REPÜLÉSTUDOMÁNYI KÖZLEMÉNYEK
36. évfolyam (2024) 1. szám 35–45. • DOI: 10.32560/rk.2024.1.3

Schuster György

A biztonságkritikus fejlesztésben alkalmazott módszertanok

Amióta a számítógépeket irányítástechnikai feladatok elvégzésére használjuk, az a célunk, hogy mind hardveres, mind szoftveres szempontból a működésbiztonságuk a kívánt mértékeknek megfelelő legyen. Azt könnyen beláthatjuk, hogy egy nehezen meghatározható szint felett a szoftverek bonyolultsága már annyira magas, hogy azokat lehetetlen átlátni és követni, és mint azt több előző cikkben is megírtuk, kellő mértékben tesztelni. Ebben az írásban rövid történeti összefoglalás után a manapság használt fejlesztési módszertanokkal foglalkozunk, kiemeljük célszerű alkalmazási területüket és főbb jellemzőiket. Mintapéldákat mutatunk be, hogy az egyes tárgyalt módszertanokat hol és hogyan alkalmazhatjuk a repülésben.

Kulcsszavak: *működésbiztonság, szoftverfejlesztés*

1. Bevezetés

Az elektronikus számítógépek irányítástechnikai feladatokra való felhasználása azonnal nyilvánvalóvá tette, hogy a kezdetleges programozási eljárások nem megfelelők. Az 1940-es évek végén az ENIAC elektronikus számítógépet még a klasszikus analóg számítógépekhez hasonló plug-board és huzalozás segítségével programozták [11]. Ez a módszer nagyon körülményes, nagyon nehézkes, nehezen tesztelhető és egyáltalán nem hatékony. Ezért a szakemberek más megoldást kerestek, programozási nyelveket alkottak, és elkezdték a számítógépeket használni a számítógépek programozására. Azonban ebben az időben arra is rádöbbentek, hogy a bonyolultsági szint annyira megnőtt, hogy egy-egy hiba gyökerét nagyon nehéz megtalálni. Ekkor történt annak a manapság furcsának tűnő megoldásnak a használata, hogy a nagy megbízhatósági igényeket kielégítő programokat kézzel fordították le sorról sorra. Természetesen ezek a programok assembly szinten íródtak. Számos üreszköz rendszerprogramja készült így. Ez másik oldalról egészen jól illeszkedett az akkori hardvertechnológiákhoz, lásd ferritmagos memória.

Az alkalmazott módszertan ebben az időben az úgynevezett programozás kipróbálás „modell" volt. Ez azt jelenti, hogy megírjuk a programot, majd teszteljük, és a felmerülő hibákat javítjuk. Ezt addig csináljuk, amíg a program megfelelően nem működik. Ez kicsi és egyszerű programok esetén működik, de ha a program nagy, akkor a sok javítás miatt követhetetlen lesz, és minden egyes javítás újabb hiba lehetőségét rejti magában. Minél nagyobb a program

és minél több funkciót lát el, annál nagyobb a kockázat. Ez a magasabb szintű programozási nyelveknél is alkalmazott módszer volt.

Természetesen a magas szintű programozási nyelvek esetén a kézi fordítás már nem volt járható út.

Az 1960-as évek végére nyilvánvalóvá vált az a tény, hogy így már nem lehet továbblépni. Ezt felismerve több kutató, E. W. Dijkstra, C. A. R. Hoare, Niklaus Wirth, megfogalmazta a strukturált programozás alapötletét. Eszerint: „A program nem más, mint absztrakt algoritmusok megfogalmazása meghatározott adatszerkezeteken", vagy Niklaus Wirth megfogalmazásában:

Algoritmusok + Adatstruktúrák = Program [13].

A tervezés folyamán felülről lefelé, a feladatból a részletek felé tervezünk és fordítva hozzuk létre a kész programot, vagyis alulról felfelé építkezünk. A program megírása során háromféle elemet használunk, ezek [13]:

- szekvencia, vagyis műveleteket hajtunk végre, eljárásokat hívunk, értékeket adunk;
- szelekció, vagyis különböző feltételeknek megfelelően elágazásokat hozunk létre;
- iteráció, ciklusokat használunk.

A klasszikus strukturált programozás tökéletesen alkalmas algoritmusok tiszta és világos leírására. Viszont korlátozottan alkalmas igen összetett műveletek elvégzésére.

Az 1960-as évek közepén az AT&T és a Bell Laboratórium közreműködésével egy MULTICS[1] nevű operációs rendszert fejlesztettek. Ebből a fejlesztői csapatból vált ki Ken Thompson, aki 1969 szeptemberére kifejlesztette a UNICS[2] nevű operációs rendszert. Ez természetesen assembly nyelven íródott [13].

Az operációs rendszer annyira jól sikerült, hogy az eredeti PDP-7-es számítógép mellett más számítógépekre is implementálták. Természetesen assembly nyelvet használva ez igen nehéz volt [13].

Ennek következménye az lett, hogy egy magas szintű programozási nyelvet próbáltak keresni a hordozhatóság biztosítására, de nem találtak. Ha nem találtak, akkor készítettek egyet, ez volt a C programozási nyelv, amely Ken Thompson mellett Dennis Ritchie nevéhez is fűződik [13].

A C egy klasszikus moduláris programozási nyelv. A moduláris nyelvek a strukturált programozás minden tulajdonságával rendelkeznek, de bevezették a modulokat. A modulok önálló forrásállományok. Logikailag a modulok egy adott feladatot látnak el, de ez a gyakorlatban nem mindig teljesül.

A modulok lehetővé teszik, a kód- és adatrejtést. Továbbá lehetővé teszik a kód újrafelhasználását és a részenkénti (modulonkénti vagy modulcsoportonkénti) tesztelésének lehetőségét, ami igen hatékonnyá teszi a fejlesztési folyamatot.

Ez a technológiai lépés nagy előnyökkel járt a klasszikus strukturált programozási módszertannal szemben. Az egyik ilyen előny, hogy több fejlesztő dolgozhat egy projekten. Csak gondoljuk meg azt a tényt, hogy egy Linux kernel több millió kódsorból áll.

Az 1960-es évek közepén felmerült az objektumorientált programozás ötlete. Ezt először Simula nevű programozási nyelven alkalmazták az 1960-as években. A következő lépés a Smalltalk programozási nyelv volt, amelyet a Xerox PARC kutatóintézet fejlesztett

[1] Multiplexed Operating and Computing System.
[2] Uniplexed Operating and Computing System.

ki az 1970-es évek közepén. Az igazi áttörés az 1980-as időszakban történt meg, amikor olyan nyelvek jelentek meg, mint a C++ és az Object Pascal. A későbbi OOP nyelvek tipikus képviselői a Java és a C# [2].

Az objektumorientált nyelvek három újabb tulajdonsággal bővítették a lehetőségeket, ezek [2]:

- öröklődés, ahol a tulajdonságok és szolgáltatások (metódusok) öröklődhetnek;
- egységbezárás, ahol az összetartozó adatok és metódusok egymáshoz vannak rendelve;
- polimorfizmus (más néven többalakúság) ahol a metódusokat statikusan vagy dinamikusan újra tudjuk definiálni.

Az objektumorientált módszertan nagyban növeli az absztrakció szintjét. Erre a megjegyzésre még visszatérünk.

Minden egymást követő módszertan magában foglalja az előző módszertan tulajdonságait, csak ezeket célszerű módon bővíti, esetleg korlátozza.

Ebben a cikkben csak olyan programozási nyelvekkel foglalkozunk, amelyek az úgynevezett imperatív programozási paradigmát követik [12].

Az imperatív programozás alapvető jellemzői a következők:

- állapotalapú programozás: a változók értékei a program futása során változhatnak. Az utasítások arra irányulnak, hogy változtassák meg ezeket az értékeket;
- változók: az állapotok leírását a változók biztosítják;
- utasítások: az imperatív nyelvek kifejezetten utasításokat használnak a végrehajtandó lépések leírására. Például az értékadás, a ciklusok, az elágazások és más vezérlési szerkezetek közvetlenül az állapotot módosítják;
- procedurális programozás: az imperatív paradigma gyakran egybefonódik a procedurális programozással, ahol a programok eljárásokból (eljárásokból, függvényekből) épülnek fel, és ezek egymásba ágyazhatók.

Az utasítások sorrendje fontos, azokat általában az adott sorrendben kell végrehajtani. Ezt nevezzük szekvenciának.

Megjegyzés: a másik paradigma a funkcionális programozás. A funkcionális programozás és működésbiztos szoftverek kérdését egy következő cikkben részletezzük.

2. A feladat és a módszertan kapcsolata

Kezdjük ezt a fejezetet a következő kijelentéssel:

Biztonságkritikus szoftver fejlesztésében nincs szükség sem „igazi programozóra”, sem divatra.

A biztonságkritikus fogalom esetünkben a működésbiztonságot takarja. Itt a hatékonyság, az áttekinthetőség, a megbízhatóság és a tesztelhetőség a legfontosabb. Hiába szép egy olyan algoritmus, amely tele van trükkökkel, rekurziókkal, önátíró kódokkal vagy többszörös erőltetett öröklődéssel, indokolatlan polimorfizmussal és végtelenségig fokozott absztrakciókkal, de nem teljesít valamilyen előírást.

A divat veszélye. Saját emlékem, amikor 1992 végén megkerestek egy viszonylag egyszerű felügyeleti szoftver megírásának feladatával. A program hardvereszközt is kezelt volna. A feladat kiválóan alkalmas volt DOS 3.22 operációs rendszerre minden szempontból. A megrendelő ragaszkodott a Windows 3.1-hez, mert idézem: „Az korszerű…". A Windows 3.1-et 1992. április 6-án bocsátották ki, megbízhatósága katasztrofális volt, és nagyon gyengén volt dokumentálva. A real-time követelményekről inkább ne beszéljünk.

Ezt a trendet napjainkban is tapasztaljuk, természetesen nem a DOS 3.22, Windows 3.1 kontextusában, inkább az alkalmazott módszertanok szempontjából.

Emlékezzünk arra, amikor Bjarne Stroustrup 1980-ra kifejlesztette a C++ programozási nyelvet, szinte minden rendszert C++-ban, objektumorientáltan fejlesztettek. Ennek az lett a következménye, hogy nagy és lassú rendszerek készültek. Vizsgálataink alapján ezeknek a rendszereknek a 75%-ánál nem volt indokolt az objektumorientált fejlesztés [8].

A cikk e pontján ha valaki azt érzi, hogy az objektumorientált fejlesztésnek ellensége lennék, az téved. Ez a cikk további részében ki fog derülni.

A kérdés továbbra is az, hogy mit, hol és miért.

2.1. A moduláris fejlesztés javasolt alkalmazási területei

Irányítási kérdésekben számos esetben előfordul, hogy az adott irányító berendezés egy, vagy néhány egymástól különböző feladatot lát el. Ez az a tipikus eset, amikor az objektumorientált fejlesztés nem indokolt.

Az adott feladatokat egyenként különböző programrészletek valósítják meg. Ezek az egységek külön-külön megírhatók, tesztelhetők és igény szerint újra felhasználhatók. Nem indokolt a magas absztrakciós szint, sokkal inkább a megbízható, gyors, megjósolható működés.

Megjegyzés: a programrészletet a szakmai zsargonban unit-nak nevezik. Ez lehet egyetlen függvény, egy modul vagy több hierarchikusan egymásra épülő modul. Lényeges, hogy ez a programrészlet egy adott feladatot vagy feladatcsoportot kezel.

Vegyük példának egy repülőgép oldalkormány-vezérlését. A példa terjedelmi okokból nem teljes, számos biztonsági megoldást és egyéb információs kapcsolatot nem részletezünk [4], [8].

Feladatok:

- a pedálok pozíciójának lekérdezése;
- az oldalkormány sebességtől függő kitérítése;
- csillapításvezérlés;
- erő-visszacsatolás a pedálokra.

Ez most egyetlen ECU[3]-ra van bízva. Kerülendő a kritikát, tisztában vagyok azzal, hogy ebben az esetben hardverredundanciára lenne szükség (A320 esetén ELAC1, ELAC2, FAC1, FAC2) [3], [4].

A feltételezett ECU szoftvere egy monolitikus, úgynevezett kis real-time operációs rendszeren futó szoftver.

A feladatok:

- pedálok pozíciójának lekérdezése;

[3] Electronic Control Unit.

    - bemeneti jelek:
    - pedálpozíció;
  - pedálerőmérés;
    - kimeneti jelek:
    - pedálkitérítés értéke;
- az oldalkormány kitérítése:
  - bemeneti jelek:
    - sebességjel a repülésvezérlő számítógéptől;
    - pedálpozíció-modul kimeneti jele;
    - oldalkormány-pozíció jele;
  - kimeneti jelek:
    - oldalkormány hidraulikus hajtás kimeneti jele;
  - feladatok:
    - a sebességjel alapján a kormánykitérítés számítása;
    - a kormánykitérítés szabályozása;
- csillapításvezérlés:
  - bemeneti jelek:
    - pedálpozíció-modul kimeneti jele;
    - függőleges tengely körüli gyorsulás jele;
    - sebességjel;
    - tömeg és súlyponti adatok;
  - kimeneti jel:
    - oldalkormány-kitérítés módosító jele;
  - feladat:
    - a repülőgép dinamikai modellje és a bemeneti jelek alapján a legyezőmozgás csillapítása;
- erő-visszacsatolás a pedálokra:
  - bemeneti jel:
    - sebesség jel;
  - kimeneti jel:
    - terhelés mértéke;
  - feladat:
    - a pedálterhelés beállítása a sebesség függvényében.

Látható, hogy a feladatok egyediek. Az is látható, hogy a feladatokat megvalósító programrészletek egymás között elég összetett módon kommunikálnak. A kommunikációt az alkalmazott real-time operációs rendszer biztosítja [5].

Vegyük kicsit részletesebben a csillapításvezérlés feladatát.

A repülőgép és a repülési adatok alapján az adott programrészletnek meg kell oldania a kérdéses leíró differenciaegyenleteket és a bemeneti jelek függvényében a legyező irányú lengő mozgás csillapítását.

Ez egy elég összetett numerikus eljárásrendszer, amelyet nyilvánvalóan rendkívül alaposan kell megtervezni és tesztelni. A feladatok jól elkülöníthetők és egyediek, ezért az objektumorientált megoldás nem indokolt.

A fejlesztés egymástól függetlenül történhet, természetesen egy előre lefektetett specifikáció alapján. A statikus tesztelés viszonylag egyszerű, mert az absztrakciós szint alacsony. A dinamikus tesztelés programegységenként szimulált környezetben részletenként elvégezhető. Így a rendszerintegrációs fázisra, amikor a programegységeket egy funkcionális egységgé alakítjuk, a programegységek megbízhatósági szintje magas.

2.2. Az objektumorientált fejlesztés javasolt területei

Az objektumorientált módszertan használata a következő biztonságkritikus területeken indokolt:

- ha több, közel hasonló fizikai objektumot kell kezelnünk úgy, hogy ezeknek a program szempontjából saját belső állapotai vannak (egységbezárás);
- ha több olyan feladatot kell ellátni, amelyek esetén számos közös tulajdonság és eljárás van, de ezeket bizonyos esetekben bővíteni kell (öröklődés);
- ha több olyan feladatot kell ellátni, amelyek esetén számos közös tulajdonság és eljárás van hasonlóan az előző ponthoz, de ezek közül egyet vagy többet bizonyos esetekben módosítani kell (öröklődés és polimorfizmus);
- ha a kérdéses szituáció egyes objektumokat „hoz létre" vagy tüntet el.

Hely hiányában csak a 2. és 4. pontra mutatunk be mintapéldát, ez a TCAS-rendszer kijelzőkezelése. Hasonlóan az előzőekben bemutatott példához ezt is leegyszerűsítettük.

A TCAS[4] olyan eszköz, amely általában a navigációs képernyőn (ND)[5] figyelmezteti a pilótát, ha egy másik repülőgép túl közel került. Ha ez a közelség eléri azt a mértéket, hogy kitérő manőverre van szükség, akkor a másik gép fedélzetén levő hasonló berendezéssel rádiójelekkel egyezteti a helyzetet, és mindkét pilótának javaslatot tesz arra a kitérő manőverre, amely biztosítja az ütközés elkerülését. A manővert a pilóták hajtják végre előírások szerint és esetleg a légi forgalmi irányító utasítása szerint [14].

1. ábra
A TCAS-képernyő [6]

[4] Traffic Collision Avoidance System.
[5] Navigation Display.

Az 1. ábrán látható, hogy a TCAS kijelzése a hatókörön belül szimbólumokkal, színkódokkal és numerikus értékekkel ábrázolja a repülőgép körüli légi járművek helyzetét és státuszát.

Az egyszerűsített modellben minden egyes szimbólum közös tulajdonsága a képernyőn lévő pozíciója, amelyet minden egyes szimbólumra azonos módon számítunk ki. Eltérő tulajdonságok a szimbólum alakja, színe, az emelkedés és süllyedés irányának kijelzése és a magasságkülönbség numerikus értéke és ennek színe.

A légi forgalmi szituáció az idő és a repülőgépek függvényében változik. Egyes gépek belépnek a rendszer hatókörébe, egyes gépek elhagyják azt.

Ezt a helyzetet objektumorientáltan az úgynevezett dinamikus objektumokkal lényegesen egyszerűbben lehet kezelni, mint moduláris esetben.

Megjegyzés: a dinamikus objektumok kezelése biztonságkritikus esetekben nem célszerű, illetőleg magasabb biztonsági (SIL)[6] szinteken nem is engedélyezett. Ez az eset egy kijelzés, amelyet szóbeli utasítás is követ. A TCAS csak a pilótákon keresztül gyakorol hatást a légi jármű viselkedésére. Ezek a plusz információk megfelelő redundanciának számítanak, így a dinamikus objektumok használata megengedett.

A képernyőn kijelzendő TCAS-szimbólumok [8]:

Saját repülőgép. Gyakran más szimbólumot használnak. Színe lehet fehér, türkiz és sárga. A képernyőn nem változik.

Ismeretlen forgalom. Magassága és függőleges sebessége ismeretlen. Színe fehér vagy türkiz. A későbbiekben változhat a pozíciója és típusa.

Közeli forgalom. Nincs sem TA-[7], sem RA-[8] tartományban, lásd 1. ábra. Színe fehér vagy türkiz. A későbbiekben változhat a pozíciója és típusa. Jelenleg 1100 lábbal magasabban van, és süllyed.

Figyelmet igénylő forgalom. TA-tartományban van, de csak figyelmeztet. Színe sárga, formája kör. A későbbiekben változhat a pozíciója és típusa. Jelenleg 900 lábbal alattunk van, és szintben repül.

Veszélyes forgalom. RA-tartományban van, ekkor már a TCAS-utasításokat ad a pilótáknak. Színe piros, formája négyzet. A későbbiekben változhat a pozíciója és típusa. Jelenleg 500 lábbal alattunk van, és emelkedik.

A jobb megértés érdekében rövid összefoglalót adunk a TA- és RA-tartományokról.

TA-tartomány: potenciális ütközésveszély esetén hangjelzést ad ki. Ez a rendszer minden „behatoló" repülőgépről figyelmezteti a pilótát „traffic, traffic" hangos bejelentéssel. Nem ad semmilyen javaslatot elkerülési manőverre [6], [10].

6 Safety Integration Level.
7 Traffic Advisory.
8 Resolution Advisory.

RA-tartomány: a konfliktushelyzet súlyosabbá válik a TA-riasztást követően, és konkrét ütközésveszély áll fenn, szöveges utasítást és vizuális figyelmeztetést generál. Ez a riasztás jelzi az érintett repülőgépet, és a pilóta által azonnal végrehajtandó elkerülő műveletet javasol [9].

A rendszer úgy van kialakítva, hogy a másik repülőgép TCAS-rendszere ellentétes jellegű műveletet javasol [10].

Amint a veszélyes helyzet megszűnik, a rendszer „Clear Conflict" üzenetet küld [10].

Térjünk vissza a szoftverfejlesztés kérdéséhez. A saját repülőgép szimbólum a TCAS bekapcsolásával kirajzolódik a képernyőre, és egészen addig ott marad, amíg a TCAS-rendszer vagy a TCAS-kijelzést ki nem kapcsolják. Ezzel így nem foglalkozunk tovább.

A további objektumok közös tulajdonsága, hogy van nekik a képernyőn vízszintes és függőleges koordinátájuk. Ezeket célszerű egy úgynevezett ős osztályba deklarálni.

Adott továbbá négy alapvető képernyőszimbólumunk, amelyek mindegyikét egy-egy önálló osztályba definiálják.

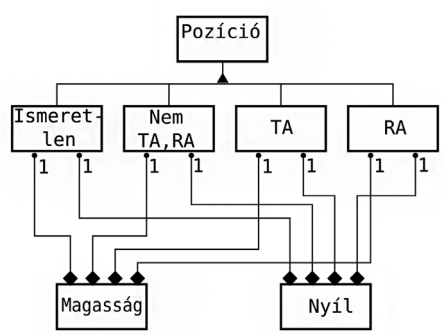
Ezekhez a kijelzésekhez tartozik a magassági információ és a függőleges mozgás szimbóluma. Ezeket szintén érdemes egy-egy osztályba definiálni.

Megjegyzés: a fenti bekezdésekben többször használtuk az osztály kifejezést. Az osztály egy objektumcsoport formális leírása [8].

Az objektum az osztály fizikai megvalósítása. Ez már működő kódokat és változó területeket tartalmaz [8].

A képernyőszimbólumok objektumai a magassági információt és a függőleges mozgást kijelző objektumait tartalmazza. Ez azt jelenti, hogy amikor egy képernyőszimbólum objektuma létrejön, ezek az objektumok is létrejönnek [7].

Az UML-diagram a 2. ábrán látható.



2. ábra
A TCAS-kijelzés UML-diagramja [6]

A következő probléma, hogy a tervezésnél nem tudhatjuk, hogy egy kérdéses helyzetben hány darab objektumot kell elhelyezni a képernyőn. Ez azt feltételezi, hogy a szimbólumokat dinamikusan kell létrehozni.

Tehát, ha egy légi jármű a TCAS hatókörébe kerül, akkor az ennek megfelelő szimbólumnak meg kell jelennie a képernyőn, és ezt kezelnünk is kell. Természetesen, ha kikerül a megfigyelési tartományból, akkor a szimbólumnak el kell tűnnie a képernyőről és a memóriából.

Továbbá, ha az adott légi jármű státusza változik, például közeli állapotból belép TA-állapotba, akkor a kijelzésnek változnia kell. Erre az egyik megoldás, hogy a dinamikus közeli objektumot megszüntetjük, de azonnal létrehozzuk a TA-típusú objektumot.

Ez számos feladatot jelent a fejlesztő számára, ami bizonyos szempontból kockázatos lehet. Természetesen ez a leegyszerűsített példa egy gyakorlott programozónak egyszerű feladat.

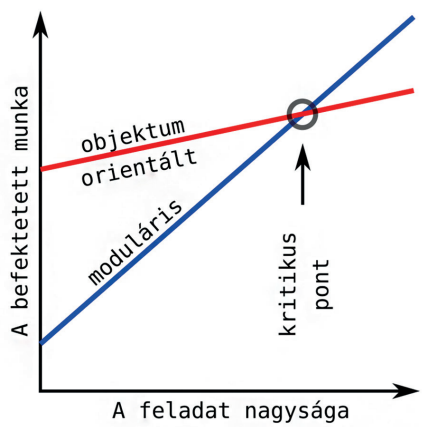
2.3. Egyéb szempontok

Tapasztalt tervezőként, programozóként, tesztelőként és auditorként az a tapasztalatom, hogy a két tárgyalt módszertan alkalmazása között jelentős különbség van a befektetett munka mennyisége és a feladat mérete között (lásd 3. ábra).

A diagram alapján láthatjuk, hogy egy feladat megoldásánál a feladat méretétől függően az elvégzett munka mennyisége változó.

Általánosan kijelenthetjük, hogy a moduláris módszertannál viszonylag kevés előkészítő munka után már van értékelhető eredmény. Ez tesztelési szempontból nagyon kedvező. Azonban ahogy nő a feladat, meredeken nő a befektetett munka.

Objektumorientált módszertan esetén kezdetben nagyon sok munkát kell elvégezni, de a feladat növekedésével a befektetett munka jóval laposabb függvényt eredményez.



3. ábra
A moduláris és az objektumorientált módszertanok munkadiagramja [6]

A 3. ábrán látható kritikus pont helyzetét sajnos egzakt módon nem tudjuk megadni, ezt mindig az adott probléma határozza meg.

Ha a két bemutatott példát vizsgáljuk, az oldalkormány-vezérlés esetén indokolatlan lenne az objektumorientált megoldás, mert minden feladatból csak egy darab van. A TCAS-kijelzésnél előre nem tudjuk, hány szimbólumot kell kezelnünk, lehet, egyetlenegyet sem, de lehet, hogy 30 darabot. Ez megoldható modulárisan, de nagyon nehézkes, míg objektum-orientáltan viszonylag egyszerű.

3. Összefoglalás

A két módszertan egymás mellett létezik. Mindkettőnek megvan a maga szerepe. Nagyon fontos, hogy a tervezés folyamán figyelembe vegyük a felhasználás módját, de ne feledkezzünk el a fejlesztési és tesztelési szempontokról sem.

Tapasztalati tény, hogy az objektumorientált fordítóprogramok által előállított kódok nagyobbak és lassabban futnak, mint a „hagyományos" fordítóprogramok kimeneti kódjai. Ez szintén szempont lehet a tervezésnél, de manapság már inkább a kifogás tartományába eső probléma, hiszen a hardverek a megfelelő szempontok figyelembevételével széles skálán rendelkezésre állnak mind memóriakapacitás, mind futási sebesség tekintetében.

Erre szokták azt mondani: „Na de az ár!" Igazából nincs jelentős különbség, sőt néha a nagyobb teljesítményű eszköz az olcsóbb.

Azt is meg kell jegyeznünk, hogy számos objektumorientált nyelv, mint például a C++, vagy az ADA [1] lehetővé teszi, hogy modulárisan, objektumokon kívül programozhassunk. Viszont a C++ fordító összehasonlítva a C fordítóval még akkor is sokkal szigorúbb a programozóval szemben, ha a C összes szigorító kapcsolóját bekapcsoljuk [7].

Ezt a MISRA-ajánlás is megemlíti [9].

Felhasznált irodalom

[1] Adacore, *Introduction to Ada.* [é. n.]. Online: https://learn.adacore.com/courses/intro-to-ada/chapters/introduction.html

[2] Adacore, *Object-oriented programming.* [é. n.]. Online: https://learn.adacore.com/courses/intro-to-ada/chapters/object_oriented_programming.html

[3] Airbus Training Flight Crew Training Manual, FLIGHT CONTROLS. Online: www.smartcockpit.com/docs/A320-Flight_Controls.pdf

[4] AviationHunt Team, „ATA 27: Airbus A320 (Technical Notes)," *Aviationhunt.com,* 2024. március 12. Online: www.aviationhunt.com/airbus-a320-ata-27/#google_vignette

[5] D. Briere, C. Favre, P. Traverse, „Electrical Flight Controls, From Airbus A320/330/340 to Future Military Transport Aircraft: A Family of Fault-Tolerant Systems," in *The Avionics Handbook,* C. Spitzer szerk., Boca Raton, CRC Press LLC, 2001. Online: https://helitavia.com/avionics/TheAvionicsHandbook_Cap_12.pdf

[6] IVAO Documentation Library, *Traffic Collision Avoidance System – TCAS.* [é. n.]. Online: https://wiki.ivao.aero/en/home/training/documentation/Traffic_collision_avoidance_system-TCAS

[7] L. Erdődi, A. Jøsang, „Exploitation vs. Prevention: The Ongoing Saga of Software Vulnerabilities," *Acta Polytechnica Hungarica,* 17. évf. 7. sz. pp. 199–218. 2020. Online: https://doi.org/10.12700/APH.17.7.2020.7.11

[8] M. Olsson, *C++20 Quick Syntax Reference A Pocket Guide to the Language, APIs, and Library.* Fourth Edition, New York, Apress, 2020. Online: https://doi.org/10.1007/978-1-4842-5995-5

[9] S. Misra, „Evaluation Criteria for Object-oriented Metrics," *Acta Polytechnica Hungarica,* 8. évf. 5. sz. 2011. pp. 109–136. Online: http://acta.uni-obuda.hu/Misra_31.pdf

[10] U.S. Department of Transportation, Federal Aviation Administration, *Advisory Circular.* 2014. Online: www.faa.gov/documentlibrary/media/advisory_circular/ac_20-151b.pdf

[11] U.S. Department of Transportation, Federal Aviation Adminstration, *Introduction to TCAS II Version 7.1.* 2011. Online: www.faa.gov/documentlibrary/media/advisory_circular/tcas%20ii%20v7.1%20intro%20booklet.pdf
[12] Wikipedia, *ENIAC.* [é. n.]. Online: https://hu.wikipedia.org/wiki/ENIAC
[13] Wikipedia, *Procedural Programming.* [é. n.]. Online: https://en.wikipedia.org/wiki/Procedural_programming
[14] Wikipedia, *Structured Programming.* [é. n.]. Online: https://en.wikipedia.org/wiki/Structured_programming
[15] Wikipedia, *Traffic Collision Avoidance System.* [é. n.]. Online: https://en.wikipedia.org/wiki/Traffic_collision_avoidance_system
[16] Meleg Á. G., *Az igazi programozó.* 2015. október 17. Online: www.scribd.com/doc/285592411/Az-Igazi-Programozo

Methodologies Used in Safety-Critical Development

Ever since we use computers to perform control engineering tasks, our goal has been to ensure that their operational safety meets the desired standards, both from a hardware and software perspective. We can easily see that above a level that is difficult to define that the complexity of the software is so high that it is impossible to see and follow it and, as we wrote in several previous articles, to test it sufficiently. In this paper, after a brief historical summary, we will deal with the development methodologies used today, highlight their appropriate field of application and their main characteristics. We present examples of where and how each discussed methodology can be applied in aviation.

Keywords: *operational security, software development*

Dr. Schuster György tanszékvezető, egyetemi docens Óbudai Egyetem Kandó Kálmán Villamosmérnöki Kar Elektronikai és Kommunikációs Rendszerek Intézet Műszertechnikai és Automatizálási Tanszék schuster.gyorgy@uni-obuda.hu orcid.org/0000-0002-8573-3670	György Schuster, PhD Head of Department, Associate Professor Óbuda University Kandó Kálmán Faculty of Electrical Engineering Institute of Electronic and Communication Systems Department of Instrumentation and Automation schuster.gyorgy@uni-obuda.hu orcid.org/0000-0002-8573-3670