiba, amit az ember elkövet, hiszen fejlesztés során a debugg részt, és próba sorokat így a legegyszerűbb kikapcsolni. Ugyanakkor 1-2 hónap után visszanézve a kódot már nem fogjuk tudni, hogy ezeket a sorokat miért kommenteztük ki. A kódsorok kikommentezése továbbá lehet nem szándékos is, ami ellen a szabály ellenőrzésével védekezni tudunk.

```
/* Lezáratan komment
megbízhatóságkritikus_függvény();
/* Ez egy nem végrehajtott a megbízhatóság
szempontjából kritikus függvény volt */
```

  Az ilyen opcionálisan használt részeket az erre szolgáló `#ifdef` feltételes fordítási paranccsal kell blokkokban zárni.

Documentation (Implementációfüggő viselkedés dokumentálása)

Ez a szabálycsoport a fordítók implementációjától függő viselkedés dokumentálására ad specifikációt. Ilyen jellegű szabályozások például a következők:

- Karakter konstansok és string literálok karakterkódolási szabványát rögzíteni kell.

- 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.

- A felhasznált könyvtári függvények és külső szoftvermodulok MISRA-C kompatibilitását ellenőrizni és dokumentálni kell.

Identifiers (Azonosítók)

Ez a rész a különböző változó/függvény azonosítókkal kapcsolatos szabályokat tartalmazza.

- Az azonosítóknak az első 31 karakterben különbözniük kell. Lehetőleg kerülni kell egy karakteres különbséget. Figyelmet kell fordítani arra, hogy a különbség ne csak könnyen összetéveszthető karakterekre korlátozódjon, mint az: egy 1, l betű; nulla 0, O betű; kettő 2, Z betű; öt 5, S betű.
- Kerülni kell a változók túlterhelését, belső láthatósági körű változónak nem lehet ugyanaz a neve, mint egy külső láthatósági körűnek, mert ez zavart okozhat.

```
int8_t i;
function()
```

```
{
 int8_t i;

 i = 3;
}
```

A példában egyértelmű, hogy a függvényben létrehozott változóra hivatkozunk, de ez egy áttekinthetetlenebb kódnál megtévesztő és félrevezető lehet (kiváltképp egy tapasztalatlanabb programozó számára).

- Tanács: semmilyen névazonosítót ne használjunk újra beleértve a (***typedef***-ekben használt, ***static*** kulcsszóval ellátott függvényeket és változókat, valamint a ***struktúra elemek***)

Példa arra, hogy milyen gondot okozhat, ha struktúra neveknek ugyanúgy nevezünk:

```
struct air_t
{
 uint16_t speed;      /* m/s-ban */
} air;

struct vehicle_t
{
 uint16_t speed;      /* km/ó-ban */
} vehicle;

air_resistent_factor = air.speed + vehicle.speed;
```

Types (Típusok):

- 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 */
```

(A GCC-ben például egy fordítási opció az, hogy egy *char* az *unsignednek*, vagy *signedned* értelmezett)

- Typedef segítségével minden elemi változótípushoz olyan azonosítót kell létrehozni, amelynek a neve tartalmazza annak előjelességét és a hosszát.

Ezek a definíciók függhetnek a processzor típusától, de a jobb oldali konform elnevezésnek architektúra függetlennek kell lennie. Példa egy 32bites uC esetében:

```
typedef   char              char_t;
typedef   signed char       int8_t;
typedef   signed short      int16_t;
```

```
typedef   signed int          int32_t;
typedef   signed long         int64_t;
typedef   unsigned char       uint8_t;
typedef   unsigned short      uint16_t;
typedef   unsigned int        uint32_t;
typedef   unsigned long       uint64_t;
typedef   float               float32_t;
typedef   double              float64_t;
```

Constant (Konstansok):

- Oktális konstansokat nem használunk

```
code[0]  =  109; /* decimális 109 */
code[1]  =  052; /* decimális 42 */
```

Declarations and definitions (deklarációk és definíciók)

- Minden függvénynek kell, hogy legyen prototípusa, ami látszik a függvény megvalósítás és függvényhívás számára is. A megvalósítás és a prototípus definíciónak pontosan meg kell egyeznie. Általában a *header* file-ban kell lennie a prototípusnak, amit aztán mindenki *#include*-olhat.

- Minden esetben, amikor változót, vagy függvényt definiálunk, azt teljes precíz megadással kell tennünk.

```
extern x;                        /* nem szabványos */
extern int16_t x;                /* szabványos */

const y;                         /* nem szabványos */
const uint16_t y;                /* szabványos */

static function(void);           /* nem szabványos */
static int16_t function(void);   /* nem szabványos */
```

- Header file nem tartalmazhat függvény vagy változó definíciót, csak deklarációt. Definíciót mindig C file-ban kell megírni.

- Ha egy változót, vagy függvényt nem használunk, a file-on kívül használjunk a *static* kulcsszót. Static kulcsszóval blokkon belül is lehet globális változót definiálni, de azt nem használhatjuk a blokkon kívül.

- External kulcsszóval hivatkozott objektum, vagy függvény csak egy helyen lehet deklarálva, és definiálva is. Ez az egy hely tipikusan egy *header* file, amit mindenki be tud include-olni akinek szüksége van az adott változóra.

  ***kulso.h** header file tartalma:*

```
extern int16_t kulso;
```

A változó definiálása:

```
#include "kulso.h"
int16_t kulso = 0;
```

Általános tanács, hogy ne használjunk *extern* változókat, azokhoz a változókhoz, vagy struktúrákhoz, vagy tömbökhöz, amelyeket más *file*-okból kezelni akarunk hozzunk létre kezelő függvényeket.

Initialisation (Inicializáció)

- Az összes automatikusan foglalt helyű változót inicializálni kell használat előtt.
- Többelemű tömbök, struktúrák inicializálásakor {}-ekkel kell jelölni a határokat inicializáláskor is.

```
int16_t x[3][2] = {1,2,3,4,5,6};   /* nem szabványos */
int16_t x[3][2] = {{1,2},{3,4},{5,6}}; /* szabványos */
```

- Felsorolásos típus inicializációjánál vagy csak az első, vagy az összes tagot =-el kell inicializálnunk

```
enum color {red = 1, blue, green, yellow = 3};
/* nem szabványos a yellow-nak, és a green-nek ugyanaz az
értéke */

enum color {red = 1, blue=2, green=3, yellow = 3};
/* szabványos a yellow-nak, és a green-nek ugyanaz az
értéke, de mi akartuk így (vagy elnéztük de akkor is
szabályosak voltunk☺) */
```

Arithmetic type conversion (Artimetikai és típus konverziók)

Ez a rész a legnehezebb, és egyik legtöbb gondot okozó terület. Változó konverzió a következő problémákkal járhat:

- Értékvesztéssel, ha a konverziós típus nem tudja a konvertált típus teljes értéktartományát megjeleníteni.
- Előjelvesztéssel
- Precízió vesztéssel

Általában ezért csak kisebb tartományból nagyobb tartományba való konverziót támogatunk.

- Előjel nélküli konstansok után mindig ki kell rakni az U betűt.
- A MISRA-C a következő változó konverziós szabályokat adja meg implicit konverzióra:

- Nincs implicit konverzió signed és unsigned típusok között
- Nincs implicit konverzió egész és lebegőpontos típusok között
- Nincs implicit konverzió nagyobb tartományról kisebb tartományra
- Nincs implicit konverzió függvény argumentumoknál
- Nincs implicit konverzió return értékeknél
- Nincs implicit konverzió komplex aritmetikai kifejezéseknél

- Explicit konverzió akkor használható komplex aritmetikai kifejezés esetében, ha kisebb értéktartományú típusra konvertálunk, de az előjelet megtartjuk. Lebegőpontos típust csak kisebb lebegőpontos típussá lehet komplex kifejezés után konvertálni

```
... (float32_t)(f64a + f64b)              /* compliant     */
... (float64_t)(f32a + f32b)              /* not compliant */
... (float64_t)f32a                       /* compliant     */
... (float64_t)(s32a / s32b)              /* not compliant */
... (float64_t)(s32a > s32b)              /* not compliant */
... (float64_t)s32a / (float32_t)s32b     /* compliant     */
... (uint32_t)(u16a + u16b)               /* not compliant */
... (uint32_t)u16a + u16b                 /* compliant     */
... (uint32_t)u16a + (uint32_t)u16b       /* compliant     */
... (int16_t)(s32a - 12345)               /* compliant     */
... (uint8_t)(u16a * u16b)                /* compliant     */
... (uint16_t)(u8a * u8b)                 /* not compliant */
... (int16_t)(s32a * s32b)                /* compliant     */
... (int32_t)(s16a * s16b)                /* not compliant */
... (uint16_t)(f64a + f64b)               /* not compliant */
... (float32_t)(u16a + u16b)              /* not compliant */
```

Egyéb esetekben egy segédváltozót kell bevezetni, és így már engedélyezett a konverzió.

Példa:

```
uint16_t u16a = 40000;
uint16_t u16b = 30000;
uint32_t u32x;

U32x = u16a + u16b; /* u32x = 70000 vagy 4464? */
```

Az összeadás aritmetikáját nem az eredmény tárolási típusa, hanem a compiler belső aritmetikája szabályozza (belső int típus), ezért, ha a belső aritmetika 16 bites akkor az összeadásban túlcsordulás következhet be. Ilyen jellegű hibalehetőségből van még jó pár darab.

- Amennyiben biteltoló műveletet használunk 8, vagy 16 bites változókon, akkor rögtön a művelet előtt *cast*-olni kell vissza ezekre a típusokra.

```
uint8_t port = 0x5aU;
uint8_t result_8;
uint16_t result_16;
uint16_t mode;

Result_8 = (~port) >> 4; /* Nem engedélyezett */
```

A köztes aritmetika szóhossz miatt tartalmazhat nagyobb helyértékű biteket is a művelet A példában a ~port lehet 0xff5a, illetve 0xffffff5a attól függően, hogy 16, vagy 32 biten hajtották e végre

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

Pointer type conversion (pointer tipus konverziók)

- Pointert minden esetben explicit módon kell cast-olni, kivéve, ha a céltípus void, vagy teljesen megegyező a forrás típussal.
- A void típusok automatikusan konvertálódnak más pointer típusokká.
- Függvény pointereket nem szabad konvertálni másfajta függvénypointerekké.
- Pointert integerré cast-olni nem szabad, mert a pointer mérete architektúra függő.
- Olyan cast-olást nem lehet végrehajtani, ami elhagyja a *constant*, vagy *volatile* jelzőt.

Javasat továbbá, hogy két különböző típusokra mutató pointert nem célszerű *cast*-olni, mert *aligment* problémához vezethet (Példa erre, ha valaki kipróbálja a Vector CCP protokollcsomagot egy ARM7 processzoron).

Expressions (kifejezésekre vonatkozó szabályok)

- A precedencia sorrendre csak a legkisebb mértékben szabad hagyatkozni, de felesleges zárójelezést nem célszerű használni.

```
x = a * -1;                    /* acceptable      */
x = a * (-1);                  /* () not required */
```

- A kifejezés értékének az utasítások szabvány által engedett kiértékelési sorrendjétől függetlennek kell lennie.

```
uint16_t a = 10;
uint16_t b = 65535;
uint32_t c = 0;
uint32_t d;

d = (a + b) + c;    /* d is 9; a + b wraps modulo 65536 */
d = a + (b + c);    /* d is 65545                       */
/* this example also deviates from several other rules  */
```

```
x = b[i] + i++;          =>    x = b[i] + i;
                               i++;

x = func( i++, i );

x = f(a) + g(a);         =>    x = f(a);
                               x += g(a);
```

A függvények módosíthatnak azonos globális változókat

- A &&, || logikai kifejezések jobb oldalán nem lehet „side effect-et” okozó kifejezés (a műveletet befolyásoló).
- A &&, || logikai kifejezéseknek elsődleges kifejezéseknek kell lenniük. Tehát zárójelezni kell őket.
- Bitmanipuláló műveletet nem szabad signed, vagy floating típusokon végrehajtani (>>, <<, ~, &, ^)
- Shift-elő operátor használatakor a bal oldalon szereplő változónak legalább 1 bitnyire le kell fednie a jobb oldali kifejezés értéktartományát.
- Az inkrementáló és dekrementáló utasításokat nem szabad más utasításokkal együtt használni.

Controll Statement expressions (Vezérlési szerkezetek szabályai)

- A nullával való egyenlőséget vizsgálni kell, hacsak a kifejezés nem boolean típusú.

```
if ( x != 0 )     /* Correct way of testing x is non-zero */
if ( y )          /* Not compliant, unless y is effectively Boolean data
                        (e.g. a flag) */
```

- Lebegőpontos típust nem lehet egyenlőségre, vagy nem egyenlőségre tesztelni, mert annak eredménye architektúra és compiler implementáció függő. A *for* ciklus kifejezései sem tartalmazhatnak lebegőpontos típust.

```
float32_t x, y;
/* some calculations in here       */
if ( x == y )     /* not compliant */
     { /* ... */ }
if ( x == 0.0f)   /* not compliant */
```

- Ciklusváltozót nem lehet a ciklus belsejében módosítani.

```
flag = 1;
for ( i = 0; (i < 5) && (flag == 1); i++ )
{
   /* ... */
   flag = 0;   /* Compliant - allows early termination of loop */
   i = i + 3;  /* Not compliant - altering the loop counter    */
}
```

Controll Flow (Vezérlési folyam szabályai)

- A program nem tartalmazhat elérhetetlen kódot.

```
switch (event)
{
case E_wakeup:
   do_wakeup();
      break;       /* unconditional control transfer    */
      do_more();   /* Not compliant - unreachable code */
   /* ... */
default:
   /* ... */
   break;
}
```

- Minden programsornak rendelkeznie kell valamilyen hatással.

```
x >= 3u;                /* not compliant: x is compared to 3,
                           and the answer is discarded */
```

- A goto használata tiltott!
- A continue használata tiltott!
- Az iterációknak egyetlen break kiszálló pontja legyen.
- Egy for, while, case szerkezet mindenképpen { … } kell hogy használjon.

```
for (i = 0; i < N_ELEMENTS; ++i)
{
    buffer[i] = 0;        /* Even a single statement must be in braces */
}

while ( new_data_available )
    process_data ();      /* Incorrectly not enclosed in braces */
    service_watchdog ();  /* Added later but, despite the appearance
                             (from the indent) it is actually not
                             part of the body of the while statement,
                             and is executed only after the loop has
                             terminated */
```

- Egy *if* szerkezet mindkét ágát { … } kell tenni, ez alól kivétel, ha az *else* ágat rögtön egy másik *if* követi.

- Az *else – if* sorozat mindenképpen *else* ággal kell záródjon.

```
if ( test1 )
{
    x = 1;                /* Even a single statement must be in braces */
}
else if ( test2 )         /* No need for braces in else if             */
{
    x = 0;                /* Single statement must be in braces        */
}
else
   x = 3;     /* This was (incorrectly) not enclosed in braces */
   y = 2;     /* This line was added later but, despite the appearance
              (from the indent) it is actually not part of the else,
              and is executed unconditionally */
```

Functions (Függvények)

- Egy függvény nem hívhatja meg saját magát. Tehát nincs iteráció.
- A prototípus deklarációban az összes paraméterhez változó nevet is kell rendelni nem elég a típus. A paraméterekkel nem rendelkező függvényeknél a *void* kulcsszót kell használni.
- A deklarációnak és a definíciónak meg kell egyeznie.
- A paraméterlistában szereplő pointereket *constant*-oknak kell definiálni, ha a pointert nem használjuk arra, hogy megváltoztassuk a változó tartalmát.

```
void myfunc( int16_t * param1, const int16_t * param2, int16_t * param3)
/* param1: Addresses an object which is modified - no const
   param2: Addresses an object which is not modified - const required
   param3: Addresses an object which is not modified - const missing */
{
   *param1 = *param2 + *param3;
   return;
}
/* data at address param3 has not been changed, but this is not const
therefore not compliant */
```

- Nem *void* függvényeknek explicit módon megadott *return* kulcsszóval kell rendelkezniük.
- Ha egy függvény visszatérési értéke hibakód, akkor azt ellenőrizni kell.
- A függvény azonosító vagy ()-hívásként, vagy &-al direkt pointer jelöléssel használható, önmagában nem.

```
if (f)     /* not compliant - gives a constant non-zero value which is
{             the address of f - use either f() or &f                  */
   /* ... */
}
```

Pointers and arrays (Pointerek és tömbök)

- Pointer aritmetika csak tömbelemekre mutató pointereknél használható
- Az összehasonlító jeleket (<,>,==) csak azonos tömbökre mutató pointerek esetében lehet alkalmazni.

```
void my_fn(uint8_t * p1, uint8_t p2[])
{
   uint8_t index = 0;
   uint8_t * p3;
   uint8_t * p4;

   *p1 = 0;
   p1 ++;        /* not compliant - pointer increment               */
   p1 = p1 + 5; /* not compliant - pointer increment                */
   p1[5] = 0;    /* not compliant - p1 was not declared as an array */
   p3 = &p1[5]; /* not compliant - p1 was not declared as an array */
   p2[0] = 0;
   index ++;
   index = index + 5;
   p2[index] = 0; /* compliant      */
   p4 = &p2[5];   /* compliant      */
}
```

- 2-nél magasabb rendű pointer indirekció nem engedett.

```
struct s *   ps1; /* compliant      */
struct s **  ps2; /* compliant      */
struct s *** ps3; /* not compliant  */
```

Structures and Unions

- Memóriaterületet nem lehet újrahasználni, az Unio típusok használatát kerülni kell.
  - *Bit sorrend, endianness, padding és még egyéb hibák forrása lehet*

Preprocessing directives

- Az *#include* direktívát, csak más direktívák, vagy kommentek előzhetik meg.
- Az *#include* direktívát vagy “filename”, vagy <filename> követheti más nem.
- A *#define* ban szereplő nem konstans értékeket figyelmesen kell zárójelezni.
- Az *#undef* makró használata tilos. *#define*-t csak blokkon kívül lehet használni.
- Minden preprocessor direktívált definiálni kell használat előtt, tehát a *#define*-nak meg kell előznie az *#ifdef*-et (Ezalól a header file-ok includolása lehet nek kivétel). Az *#ifdef, #else* részeknek egy file-on belül kell végződniük.

- Headerfile-ok kétszeri inklúdálását a következő módon kell elkerülni:

```
#ifndef AHDR_H
#define AHDR_H
/* The following lines will be excluded by the
   preprocessor if the file is included more
   than once */

...

#endif
```

Standard libraries

- A Standard library-k makróit, lefoglalt azonosítóit nem szabad újradefiniálni.

- A library-knak átadott paraméterek értéktartományát ellenőrizni kell (a library nem biztos, hogy megteszi ezt.)

- Dinamikus memória allokációt nem szabad használni (*calloc, malloc, free*).

- Az *stdio.h* könyvtár nem használható a végtermékben.
- Implementáció függő végrehajtás
- A *time.h* könyvtár függvényei nem használhatóak (Implementáció függő, nem specifikált formátum)

Runtime failures

Run-time hibának nevezünk minden itt felsorolt a program futása közben bekövetkező hibát:

- Aritmetikai hibák: overflow, underflow, divide by zero…
- Pointer aritmetikai hibák
- Tömbtúlindexelések
- Függvény paraméterek

- 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

5. Irodalomjegyzék

[1] Standish Group. Honlap: http://www.standishgroup.com/
[2] The Bosch Yellow Jackets. *„Electronic Transmission Control ETC”* Edition 2004
[3] S. Rathmann and R. Fischerkeller *„Latest Trends In Automotive Electronic Systems - Highway Meets Off-Highway?”* Bosch Engineering GmbH. 2007
[4] Model Checking of Software, Stefan Leue 2005
[5] Dr. Balla Katalin, Minőségmenedzsment a szoftverfejlesztésben, ISBN: 978 963 545 473
[6] Carnegie Mellon University Software Engineering Institute *„CMMI® for Development, Version 1.2”* CMU/SEI-2006-TR-008, ESC-TR-2006-008. 2006.
[2.1] Openproj
[2.2] Gantt
[2.3] COCOMO
[2.4] DOORS
[3.1] Vízesés Modell
[3.2] Spiral Modell