

# 20 KRITIKUS RENDSZEREK FEJLESZTÉSE

## TÉMAKÖRÖK

Ennek a fejezetnek a célja a kritikus rendszerek fejlesztésénél használt implementációs technikák bevezetése. A fejezetet elolvasva:

- megértjük, hogy a megbízható rendszerek fejlesztésébe miként épül be a hibaelkerülés és a hibatűrés;
- megismerjük a megbízható szoftverfolyamatok jellemzőit és a tevékenységeit;
- megismerjük a hibaelkerülő programozás technikáit;
- megértjük a hibatűrés implementációs lépéseit, valamint azt, hogy a diverzitás és a redundancia miként használható a hibatűrő architektúráknál.

## TARTALOM

- 20.1. Megbízható folyamatok
- 20.2. Megbízható programozás
- 20.3. Hibatűrés
- 20.4. Hibatűrő architektúrák

A fejlettebb szoftvertervezési technikák, a jobb programozási nyelvek és a jobb minőségbiztosítás jelentős javuláshoz vezetett a legtöbb szoftver üzembiztonságát illetően. Kritikus rendszereknél, mint például felügyelet nélküli gépezetek, telefonközpontok vagy repülőgépmotorok azonban további intézkedés szükséges az üzembiztonság magas szintjének eléréséhez. Ezekben az esetekben speciális programozási technikák alkalmazhatók abból a célból, hogy a rendszer biztonságos, védett és megbízható legyen.

Három – egymást kiegészítő – megközelítés létezik üzembiztos szoftver kidolgozására:

1. *Hiba elkerülése.* A rendszer tervezése és implementációja folyamán olyan szemléletmódú szoftverfejlesztésre van szükség, amely segít a programozási hibák elkerülésében, és egyúttal minimalizálja a hibák számát is.

2. *Hiba detektálása.* A verifikációs és validációs folyamatok azért alakultak ki, hogy felfedjék és eltávolítsák a hibákat még azelőtt, hogy a program kikerülne a működési környezetébe.
3. *Hibatűrés.* A rendszert úgy kell tervezni, hogy a hibák vagy a rendszer más váratlan viselkedése a végrehajtás alatt olyan mértékben észlelhető és kezelhető legyen, hogy ne vezessen a rendszer hibás működéséhez.

Ez a fejezet azokra a folyamatokra és technikákra összpontosít, amelyek támogatják a hibaelkerülést és a hibatűrést. A hibadetektálás fontos témakör a maga nemében, így az 5. fejezetben tárgyalom. A hibadetektáláshoz kapcsolódó statikus technikákat a 22. fejezetben, a programtesztelést a 23. fejezetben, a kritikus rendszerek verifikációjához és validációjához kapcsolódó technikákat pedig a 24. fejezetben ismertetem.

Az üzembiztonság eléréséhez bármely rendszer esetén alapvető szerepet játszik a diverzitás és a redundancia. Mindkettő a hibaelkerülésnek mindennapi fedőstratégiája. Ha például az árukereskedelemben invesztálunk, akkor sohasem fektetünk kizárólag egy cégbe, mert ha az tönkremegy, akkor elvesztünk mindent (diverzitás). Az emberek például otthonaikban tartanak tartalékelemeket és -izzókat, hogy hiba esetén azonnal cserélni tudják (redundancia). A számítógépünkéről is rendszeresen mentéseket kellene készítenünk a merevlemez meghibásodásának esetére (redundancia), és az otthonunk biztonsága érdekében is többnyire többféle zárat szerelünk fel (diverzitás).

A kritikus rendszerek tartalmazhatnak olyan komponenseket, amelyek más komponensek funkciójával is rendelkeznek (redundancia), vagy olyan – további – ellenőrző kódot, amely nem feltétlenül szükséges a rendszer működéséhez (redundancia). A hibák így talán még az előtt detektálhatók, hogy meghibásodást okoznának, és a rendszer így talán tovább folytathatja a működését egyes komponensek meghibásodása esetén. Amennyiben a redundáns komponensek nem azonosak a többi komponenssel (diverzitás), akkor egy, a replikált komponensben bekövetkező általános hiba még nem okozza a teljes rendszer meghibásodását.

Azoknál a rendszereknél, ahol a rendelkezésre állás kritikus követelmény, többnyire redundáns szervereket alkalmaznak. Ezek automatikusan működésbe lépnek, ha egy adott szerver meghibásodik. Néha, hogy biztosítsák a rendszer támadások elleni védelmét, a szerverek különböző típusúak és különböző operációs rendszer fut rajtuk. A különböző típusú operációs rendszerek használata csak egy a példák közül a szoftverdiverzitásra és -redundanciára. Más esetekben, amelyekről majd később lesz szó, a diverzitás és a redundancia beépülhet a szoftverbe olyan redundáns szoftverkomponensek alakjában, amelyeket szándékosan különböző technikák segítségével programoztak.

A diverzitás és a redundancia ezenkívül még üzembiztos folyamatok létrehozására is alkalmazható. A tesztelés mellett hibakereső technikaként alkalmazható még a program vizsgálata és statikus elemzése is. Ezek a validációs technikák egymás kiegészítői: amit az egyik nem szűrt ki, azt majd a másik megtalálja. Továbbá ugyanazon tevékenységet (például a program átvizsgálását) a csoport



több tagja is elvégezheti, mert személyiségük, tapasztalatuk és tudásuk alapján különböző módon fogják végrehajtani, azaz a redundanciának eme formája a rendszer különböző perspektíváit nyújtja.

Sajnálatos módon, a diverzitás és a redundancia a rendszereket sokkal összetettebbé, és így nehezebben érthetővé is alakítja. Egyúttal annak a valószínűsége megnő, hogy a programozók hibákat vétsenek, illetve csökken annak az esélye, hogy az ellenőrzést végző emberek észrevegyék a hibákat. Ennek a következménye, hogy vannak, akik úgy gondolják, hogy jobb elkerülni a redundanciát és a diverzitást a szoftverekben, a rendszereket tervezzük a lehető legegyszerűbb módon, és alkalmazzunk különösen szigorú verifikációs és validációs folyamatokat (Parnas és mások, 1990). Azonban a kereskedelmi biztonságkritikus rendszerekben mindkét megközelítést alkalmazzák. Az Airbus 340 repülésirányító rendszerre például diverz és redundáns (Storey, 1996), míg a Boeing 777 repülésirányító rendszere a szoftver egyetlen verzióján alapul.

A szoftvertechnológiai kutatások célja, hogy olyan eszközök, technikák és módszerek fejlődjenek ki, amelyek hibamentes szoftverek előállításához vezetnek. A hibamentes szoftver olyan szoftver, amely pontosan megfelel a specifikációjának. Ez természetesen nem jelenti azt, hogy a szoftver sohasem fog hibázni. A specifikációban is rejtőzhetnek hibák, amelyek majd megjelennek a szoftverben, továbbá a felhasználók is alkalmazhatják, illetve értelmezhetik hibásan magát a szoftverrendszert. Mindazonáltal a szoftverhibák kiiktatása bizonyosan nagy hatással lehet a rendszerhibák számára.

Kis- és közepes méretű rendszerek esetében a szoftvertechnológiai technikáink segítségével jó eséllyel fejleszthetünk hibamentes szoftvereket. A cél elérésének érdekében számos szoftvertechnológiai technikát kell alkalmaznunk:

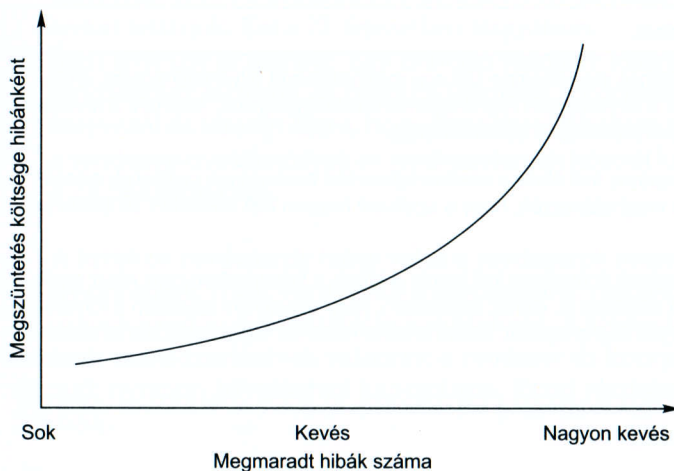
1. *Üzembiztos szoftverfolyamat.* Az üzembiztos szoftverfolyamat (20.1. fejezet) használata hozzáálló verifikációs és validációs tevékenységekkel elengedhetetlen, ha minimalizálni akarjuk a programhibák, valamint az átszúszó, de detektálható hibák számát.
2. *Minőségkezelés.* A szoftvert fejlesztő szervezetnek rendelkeznie kell olyan helyi minőségkultúrával, ahol a szoftverfejlesztés hajtóereje a minőség. Általában elvárás a programozóktól a hibamentes programok írása. Rendelkeznie kell olyan szabályzattal, tervezési és fejlesztési szabványokkal, amelyek biztosítják ennek a betartását.
3. *Formális specifikáció.* Léteznie kell precíz (lehetőleg formális) rendszer-specifikációnak, amely az implementálandó szoftvert definiálja. Számos tervezési és programozási hiba vezethető vissza a zavaros vagy nem kellően kifejtett specifikáció félreértelmezésére.
4. *Statikus verifikáció.* A statikus verifikációs technikák, mint például a statikus elemzők, számos rendellenes programsajátosságot tárhatnak fel, amelyek utalhatnak hibákra. Ezenkívül alkalmazható még a rendszer-specifikáción alapuló formális verifikáció is.
5. *Erős típusosság.* A fejlesztésre szigorúan típusos nyelveket kell alkalmazni, mint például a Java vagy az Ada. Szigorúan típusos nyelvben sok programo-

zási hibát már a nyelv fordítóprogramja is felismerhet, elkerülve, hogy bekerüljenek a szállított termékbe.

6. *Biztonságos programozás.* Néhány programozási nyelvben léteznek olyan programozási szerkezetek, amelyek a többitől eltérően összetettebbek és hibára hajlamosabbak, használatukkal pedig a hibák előfordulási gyakorisága megnövekedhet. A biztonságos programozás azt jelenti, hogy elkerüljük, de legalábbis minimalizáljuk ezeknek a szerkezeteknek a használatát.
7. *Védett információ.* Az egységbezáráson és az információrejtésen alapuló szoftvertervezési és implementációs szemléletet kell követni. Az objektumorientált nyelvek, mint a Java, nyilván kielégítik ezt az elvárást. Olvasható és érthető programok tervezését kell szorgalmazni.

Ezen technikák közül már többet is érintettem a könyv különböző fejezeteiben. Ebben a részben azokra az üzembiztos szoftverfolyamatokra és programozási technikákra összpontosítok, amelyek elősegítik a hibaelkerülést.

Úgy gondoljuk, hogy a szoftverfejlesztő technikáinkkal a hibamentes szoftver elérhető, de gazdaságilag nem jók. Rendkívül bonyolult és költséges ennek a célnak az elérése. A megmaradt hibák megtalálásának és eltávolításának költsége a validáció során exponenciálisan növekszik (20.1. ábra). Miközben a szoftver megbízhatóbbá válik, egyre több tesztre van szükség egyre kevesebb hiba megtalálásához.



20.1. ábra. A megmaradt hibák megszüntetésének növekvő költségei

Következésképpen a szoftverfejlesztő szervezetek explicit vagy implicit módon elfogadják, hogy az általuk fejlesztett szoftver néhány rezidens módon megmaradó hibát tartalmaz. A hibák szintje függ a rendszer típusától. A dobozos rendszerek viszonylag több hibát tartalmaznak (habár ezek is sokkal jobbak, mint 10 évvel ezelőtt), miközben a kritikus rendszereknél rendszerint sokkal kisebb a hibasűrűség.



A hibák elfogadásának az adhat értelmet, hogy olcsóbb lehet a hibás működés következményeit kifizetni, mint felfedezni és eltávolítani egy hibát a rendszer leszállítása előtt. Ez eléggé általános gyakorlat a személyi számítógépek szoftvertermékeinek forgalmazóinál. Amint azt a 3. fejezetben már említettem, hibás szoftver kiadása mellett dönteni azonban nem egyszerűen gazdasági döntés. A rendszer hibás működésének társadalmi és politikai hatását is mérlegelni kell.

## 20.1. MEGBÍZHATÓ FOLYAMATOK

A megbízható szoftverfejlesztési eljárások szoros kapcsolatban vannak a hibaelkerüléssel és a hibadetektálással. A megbízható szoftverfejlesztési folyamat jól definiált, ismételhető, a validációs és verifikációs tevékenységek egész spektrumát foglalja magában. A jól definiált folyamat szabványosított és dokumentált. Az ismételhető folyamat nem alapoz egyéni interpretációra és megítélésre. Sőt tekintet nélkül arra, hogy kik kapcsolódnak be a fejlesztésbe, a szervezet bizonyos lehet abban, hogy a folyamat sikeres lesz. A könyv 28. fejezete szól a fejlesztési folyamatok fontosságáról a szoftverminőségben, illetve a folyamat továbbfejlesztéséről. A megbízható szoftverfejlesztési folyamatok legfontosabb jellemzői a 20.2. ábrán láthatók.

| Jellemző        | Magyarázat                                                                                                                                                                                                                                 |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Dokumentálható  | A folyamatnak rendelkeznie kell egy meghatározott folyamatmodellel, amely felsorolja a folyamatban végrehajtandó tevékenységeket, valamint a tevékenységek által előállítandó dokumentációt.                                               |
| Szabványosított | Rendelkezésre kell állnia a szoftverfejlesztési szabványok gyűjteményének, amelyek meghatározzák, hogy a szoftvert hogyan kell előállítani és dokumentálni.                                                                                |
| Auditálható     | A folyamatnak érthetőnek kell lennie a külső, a folyamatban nem részt vevő emberek számára is, annak érdekében, hogy ellenőrizni lehessen a folyamat szabványos végrehajtását, illetve további hasznos változtatásokat lehessen javasolni. |
| Diverz          | A folyamatnak tartalmaznia kell redundáns és diverz verifikációs és validációs tevékenységeket.                                                                                                                                            |
| Robusztus       | A folyamatnak képesnek kell lennie arra, hogy az egyes tevékenységek hibáit korrigálni tudja.                                                                                                                                              |

**20.2. ábra.** A megbízható folyamatok jellemzői

A fejlesztési folyamatnak tartalmaznia kell jól megtervezett, minden részletre kiterjedő verifikációs és validációs tevékenységeket annak érdekében, hogy a szoftverben maradt hibákat még a kihelyezés előtt feltárhassák. A hiba elkerülése és a hiba detektálására vonatkozó tevékenységek a következők:

1. *Követelményvizsgálatok.* Mint ahogyan azt a 7. fejezetben írtam, ezek célja, hogy felfedjék a rendszer-specifikáció problémáit. A szállított szoftverekben megjelenő hibák nagy hányada a hibás követelményekből adódik. Ha ezek felfedezhetők és kiküszöbölhetők a specifikációból, akkor ez a hibaosztály minimalizálható.
2. *Követelménykezelés.* A 7. fejezetben is tárgyalt követelménykezelés azzal kapcsolatos, hogy nyomon követjük a követelményekben bekövetkezett változásokat, és ezeket érvényesítjük a tervezés és implementáció folyamán. A szállított rendszerekben előforduló számos hiba annak következménye, hogy nem sikerült biztosítani a követelményekben bekövetkezett változások figyelembevételét a tervezésnél és az implementációban.
3. *Modell ellenőrzése.* A modellellenőrzés a rendszermodellek automatikus ellenőrzését jelenti valamilyen CASE-eszköz segítségével, hogy megállapítsuk, a modelleknek megvan-e a belső és külső konzisztenciája. A *belső konzisztencia* azt jelenti, hogy a modell önmagában konzisztens, míg a *külső konzisztencia* azt jelenti, hogy a rendszer különböző modelljei (például állapotmodell és objektummodell) konzisztensek.
4. *Terv- és kódvizsgálat.* Mint azt a 22. fejezetben tárgyalom, a terv- és kódvizsgálatok gyakran a közönséges hibalistákon alapulnak, és céljuk ezeknek a hibáknak az eltávolítása még a rendszer tesztelése előtt.
5. *Statikus elemzés.* A statikus elemzés a programelemzés automatizált módszere, amelynek során a programot részletesen elemezzük, hogy a hibára utaló helyeket feltárjuk. Ezt a 22. fejezetben tárgyalom.
6. *Teszt tervezése és kezelése.* Egy minden részletre kiterjedő tesztkészletet kell a rendszerhez tervezni, és magát a tesztelési folyamatot is gondosan kell megszervezni és lebonyolítani, hogy biztosítani lehessen a rendszertesztek, illetve a rendszerkövetelmények és rendszertervek közötti kapcsolatokat. A tesztelés a 23. fejezet témája.

A kritikus rendszerek hibái néha a rendszerek rossz komponenseinek vagy a komponensek rossz verzióinak integrálásából származnak. Ennek elkerülésére szigorú konfigurációkezelésre van szükség. A *konfigurációkezelés* a szoftverváltozások menedzselésével, valamint a rendszer és komponensei különböző verzióinak nyomon követésével kapcsolatos. Ezzel részletesebben a 29. fejezet foglalkozik.

## 20.2. MEGBÍZHATÓ PROGRAMOZÁS

A megbízható programozás olyan programozási konstrukciókat és technikákat foglal magában, amelyek elősegítik a hibák elkerülését és a hibatűrést. A programhibák azért következnek be, mert a programozók hibákat vétének. Míg a hibák egyik része a specifikáció félreértéséből ered, a másik része viszont a túlságosan összetett programok, illetve az eredendően hibára hajlamos konstrukciók



használatából. Ezért a megbízhatóság miatt törekednünk kell az egyszerűsége, az információ védelmére a jogosulatlan hozzáférésekkel szemben, és minimalizálni kell a potenciálisan nem biztonságos programozási konstrukciók használatát.

A hibatűrő programozási technikák azon a tényen alapszanak, hogy különbség van a hiba és a meghibásodás között. A *meghibásodás* az, ami megfigyelhető a rendszert használó felhasználók által, míg a *hiba* egy belső rendszerjellemző. Ha hiba lép fel egy programfuttatás közben, akkor a detektálásával és javításával ez még tolerálható, megelőzve egy rendszer-meghibásodás bekövetkezését. Ebben a részben szó lesz még a kivételkezelésről is, amelynek a segítségével programjainkat hibatűrőbbé és könnyebben érthetővé tehetjük.

### 20.2.1. Információrejtés

A katonai szervezetektől eredő védelmi elv a „tudni kell” elve. Az információ csak azokhoz a személyekhez juthat el, akiknek erre a munkájuk elvégzéséhez szükségük van. Azt az információt, amely munkavégzésükhöz nem releváns, elhallgatják előlük.

Amikor programozunk, hasonló elvet fogadhatunk el a rendszeradatokhoz történő hozzáférésre. A programkomponenseknek csak azokhoz az adatokhoz kell a hozzáférést megengedni, amelyek szükségesek az implementációjukhoz. Más adatokhoz történő hozzáférést meg kell tagadni a programozási nyelv hatásköri szabályainak alkalmazásával. Ha az információrejtést alkalmazzuk, az elrejtett információt nem lehet elrontani azokkal a programkomponensekkel, amelyeknek nem megengedett az információ használata. Ha az interfész változatlan marad, az adatrepresentációt megváltoztathatjuk anélkül, hogy ez a rendszer más komponenseit érintené.

Az információrejtés sokkal egyszerűbb a Javában, mint olyan régebbi programozási nyelvekben, mint a C és a Pascal. Ezek a nyelvek nem rendelkeznek bezárási mechanizmussal, mint az objektumosztályok, és így az adatszerkezetek implementációjának részletei nem védhetők. Más programrészek direkt módon elérhetik a szerkezetet. Ez változtatások esetén nem kívánt mellékhatásokhoz vezethet.

Objektumorientált nyelven programozva általában jó programozási megoldás olyan metódusok definiálása, amelyek maguk érnek el és módosítanak attribútumértékeket ahelyett, hogy az attribútumok közvetlen elérését más objektumoknak megengednék. Ez azt jelenti, hogy az attribútum deklarációja különösebb hivatkozás nélkül megváltoztatható azokra az objektumokra, amelyek az attribútumot felhasználják. Gyakorlati szempontból is fontos, hogy ez a szemlélet használatos adatszerkezetekre és más komplex szerkezetekre.

A Java interfész-definíciós konstrukciója azt jelenti, hogy lehetséges ennek a szemléletnek az érvényesítése, és lehet egy objektum interfészét az implementációjára történő hivatkozás nélkül deklarálni. Ezt illusztrálja a 20.3. ábra. A Queue (sor) típusú objektum használói objektumokat helyezhetnek el a sorba, eltávo-

líthatják őket a sorból, és lekérdezhetik a sor méretét. Abban az osztályban ellenben, amelyik implementálja ezt az interfészt, a sor aktuális implementációjának az objektumok és metódusok attribútumát privátnak nyilvánítva, az illető osztályra vonatkoztatva rejtve kell maradnia. Az interfészeknek az implementációjuktól történő elválasztása a komponensalapú szoftverfejlesztési módszerek meghatározó része.

```
interface Queue {  
  
    public void put (Object o) ;  
    public void remove (Object o) ;  
    public int size () ;  
  
} //Queue
```

**20.3. ábra.** Egy sor specifikációja Java interfész-deklaráció segítségével

Az információrejtés egy hasonló esete található a 20.4. ábrán. Azokban a szituációkban, amikor korlátozott számú értéket lehet egy változóhoz rendelni, ezeket az értékeket konstansokként kell deklarálni. Az olyan programozási nyelvek, mint a C++ támogatják a felsorolásos típusokat, de a Javában ezeket úgy kell implementálni, hogy összekapcsoljuk őket az osztálydeklarációval.

```
class Signal {  
  
    static public final int red = 1 ;  
    static public final int amber = 2 ;  
    static public final int green = 3 ;  
  
    public int sigState ;  
  
}
```

**20.4. ábra.** Egy Signal osztály deklarációja Javában, ami elrejtí a típus reprezentációját

Tekintsünk például egy jelzőrendszert, amely vörös, sárga és zöld jelzéseket adhat, és ezt Javában implementáljuk.

Egy Signal típust kell definiálnunk, amely tartalmazza az ezeket a színeket tükröző konstansokat. Ezért lehetséges hivatkozni a Signal.red-re, Signal.green-re stb. Azonfelül, hogy ezzel védjük a Signal típusú változókat a helytelen értékadáستól, egyben elrejtjük a piros, a sárga és a zöld reprezentációját is. Ezen konstansok értékét így a későbbiekben bármikor megváltoztathatjuk anélkül, hogy bárhol máshol módosítani kellene a programban.



## 20.2.2. Biztonságos programozás

A programok hibái, és ebből következően a hibás programműködés, gyakran emberi hibák következményei. A programozók hibáznak, mert nem tudják nyomon követni az állapotváltozók között fennálló valamennyi összefüggést. Olyan programutasításokat írnak, amelyek váratlan viselkedést eredményeznek, és a rendszer állapota megváltozik. Az emberek mindig is követnek majd el hibákat, de már a 60-as évek végére világossá vált, hogy bizonyos programozási szemléletek sokkal inkább hibára hajlamosak, mint mások.

Dijkstra (Dijkstra, 1968) felismerte, hogy a goto utasítás vagy feltétel nélküli ugrás eredendően hibára hajlamosító programozási szerkezet. Bonyolulttá teszi az állapotváltozások lokalizálását. Ez a megfigyelés vezetett az úgynevezett strukturált programozás kidolgozásához. A strukturált programozás goto nélküli programozást jelent, azaz csupán while és if vezérlési szerkezetekkel való programozást, továbbá fentről lefelé történő tervezési szemléletet. A strukturált programozás adaptálása a szoftvertervezés fejlődésének fontos mérföldköve lett, mivel ez volt az első elmozdulás a szisztematikus szoftverfejlesztés irányába.

Van néhány más, hibára hajlamos programnyelvi konstrukció és programozási technika is. Kisebb az esélye, hogy hibákat viszünk be a programba, ha elkerüljük, de legalábbis ritkábban használjuk ezeket. Potenciális hibára hajlamos szerkezetek a következők:

1. *Lebegőpontos számok.* A lebegőpontos számok eredendően pontatlanok. Ez részben problémát jelent, ha összehasonlítjuk őket, mert az ábrázolási pontatlanság érvénytelen összehasonlításhoz vezethet. Például a 3.00000000 néha úgy ábrázolható, mint a 2.99999999, máskor mint a 3.00000001. Az összehasonlítás ezeket különbözőknek mutatná. A fixpontos számok, ahol egy szám adott számú decimális számjeggyel ábrázolódik, biztonságosabb, mert pontos összehasonlítás lehetséges.
2. *Mutatók.* A mutatók alacsony szintű konstrukciók, amelyek a számítógép memóriájának egyes területeire közvetlenül hivatkozó címeket tartalmaznak. Ezek veszélyesek, ugyanis a hibás használatuk romboló lehet, mert megengedik az „álnevek” alkalmazását (lásd alább), és mert megnehezítik a tömbök és egyéb struktúrák indexhatárainak ellenőrzését.
3. *Dinamikus memóriakiosztás.* A programhoz futási időben is rendelhető memória, szemben a fordítási időben történő hozzárendeléssel. Ezáltal az a veszély fenyeget, hogy a memóriát esetleg nem szabadítják fel, és elfogy a rendszer rendelkezésére álló memória. Ez egy nagyon finoman szövevényes hibatípus, mert a rendszer hosszú ideig működhet normálisan a problémák jelentkezése előtt.
4. *Párhuzamosság.* A párhuzamosság veszélyes, mert nehéz megjósolni az időzítési kölcsönhatások finom hatását a párhuzamos folyamatok között. Az időzítési problémák általában nem tárhatók fel programvizsgálattal, és a körülmények különleges kombinációja, amely végül az időzítési problémához vezet, nem nagyon fordul elő a rendszer tesztelése alatt. Lehet, hogy a

- párhuzamosság elkerülhetetlen, de a használatát gondosan ellenőrizni kell, hogy minimalizáljuk a folyamatok közötti függőségeket. Programnyelvi lehetőségek, mint az Ada-taszkok vagy a Java-szálak, segíthetnek úgy kezelni a párhuzamosságot, hogy a programozási hibák elkerülhetők legyenek.
5. *Rekurzió.* Rekurziónak nevezzük azt a helyzetet, amikor egy eljárás vagy módszer önmagát hívja, vagy egy másikat hív, de az eredeti hívó eljárást az hívja. Használata tömör programokat eredményezhet, de nehezen követhetővé válhat a rekurzív program logikájának követése. Ezért a programhibákat is nehezebb felfedezni. A rekurzió során elkövetett hibák a teljes memória allokálásához vezethetnek, mivel ideiglenes veremváltozók jönnek létre.
  6. *Megszakítások.* A megszakítások a vezérlés kényszerű átadásának eszközei egy másik kódszekciónak, tekintet nélkül a végrehajtás alatt álló kódra. Ennek veszélyei nyilvánvalók, mert a megszakítás egy kritikus műveletet is felfüggeszthet.
  7. *Öröklődés.* Az öröklődés az objektumorientált programozási nyelvekben támogatja az újrafelhasználást és a problémafelbontást, de azt is jelentheti, hogy egy objektumhoz társult kód nem egészen egy helyen található. Ez bonyolultabbá teszi az objektum viselkedésének megértését. Ezért valószínűbb, hogy programozási hibákat elnézünk. Ha pedig az öröklődés dinamikus kötéssel kombinálódik, futási időben időzítési problémákhoz vezethet. Különböző pillanatokban egy speciális módszer különböző példányait hívhatjuk, és különböző hosszúságú időre lehet szükség a helyes módszerpéldány megkeresésére.
  8. *Álnevek használata.* Akkor fordul elő, ha a programban ugyanannak a dolognak az elnevezésére különböző neveket használunk. A program olvasójának nagyon könnyű elvétenie az adott dolgot módosító utasítást, ha több nevet is figyelnie kell.
  9. *Nem korlátozott tömbök.* Az olyan nyelvekben, mint például a C, a tömbök igen egyszerű utat nyújtanak a memória-hozzáféréshez, bár könnyedén hivatkozhatunk a tömb végén túlra is. A futtatórendszer nem ellenőrzi, hogy a hivatkozás ténylegesen a tömb elemeire vonatkozik-e. A puffertúlcsordulás, ahol a támadó előre kitervelt módon olyan programot ír, amely egy – tömbként implementált – pufférhez rendelt memóriaterület vége után ír, ismert védelmi probléma.
  10. *Alapértelmezett input feldolgozása.* Bizonyos rendszerek akkor is feltételeznek valamilyen bemenetet, ha az ténylegesen nem érkezett be. Ez védelmi lyuk, mert a támadók ezt kihasználva olyan váratlan bemenetet adhatnak meg, amit a rendszer nem utasít vissza.

Egyes, a biztonságosságkritikus rendszerek fejlesztésére vonatkozó szabványok teljesen megtiltják ezeknek a szerkezeteknek a használatát. Azonban ez a szélsőséges álláspont normális esetben nem célszerű. Mindezek a szerkezetek hasznosak, csak körültekintően kell őket használni. Ahol csak lehetséges, potenciálisan veszélyes hatásukat absztrakt adattípusokon és objektumokon belül történő alkalmazásukkal kell ellenőrzésünk alatt tartani. Ezek természetes „tűzfalként” viselkednek, korlátozva a hibák fellépésével okozott kárt.



A Java nyelv tervezői felismerték a hibára hajlamos szerkezetek problematikáját. A nyelvben nem létezik eszköz a feltétel nélküli vezérlésátadásra (nincs goto utasítás), beépített hulladékgyűjtés van, így nincs szükség a dinamikus memóriakiosztására, és nem támogatja a mutatókat, illetve a méret nélküli tömböket. Ellenben a numerikus adatok reprezentációja olyan, hogy a túlsordulást a futtatórendszer nem veszi észre, s így előfordulhat lebegőpontos műveletek által okozott hibás működés.

### 20.2.3. Kivételkezelés

A programok futtatása során elkerülhetetlenül felléphetnek hibák vagy nem várt események. Ezek bekövetkezhetnek programhiba vagy akár egy előre nem látható külső körülmény hatására is. Ha valamilyen hiba vagy nem várt esemény következik be egy program végrehajtása során, akkor ezt *kivételnek* nevezzük. Kivételeket okozhatnak hardver- vagy szoftverfeltételek. Kivételekre példa az áramkimaradás, egy nem létező adatelem elérésének megkísérlése, numerikus túlsordulás vagy alulsordulás stb.

Kivétel fellépésekor azt a rendszernek kell kezelnie. Ez megtehető magában a programban, vagy átadható a vezérlés a rendszer kivételkezelő mechanizmusának. Tipikusan a rendszer kivételkezelő mechanizmusa egyszerűen jelenti a hibát, és leállítja a futtatást. Éppen ezért, annak biztosítására, hogy a programkivételek nem okoznak rendszerhibát, minden várható kivételhez kivételkezelőt kell definiálni, és biztosítani kell, hogy minden kivétel explicit kezelve legyen.

Olyan programozási nyelvekben, mint a C, if utasításokat kell használnunk a kivétel észlelésére és a vezérlés átadására a kivételkezelő rutinnak. Ez azt jelenti, hogy explicit ellenőrizni kell a kivételeket mindenhol, ahol előfordulhatnak. Mindamellett, hogy ez növeli a komplexitást, növeli annak az esélyét, hogy hibákat vétünk, és a kivételeket nem kezeljük megfelelően.

Néhány programozási nyelv (pl. Java, C++, Ada) tartalmaz olyan szerkezeteket, amelyek a kivételkezelést támogatják, így nem kell külön feltételes utasításokat írunk a kivétel bekövetkezésének ellenőrzésére. Néha a programozási nyelv támogat egy speciális beépített típust (a neve általában *Exception*) és különböző kivételek deklarálhatók ezzel a típussal. Ha egy kivételes helyzet adódik, a kivétel kiváltódik, és a programozási nyelv futtatórendszere átadja a vezérlést a kivételkezelőnek. Ez egy olyan kódrészlet, amely megállapítja a kivétel nevét és a teendőket minden egyes kivételtípus esetében.

A Javában újabb kivételtípusok deklarálhatók a beépített *Exception* osztály kiterjesztésével. A Javában a kivételek a *throw* utasítás segítségével válthatók ki. Egy kivételkezelő a *catch* kulcsszóval kezdhető, amelyet a kivételt kezelő utasítássorozat kódja követ.

A 20.5. ábra a kivételek használatát illusztrálja Javában. A példa része a 3. fejezetben bevezetett inzulinpumpa szoftverének. Ez egy szenzorvezérlő, amely a vércukorszint értékét olvassa egy szenzorról. A 20.5. ábrán az első deklaráció

```
class SensorFailureException extends Exception {

    SensorFailureException (String msg) {
        super (msg);
        Alarm.activate (msg);
    }
} // SensorFailureException

class Sensor {
    int readVal () throws SensorFailureException {

        try {
            int theValue = DeviceIO.readInteger ();
            if (theValue < 0)
                throw new SensorFailureException ("Szenzorhiba!");
            return theValue;
        }
        catch (deviceIOException e)
        {
            throw new SensorFailureException ("Szenzorolvasási hiba!");
        }
    } // readVal
} // Sensor
```

**20.5. ábra.** Kivételek az inzulinpumpa hibáinak kezelésére

megmutatja, hogyan deklaráljuk a kivételeket Javában. Az `Exception` nevű beépített osztályt kiterjesztettük, és a konstruktor módszer definiálja a végrehajtandó kódot, amit végre kell hajtani a kivétel bekövetkezésekor. Ebben az esetben egy riasztás aktivizálódik.

A `Sensor` osztály tartalmaz egy `readVal` nevű módszert, amely deklarációjában tartalmaz egy kivételdobó utasítást. Ez azt jelenti, hogy a `SensorFailureException` kiváltódhat a módszerben, de az nem kezeli le, ezért a hívási környezetben kell lenni egy kivételkezelőnek a `SensorFailureException` kivételhez. A kivételek számára általában az a legjobb, ha a hívó metódus kezeli, mert az a módszer tudja, hogy mit akar kezdeni a meghívott módszer eredményével. Mindezek ellenére, a későbbiekben arra is mutatok példát, hogy léteznek olyan szituációk is, amikor jobb, ha a kivételt lokálisan kezeljük, mert biztosítanunk kell, hogy a módszerhívás mindig érvényes legyen.

A `try` kulcsszót követő blokkban váltódhat ki a kivétel. A `SensorFailureException` kivétel akkor váltódik ki, ha a szenzor ellenőrzése során nullánál kisebb értéket kapunk. A `DeviceIO.readInteger` metódus eldobhat egy `deviceIOException` kivételt, ennek kezelését végzi a `catch` kulcsszót követő kódrészlet. Itt most a kezelő kivált egy `SensorFailureException` kivételt, és ezzel a kivétel kezelését átadja a hívó környezetnek.

A programozási nyelvek által tartalmazott kivételkezelő eszközöket nem csupán hibakezelésre használhatjuk. Segíthetnek a programok egyszerűsítésében, illetve könnyítik az olvashatóságot és megértést. Csökkentik a programozói hi-



bák valószínűségét, és növelik annak az esélyét, hogy a programellenőrök megtalálják a bennmaradt hibákat. A kivételkezelő segítségével el tudjuk választani a kivételt kezelő kódot a normál feldolgozásért felelős kódtól.

Ezt illusztrálja a 20.6. ábra. Ez a Java-osztály egy fagyasztó hőszabályozójának vezérlője. A kívánt hőmérsékletet  $-18\text{ }^{\circ}\text{C}$  és  $-40\text{ }^{\circ}\text{C}$  között lehet szabályozni. A lefagyasztott élelmiszer  $-18\text{ }^{\circ}\text{C}$  felett kiolvadásnak indulhat, és a baktériumok elszaporodhatnak. A vezérlőrendszer fenntartja ezt a hőmérsékletet, ki- és bekapcsolva egy hűtőszivattyút, az érzékelő által mutatott hőmérséklettől függően. Ha a kívánt hőmérséklet nem tartható fenn, a vezérlő vészjelet bocsát ki.

```
class FreezerController {
    Sensor tempSensor = new Sensor ();
    Dial tempDial = new Dial ();
    float freezerTemp = tempSensor.readVal ();
    final float dangerTemp = (float) - 18.0 ;
    final long coolingTime = (long) 200000.0 ;

    public void run () throws InterruptedException {
        try {
            Pump.switchIt (Pump.on);
            do {
                if (freezerTemp > tempDial.setting ())
                    if (Pump.status == Pump.off)
                    {
                        Pump.switchIt (Pump.on);
                        Thread.sleep (coolingTime);
                    }
                else
                    if (Pump.status == Pump.on)
                        Pump.switchIt (Pump.off);

                if (freezerTemp > dangerTemp)
                    throw new FreezerTooHotException ();
                freezerTemp = tempSensor.readVal ();

            } while (true) ;

        } // try blokk
        catch (FreezerTooHotException f)
        { Alarm.activate () ; }
        catch (InterruptedException e)
        {
            System.out.println ("szál kivétel");
            throw new InterruptedException ();
        }

    } // run
} // FreezerController
```

**20.6. ábra.** Kivételek egy fagyasztó hőmérséklet-szabályozójánál

A Java-implementációban a fagyasztó hőmérséklete a `tempSensor` nevű objektum és a kívánt hőmérséklet összevetésével deríthető ki, amit a `tempDial` nevű objektum felügyel. Egy szivattyú objektum (`Pump`) az állapotának megváltozásával válaszol a jelre. Ha a szivattyú bekapcsolódott, a rendszer egy ideig vár (a `Thread.sleep` hívásával) a hőmérséklet csökkenésére. Ha az nem csökkent eléggé, akkor egy `FreezerTooHotException` kivétel váltódik ki.

A kivételkezelő (amely a kód végén helyezkedik el) megkapja a kivételt, és aktivizálja az `Alarm` objektumot. Egy másik kezelő is van a beépített `InterruptedException` számára, amelyet a `Thread.sleep` képes kiváltani. Ez naplózza a kivételt, aztán visszaadja azt a `main` módszernek.

### 20.3. HIBATŰRÉS

A hibatűró rendszer folytatni tudja működését abban az esetben is, ha rendszerhibák lépnek fel. A hibatűrés célja biztosítani, hogy a rendszerhibák ne vezessenek sikertelen rendszerműködéshez. A hibatűrésre ott van szükség, ahol a sikertelen működés katasztrófához vezethet, vagy a rendszer működésében előforduló kihagyás nagy gazdasági hátrányokkal járna. Például a repülőgép fedélzeti számítógépeinek addig kell működniük, míg le nem szállt a gép; a légi közlekedés irányító rendszerének pedig addig, míg csak egy gép is van a levegőben.

A hibatűrés négy vonatkozását különböztetjük meg:

1. *Hiba detektálása.* A rendszernek fel kell ismernie az olyan rendkívüli állapotkombinációt, amely hibás működéshez vezethet. Általában ez a rendszerállapotok konzisztenciájának ellenőrzését jelenti.
2. *Veszteség felmérése.* Meghatározandók a rendszer állapotának azon részei, amelyeket a hiba érintett.
3. *Hiba utáni helyreállítás.* A rendszernek tudnia kell az állapotát biztonságosra visszaállítani. Lehet, hogy ezt a tönkretett állapot javításával éri el (előreható helyreállítás), vagy egy korábbi ismert biztonságos állapot visszaállításával (visszaható helyreállítás).
4. *Hibajavítás.* Ez magában foglalja a rendszer olyan jellegű módosítását, hogy a hiba ne térjen vissza. A szoftverhibák sokszor átmeneti hibaként jelennek meg. Ezeket a rendszer bemeneteinek valamilyen nagyon különleges kombinációja váltja ki. Nem szükséges a javítás, mert a normális feldolgozási mód a hiba helyreállítása után közvetlenül visszaáll.

Azt gondolhatnánk, hogy azokban a rendszerekben, amelyeket a hibaminimalizáló módszertannal fejlesztettek, már nem szükséges a hibatűró képesség beépítése. Ha nincsenek hibák a rendszerben, akkor nincs esély a hibás működésre. A „hibamentes” azonban nem azt jelenti, hogy „hibátlan működésű”. Csupán annyit jelent, hogy a program megfelel a specifikációjának. A specifikáció is



tartalmazhat hibákat vagy hanyagságot, esetleg alapulhat hibás feltételezéseken a rendszer környezetét illetően. És természetesen mi soha nem tudjuk következetesen demonstrálni, hogy a rendszer teljesen hibamentes. Azokban a rendszerekben tehát, amelyek magas fokú megbízhatósági és rendelkezésre állási követelményeken alapulnak, a hibatűrés és hibaelkerülés redundáns és diverz megközelítéseit kell alkalmazni.

### 20.3.1. Hibadetektálás és a veszteség felmérése

A hibatűrés első lépéseként azt kell érzékelnünk, hogy egy hiba (hibás rendszerállapot) bekövetkezett vagy be fog következni, hacsak azonnal nem teszünk valamit ez ellen. Ennek megvalósításához ismernünk kell, hogy az állapotváltozóknak melyek a nem megengedett értékeik, illetve milyen esetekben nem tartható fenn közöttük a kapcsolat. Ezért olyan állapotmegszorításokat szükséges definiálni, amelyek rögzítik a feltételeket az összes megengedett állapot fenntartásához. Ha ezek az előfeltételek sérülnek, akkor hiba következik be. Az inzulinpumpa szoftverére vonatkozóan a 20.7. ábrán láthatunk állapotmegszorításokra példát. A példában szándékosan nem használtam Java-állításokat (assert), amelynek okát még a későbbiekben kifejtem.

```
// A szolgáltatott inzulinadagnak mindig nullánál nagyobbnak
// és egy megadott maximumnál kisebbnek kell lennie

insulin_dose >= 0 & insulin_dose <= insulin_reservoir_contents

// Az egy napon szolgáltatott inzulinadagoknak együttesen kisebbnek
// vagy egyenlőnek kell lennie mint egy megadott napi maximum

cumulative_dose <= maximum_daily_dosea
```

20.7. ábra. Állapotmegszorítások az inzulinpumpára

A hibadetektálásnak két típusa van:

1. *Preventív hibadetektálás.* Ebben az esetben a hibadetektáló mechanizmus a rendszerállapot megváltozása előtt aktivizálódik. Ha egy potenciálisan hibásnak tűnő állapotot érzékel, megakadályozza az állapotváltoztatást.
2. *Retrospektív hibadetektálás.* Ebben az esetben a hibadetektáló mechanizmus a hibák feltárására azután aktivizálódik, miután a rendszerállapot megváltozása bekövetkezett. Ha egy hibás állapotot érzékel, kivételt jelez, és egy javító mechanizmust használ a helyreállításra.

Preventív hibadetektálást akkor használhatunk, ha az állapotmegszorítások csak egyedülálló állapotváltozókra vonatkoznak. Például ezt a megközelítést al-

kalmazhatjuk, ha egy állapotváltozó értékének egy meghatározott intervallumba kell esnie. A preventív hibadetektálás megelőzi a kárhelyreállítással járó többletterhet, mert a rendszerállapot mindig érvényes lesz – csak nem feltétlenül helyes. Természetesen a rendszernek rendelkeznie kell egy mechanizmussal a működés folytatására hibás állapotok esetére is, hogy a rendszerhibát elkerüljük.

Javában a legbiztosabb mód a preventív hibadetektálás implementálására, ha explicit módon ellenőrizzük a hibákat, és ha egy hibás állapotot észlelünk, akkor a nyelv kivételkezelő mechanizmusát használjuk. Ezt illusztrálja a 20.8. ábra a pozitív páros számok osztályának egy implementációjával.

```
class PositiveEvenInteger {
    int val = 0 ;
    PositiveEvenInteger (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException () ;
        else
            val = n ;
    } // PositiveEvenInteger
    public void assign (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException () ;
        else
            val = n ;
    } // assign
    int to Integer ()
    {
        return val;
    } // toInteger
    boolean equals (PositiveEvenInteger n)
    {
        return (val == n.val) ;
    } // equals
} // PositiveEvenInteger
```

**20.8. ábra.** A PozitiveEventInteger osztály a Javában

Ha kísérlet történik olyan érték adására, amely páratlan vagy kisebb, mint 0, akkor egy kivétel (NumericException) váltódik ki.

A Java 1.4-ben került bevezetésre az állítás-ellenőrzés lehetősége, amelynek segítségével állapotmegszorításokat adhatunk meg állítások segítségével. Így ha azt szeretnénk kifejezni, hogy egy számnak pozitívnak és párosnak kell lennie, a következőt írhatjuk:

```
assert n >= 0 & n%2 == 0: "Az értéknek pozitívnak és párosnak kell lennie"
```

A futtatórendszer ellenőrzi, hogy a feltétel teljesül-e, és ha nem, akkor kivált egy kivételt, valamint végrehajtja a megadott üzenet megjelenítését.



Azonban a Java állítás-ellenőrző lehetősége eredetileg a fejlesztés és a belövés folyamán az inkonzisztens állapotok felderítését hivatott segíteni, nem pedig a biztonságos programozás támogatását. Az állítás-ellenőrzést lehetőségünk van ki- és bekapcsolni is, így nem bízhatunk mindig abban, hogy ezek ellenőrzésre kerülnek. Továbbá az egyes állításokhoz nem tudunk különböző típusú kivételeket sem rendelni, így a különböző állítás-ellenőrzési hibákat sem tudjuk egymástól megkülönböztetni. Ez átgondolt döntés volt a tervezés során – a fejlesztők ne számíthassanak arra, hogy egy állítás megsértése után még helyreállító tevékenységek következhetnek.

A preventív hibadetektálás akkor lehetséges, ha ismerjük azt az értéktartományt, amelyből egy állapotváltozó felveheti az értékeit. Azonban ha egy állapot esetében az egyik érték érvényessége egy másik változó értékétől függ, a preventív hibadetektálás nem mindig lehetséges. Például vegyünk azt az esetet, hogy a programunk három értéket szeretne beolvasni adott sorrendben, mondjuk A, B és C-t.

Az állapotmegszorítás a következő:

$A < B \ \& \ B < C$

Ebben az esetben nem alkalmazható a preventív hibadetektálás az A értékre, mert nem tudjuk, hogy mi lesz a B és C értéke. Hasonlóan, amikor a B-t olvassuk, nem tudjuk ellenőrizni, hogy kisebb-e, mint a C. Ezért retrospektív hibadetektálást kell alkalmazni, azaz az állapotmegszorítást csak azután ellenőrizzük, hogy mindhárom értéket beolvastuk. Amennyiben a megszorítás sérül, végrehajthatunk további lépéseket, hogy visszaállítsuk a rendszer konzisztenciáját.

A retrospektív hibadetektálás egyik lehetséges Java-implementációja, hogy egy ellenőrző függvényt társítunk az objektumokhoz. Az állapotváltások után ennek a függvénynek a meghívásával gondoskodhatunk az állapotmegszorítások fenntartásáról. A hívásokat csak akkor kell elvégezni, amikor szükségesek – ugyanis előfordulhat, hogy az állapotot nem szükséges minden egyes váltáskor ellenőrizni.

A következő interfészt használhatjuk az ellenőrző függvényekhez:

```
interface CheckableObject {  
    public boolean check ();  
}
```

Az ellenőrzendő objektumok egy olyan objektumosztály példányai, amelyek ezt az interfészt implementálják, és így minden objektumnak van egy kapcsolódó ellenőrző függvénye. Minden egyes objektumosztály implementálja a saját ellenőrző függvényét, amely definiálja azokat az egyedi megszorításokat, amelyeket ennek az osztálynak az objektumainál kell érvényesíteni. Ha az állapot egészében ellenőrzött, dinamikus kötés szükséges annak biztosítására, hogy éppen azt az ellenőrző függvényt alkalmazzuk, amely megfelelő az ellenőrzés alatt álló objektumosztály objektumainak ellenőrzésére. Ennek egy példáját láthatjuk a 20.9. ábrán, ahol az ellenőrző függvény ellenőrzi, hogy egy tömb elemei eleget tesznek-e bizonyos korlátozásoknak.

```

class RobustArray {
    // Ellenőrzi, hogy egy objektumtömb minden egyes objektuma
    // kielégít-e valamilyen adott korlátozásokat
    boolean [] checkState ;
    CheckableObject [] theRobustArray ;
    RobustArray (CheckableObject [] theArray )
    {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
    } // RobustArray
    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false ;
        for (int i = 0 ; i < this.theRobustArray.length ; i++)
        {
            if (! theRobustArray [i].check () )
            {
                checkState [i] = true ;
                hasBeenDamaged = true;
            }
            else
                checkState [i] = false ;
        }
        if (hasBeenDamaged)
            throw new ArrayDamagedException () ;
    } // assessDamage
} // RobustArray

```

**20.9. ábra.** Egy tömbosztály kárfelméréssel

Egynél több állapotváltozóra alkalmazott retrospektív hibadetektálást illusztrál a 20.10. ábra. Ebben a példában a hibadetektálási ellenőrzést egy tömb szomszédos elemeire alkalmazzuk annak megállapítására, hogy a tömb elemei rendezettek-e.

A veszteség felmérése a rendszer állapotának vizsgálatát jelenti abból a célból, hogy a hiba kiterjedését meghatározzuk. Ez sokszor elkerülhető a hiba előfordulásának figyelésével még az állapotváltozások nyugtázása előtt. Ha hibát fedezünk fel, az állapotváltozást nem fogadjuk el, így nincs károkozás. A veszteségfelmérést azonban nem kerülhetjük el, ha az állapotváltozás elfogadása elkerülhetetlen, vagy a hibás állapot egyenként helyes állapotváltások szabálytalan sorozatának eredményeként adódott.

A veszteségfelmérés szerepe nem a helyreállításban rejlik, hanem annak megállapításában, hogy a hiba az állapottér mely részeit érintette. A veszteség csak akkor mérhető fel, ha alkalmazható valamilyen „validitási függvény”, amely ellenőrzi, hogy az állapot konzisztens-e. Ha inkonzisztenciát fedez fel, akkor ezeket valamilyen módon jelzi.

A 20.9. ábrán illusztráljuk a veszteségfelmérés implementálását Javában. Felteesszük, hogy a RobustArray-nek nevezett adatstruktúra CheckableObject típusú objektumok egy kollekciója. Annak az osztálynak, amelyik a CheckableObject tí-



```

class SafeSort {
    static void sort ( int [] intarray, int order ) throws SortError
    {
        int [] copy = new int [intarray.length] ;
        // a bemenő tömb másolata
        for (int i = 0 ; i < intarray.length ; i++)
            copy [i] = intarray [i];
        try {
            Sort.bubblesort (intarray, intarray.length, order) ;
            if (order == Sort.ascending)
                for (int i = 0 ; i <= intarray.length - 2; i++)
                    if (intarray[i] > intarray[i+1])
                        throw new SortError () ;
            else
                for (int i = 0 ; i < intarray.length - 2; i++)
                    if (intarray[i+1] < intarray[i])
                        throw new SortError () ;
        } // try blokk
        catch (SortError e)
        {
            for (int i = 0 ; i < intarray.length ; i++)
                intarray [i] = copy [i] ;
            throw new SortError ("A tömb nem rendezett.") ;
        } // catch
    } // sort
} // SafeSort

```

**20.10. ábra.** Biztonságos rendezőeljárás visszaható hiba-helyreállítással

pust implementálja, tartalmaznia kell egy `check`-nek nevezett módszert, amelyik tesztelni tudja, hogy az objektum egy értéke eleget tesz-e bizonyos korlátozásoknak. Értelmesebb az ellenőrző módszert ezzel az objektummal társítani, mint a `RobustArray` objektummal, mert az ellenőrzés részletei a `CheckableObject` típusú objektum használatától függenek.

Az `assessDamage` módszer a `RobustArray` osztályban megvizsgálja a tömb minden elemét, és ellenőrzi, hogy az állapota helyes-e. Ha a tömb egy vagy több eleme nem felel meg azoknak a korlátozásoknak, amelyeket a `check` függvény definiál, akkor az elemek feljegyzésre kerülnek a `checkState` nevű tömbben. Kiváltódik egy `ArrayDamageException` nevű kivétel. E veszteség kezelőjének szerepnie kell a hívó módszerben. Felhasználhatja a `checkState`-ben őrzött információt annak eldöntésére, hogy mit kell tennie.

Más, hibadetektálásra és veszteségfelmérésre használt technikák függenek a rendszerállapot ábrázolásától és az alkalmazástól. A lehetséges módszerek a következők:

1. Ellenőrző összegek az adatcserénél és ellenőrző számjegyek a numerikus adatoknál.
2. Redundáns kapcsolók alkalmazása azoknál az adatszerkezeteknél, amelyek mutatókat tartalmaznak.
3. Órzkutya-időzítők alkalmazása a konkurens rendszereknél.

Kódel ellenőrzés (Fujiwara és Pradhan, 1990) ott alkalmazható, ahol az átvitt (numerikus) adatokhoz ellenőrző összeg is társul. Az ellenőrző összeg az adatokra alkalmazott matematikai függvényből adódó számérték. Az alkalmazott függvénynek egyértelműen meghatározott értéket kell szolgáltatnia az átvitt adatcsomagokra. Ezt az ellenőrző összeget kiszámítja a küldő, és hozzáfűzi az átvendő adatcsomaghoz. A vevő ugyanazt a függvényt alkalmazza az adatokra, és összehasonlítja a két ellenőrző összeg értékét. Ha különböznek, valamilyen adathiba léphetett fel. Ez a módszer alkalmazható betörés és szándékos adattorzítás esetén is. Az elrontott adat ellenőrző összege más, így a hiba felismerhető lesz.

Mutatókkal realizált adatszerkezetek esetében az ábrázolást visszamutató hivatkozásokkal tehetjük redundánssá. Azaz minden egyes A-ról B-re történő hivatkozás esetén van egy ehhez társítható hivatkozás B-ről A-ra. A struktúra elemeinek a számát is számon tarthatjuk. Egy ellenőrzés eldöntheti, hogy az előre- és visszamutató referenciák konzisztensek-e (ezeknek egymásra kell mutatniuk) és hogy a tárolt méret és a szerkezet kiszámított mérete megegyezik-e.

Ha a folyamatoknak meghatározott idő alatt kapcsolatba kell lépniük, őrzőkutya-időzítőt alkalmazhatunk. Egy őrzőkutya-időzítő olyan időzítő, amely alaphelyzetbe állítható, ha egy folyamat befejeződött. Együtt indítható egy folyamattal, és így mérni tudja a végrehajtási időt. Együttműködhet egy vezérlővel meghatározott időközökben. Ha valamilyen oknál fogva a folyamat nem akar megállni, az őrzőkutya nem áll alapra. Ezért a vezérlő detektálhatja, hogy probléma adódott, és kezdeményezheti a folyamat felfüggesztését.

### 20.3.2. Hiba helyreállítása és javítása

A hiba-helyreállítás folyamán úgy módosítjuk a rendszer állapotát, hogy a hiba hatása minimális legyen. A rendszer folytathatja a működését, habár valamilyen redukált formában. Az *előreható helyreállítás* megkísérli kijavítani a hibás rendszerállapotot és elérni a kívánt állapotot. A *visszaható helyreállítás* visszaállítja a rendszerállapotot egy korábbi „helyes” állapotra.

Az előreható hiba-helyreállítás csak akkor lehetséges, ha az állapotinformáció tartalmaz beépített redundanciát. Két általános (az előző részben már kifejtett) szituációban alkalmazható az előreható hiba-helyreállítási stratégia:

1. *Ha a kódolt adat sérült.* Azoknak a kódolási technikáknak a használata, amelyek az adatahoz redundanciát fűznek hozzá, lehetővé teszi a korrekciót, ha a torzulást észleltük.
2. *Ha a láncolt szerkezetek sérültek.* Ha az adatszerkezet előre- és visszamutató mutatókat is tartalmaz, a szerkezet újra felépíthető, ha nem túl sok mutató sérült. Ezt a módszert gyakran használják állományrendszerek és adatbázisok helyreállítására.

A visszaható helyreállítás egyszerűbb megoldás, amely egy hiba detektálása után visszaállítja a rendszer állapotát egy korábbi biztonságos állapotra. A leg-



több adatbázisrendszer visszaható hiba-helyreállítást tartalmaz. Ha egy felhasználó adatbázis-műveletet kezdeményez, tranzakció jön létre. A tranzakció okozta változások nem kerülnek be azonnal az adatbázisba. Az adatbázis csak akkor frissül, ha a tranzakció befejeződött és nem történt hiba. Ha a tranzakció sikertelen, az adatbázis nem módosul.

A tranzakciók lehetővé teszik a hiba-helyreállítást, mivel nem változtatnak az adatbázison, amíg be nem fejeződtek. Ugyanakkor nem teszik lehetővé a helyreállítást olyan változások esetében, amelyek érvényesek, de helytelenek. Az ellenőrző pont alkalmazása olyan technika, ami segíthet ebben a helyzetben. A rendszer állapota periodikusan duplikálódik. Ha felfedezünk egy hibát, a helyes állapot visszaállítható valamelyik másolatból.

A visszaható helyreállítás implementálására kivételek alkalmazásával tekintünk a 20.10. ábrán bemutatott SafeSort Java-osztályt, amely hibadetektálásra és korrekcióra szánt kódot is tartalmaz.

A módszer lemásolja a tömböt a rendezési művelet előtt. A példában az egyszerűség kedvéért a buborékrendezést alkalmaztuk, de bármely más rendezés is megfelelő lenne. Ha valamilyen hiba lép fel a rendezési algoritmus során, és a tömb nem lesz megfelelően rendezett, ez kiderül az elemek sorrendjének explicit ellenőrzéséből. Ha a tömb nem megfelelően rendezett, SortError kivétel jön létre. A kivételkezelő nem próbálja meg a hibát kijavítani, hanem visszaállítja az eredeti állapotát a tömbnek, és kiváltja a SortError-t, hogy jelezze a hívónak, hogy a rendezés sikertelen volt. A hívó felelőssége eldönteni, hogy miként folytatja a futást.

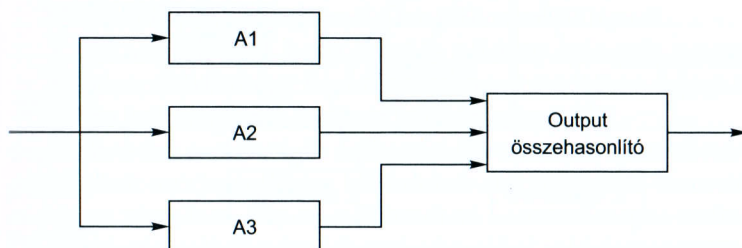
Ahogy arra már korábban utaltam, számos szoftverhiba tranziens, és ezért nem szükséges explicit javítás az általuk okozott körülmények korrigálására. A rendszer későbbi futásában már nem látszódnak. Ahol azonban ettől eltérő a helyzet, ott adott a lehetőség különböző javítások végrehajtására. A leggyakrabban alkalmazott szoftveres javítótevékenység a rendszer újrainicializálása, az állapotnak a kezdeti, biztos értékekre történő visszaállítása (Huang és Kintala, 1993). Ez néha a rendszer megállítása nélkül végrehajtható, feltéve, hogy az inicializálás gyors és a szolgáltatáskérélmeket lehet késleltetni. További javítási alternatívák, mint például a dinamikus újrakonfigurálás, normálisan csak akkor lehetségesek, ha a rendszer tervezésekor erre explicit rendelkezéseket hoztunk.

## 20.4. HIBATÚRÓ ARCHITEKTÚRÁK

A rendszerek többségénél megvalósítható a szoftveres hibatúrás implementációja, amennyiben explicit módon a szoftverben elhelyezünk ellenőrző és visszaállító tevékenységeket. Ezt nevezik *defenzív programozásnak*. A hardver és a szoftver együttműködése során keletkező hibákat azonban nem tudja hatékonyan kezelni. Sőt, a követelmények félreértése jelentheti azt, hogy helytelen a rendszerkód és a hozzá tartozó védelem is.

A legtöbb kritikus rendszernél, különösen a szigorú rendelkezésre állási követelményekkel rendelkezőknél, szükség lehet egy olyan specifikus rendszerarchitektúrára, amelyet a hibátűrés támogatására terveztek. Ezt a módszert alkalmazzák például az olyan repülőgépes rendszereknél, amelyeknek a repülés egész ideje alatt működniük kell, telekommunikációs rendszereknél, valamint vezérlő- és ellenőrző rendszereknél. Pullum (Pullum, 2001) írásában a tervezett hibátűrő architektúráknak több típusát is bemutatta.

Hibatűrő hardverek létrehozására szükség van. A leggyakrabban használt hardveres hibátűrő technika a háromszoros moduláris redundancia elvén alapul. A hardveregységet megháromszorozzák (vagy többszörözik). Ha az egyik egység elromlik, és nem ugyanazt az outputot adja, mint a többi, akkor azt az outputot a rendszer figyelmen kívül hagyja. Egy hibakezelő megpróbálhatja automatikusan megjavítani az egységet, de ha nem lehet, akkor a rendszer automatikusan újrakonfigurálódik úgy, hogy a hibás egység kiiktatódik. Ezután a rendszer két jó egységgel működik tovább (20.11. ábra).



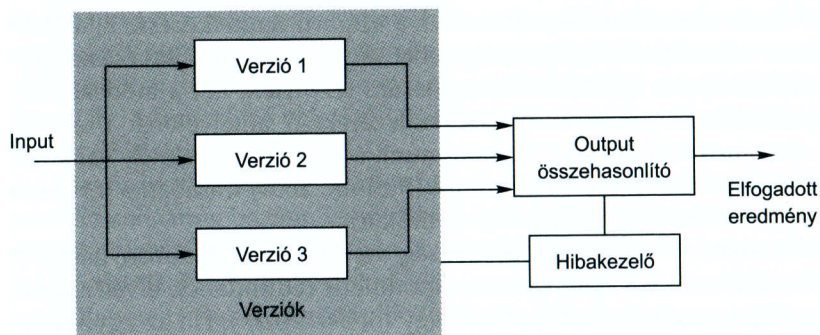
20.11. ábra. Háromszoros moduláris redundancia a hardverhiba leküzdésére

Ez a hibátűrési módszer abból indul ki, hogy a hardverhibák nagy része inkább az összetevők hibáiból származik, nem pedig tervezési hiba következménye. Ezért a komponensek valószínűleg egymástól függetlenül fognak meghibásodni. Ez azt feltételezi, hogy ha teljesen működőképes, akkor minden hardveregység a specifikációnak megfelelően működik. Így kicsi a valószínűsége annak, hogy minden hardveregységben egyszerre hibásodnak meg az összetevők.

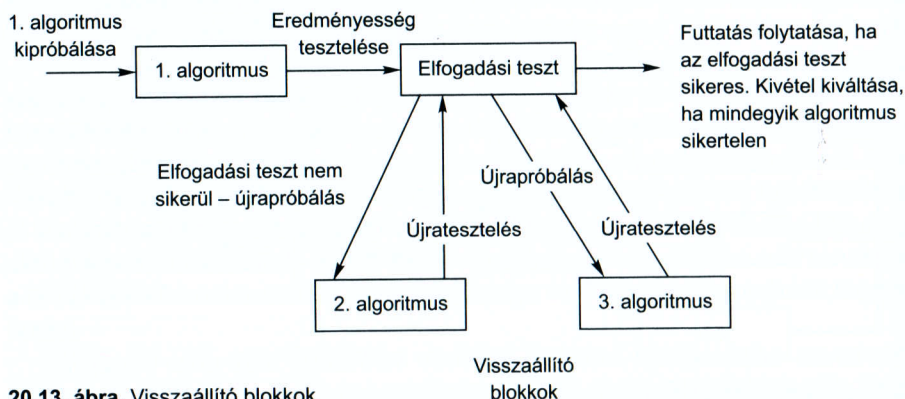
Természetesen előfordulhat minden összetevőnél ugyanaz a tervezési hiba, és így mindegyiknél ugyanaz a (rossz) válasz. Ennek az esélyei úgy csökkenthetők, hogy olyan hardveregységeket használunk, amelyeknek megegyezik a specifikációja, de más-más cég tervezte és gyártotta őket. Feltételezzük, hogy kicsi a valószínűsége annak, hogy különböző csapatok ugyanazon tervezési vagy gyártási hibát követik el.

Ha egy rendszer rendelkezésre állási és megbízhatósági követelményei olyanok, hogy hibátűrő hardverre van szükség, akkor szoftverhiba tűrésére is szükség van. Két, egymáshoz hasonló módja van a hibátűrő szoftver biztosításának (20.12. és 20.13. ábra). Mindkettő azon a hardvermodellen alapul, amelynél a felesleges összetevőket (vagy esetleg a felesleges rendszereket) ki lehet iktatni.





20.12. ábra. Többverziós programozás



20.13. ábra. Visszaállító blokkok

A hibátűrő szoftver kialakításának e két módszere a következő:

1. *Többverziós programozás.* Ugyanazt a specifikációt használva a különböző csapatok számos különböző verzióban kivitelezhetik a szoftverrendszert. Ezeket a verziókat párhuzamosan futtatják különböző számítógépeken. Az outputjaikat egy minősítőrendszer segítségével összehasonlítják, és az ellentmondó, illetve a későn érkező outputokat elutasítják. A rendszer legalább két verziójának elérhetőnek kell lennie, hogy két verziónál megegyező output legyen abban az esetben is, ha egy meghibásodik. Ezt használják a vasúti jelzőrendszereknél, a repülőgépeknél, a reaktorok védelmi rendszerénél. Avizienis (Avizienis, 1985; 1995) ír erről a módszerről.
2. *Visszaállító blokkok.* Ennél a módszernél minden összetevőben van egy teszt, amely ellenőrzi, hogy az adott összetevő sikeresen végrehajtotta-e, amit kellett. Emellett van benne egy alternatív kód is, amely lehetővé teszi, hogy a rendszer lemásolja és megismételje a műveletet, ha a teszt hibát talált. A kivitelezések szándékosan különböző interpretációi ugyanannak a specifikációnak. Egymás után vannak végrehajtva, nem párhuzamosan, így nincs szükség többszörös hardverre. A többverziós programozásnál a kivitelezés különböző lehet, ám nem ritka, hogy a különböző fejlesztési csoportok ugyanazt az al-

goritmust választják a specifikáció kivitelezésénél. Randell (Randell, 1975) és Randell és Xu (Randell és Xu, 1995) írnak a visszaállító blokk módszeréről.

A hibatűró szoftver hibatűréséhez szükséges, hogy a szoftvert egy hibatűró szabályozóval működtessük, amely biztosítja, hogy egy hiba tűrésének lépései megtörténjenek. Ez a szabályozó megvizsgálja az outputokat és összehasonlítja őket. Ha azok eltérnek, akkor valamilyen javító műveleteket indít. Laprie és mások (Laprie és mások, 1995) leírják a hibatűró rendszerek architektúráit.

Mindkét módszer a tervezés és kivitelezés sokféleségének az elvén alapul. Amikor különböző módszereket használnak ugyanazon specifikáció kivitelezésére, ésszerűnek tűnik az a feltételezés, hogy a szoftver különböző verziói nem tartalmazzák ugyanazokat a hibákat, így ugyanolyan sikertelenségek nem valószínűek. A változatosságot többféleképpen el lehet érni:

1. Olyan követelmények előírásával, hogy a tervezésnél különböző módszereket kelljen használni. Például az egyik csapatnak objektumorientált típust kell létrehozni, egy másiknak pedig funkcióorientált típust.
2. Azzal az előírással, hogy minden példány más-más programnyelven legyen megírva. Például egy háromverziós rendszernél az Ada, a C++ és a Java nyelveken kell megírni a szoftver különböző verzióit.
3. Különböző eszközök és fejlesztési környezetek használatának az előírásával.
4. Explicit módon előírva különböző algoritmusok használatát a kivitelezés egyes részeiben. Bár ez csökkentheti a tervezőcsapat szabadságát, és megnehezítheti a teljesítménykövetelményekkel való összeegyeztetését.

Minden csapatnak egy külön specifikációval (a verzióspecifikációval) kell dolgoznia, amely a rendszer követelményeinek specifikációjából származik (Avizienis, 1995). A rendszer funkcionalitásának megadása mellett a verzióspecifikációnak az összehasonlításhoz definiálnia kell, hogy hol keletkezzenek a rendszer-outputok. Minden egyes fejlesztőcsapatnak elszigetelten kell dolgoznia, hogy csökkenjen annak a valószínűsége, hogy a rendszerrel kapcsolatban egyforma téves elképzelések alakuljanak ki.

A többszörös tervezés természetesen növeli a rendszer teljes megbízhatóságát. Számos kísérlet azonban azt sugallja, hogy az a feltevés, mely szerint különböző tervezőcsapatok nem követik el ugyanazokat a hibákat, talán nem mindig érvényes (Knight és Leveson, 1986; Brilliant és mások, 1990; Leveson, 1995). Különböző fejlesztőcsapatok is elkövethetik ugyanazokat a hibákat, a specifikáció hasonló téves értelmezése miatt, vagy mert egymástól függetlenül ugyanazt az algoritmust találják a feladat megoldására. A visszaállító blokkok csökkentik a közös hibák valószínűségét, mert minden egyes visszaállító blokknál más algoritmust használnak.

A hibatűrés e két módszerének az a gyenge pontja, hogy mindkettő azon a feltevésen alapul, hogy a specifikáció helyes. A specifikációs hibákat nem tolerálják. Sok esetben azonban a specifikáció hibás, vagy nem teljes, így a rendszer kiszámíthatatlanul viselkedik. A közös specifikációs hibák valószínűségét oly módon



csökkenthetjük, hogy a verzióspecifikációkat a rendszerhez külön-külön hozzuk létre, és ezeket különböző nyelveken definiáljuk. Így egy fejlesztőcsapat dolgozhat formális specifikációból, egy másik állapotalapú rendszer-specifikációból és a harmadik egy természetes nyelvű specifikációból. Ez segít elkerülni a specifikáció interpretációjából származó hibákat, de nem oldja meg a specifikációs hibák problémáját.

## Kulcsfogalmak

- Egy program üzembiztonsága úgy érhető el, ha elkerüljük a hibákat, illetve észrevevesszük és eltávolítjuk a hibákat még azelőtt, hogy a rendszert kihelyezzünk, továbbá olyan hibatűrő eszközöket adunk hozzá, amelyek lehetővé teszik, hogy a rendszer hiba esetén is működőképes maradjon.
- A megbízható rendszerek fejlesztésénél kulcsfontosságú, hogy mind a szoftverfolyamatokban, mind a szoftverrendszerekben alkalmazzuk a redundanciát és a diverzitást.
- A rendszerben előforduló hibák minimalizálásához nagyon fontos a jól definiált, ismételhető folyamat. A folyamatnak minden lépéshez tartalmaznia kell verifikációs és validációs tevékenységeket, a követelmények meghatározásától a rendszer kivitelezéséig.
- Néhány programozási konstrukció és technika önmagában is hibára hajlamos. Például a goto utasítás, pointerrek, rekurzió, öröklődés és a lebegőpontos számok ilyenek. Ezeket nem szabad üzembiztos rendszerek fejlesztésénél használni.
- Megbízható rendszereknél a kivételek támogatják a hibakezelést. Egy megbízható rendszerben minden kivételt explicit módon kezelni kell.
- A hibatűrésnek négy aspektusa van: hibafelismerés, a veszteség felmérése, hiba-visszaállítás és hibajavítás.
- A többverziós programozás és a visszaállító blokkok olyan alternatív változatai a hibatűrő architektúráknak, amelyek redundáns hardver- és szoftvermásolatokat tartalmaznak. Mindkettő a többszörös tervezésre és redundáns programegységek működésének koordinálására szolgáló hibatűrő szabályozó használatára épül.

## További irodalom

*Software Fault Tolerance Techniques and Implementation.* Átfogó értekezés azon technikákról, amelyek segítenek megvalósítani a hibatűrő architektúrákat és a szoftveres hibatűrést. (Pullum, L. L., 2001, Artech House.)

*Handbook of Software Reliability Engineering.* Ez a gyűjtemény néhány olyan cikket tartalmaz, amelyek a visszaállító blokkokról és a többverziós programozásról szólnak. Szintén tartalmaz egy jó cikket a hibatűrő rendszerek architektúráiról. (Lyu, M. R. (ed.), 1996, McGraw-Hill.)

## Feladatok

- 20.1. Mondjon négy okot arra, hogy a cégek számára szinte sohasem költség-hatékony annak a biztosítása, hogy a szoftverük hibamentes.
- 20.2. Mondjon két példát diverz és redundáns tevékenységekre, amelyek bevonhatók a megbízható szoftverfejlesztési eljárásokba.
- 20.3. Magyarázza el, miért hibára hajlamos struktúra az öröklődés, és miért kell minimalizálni a használatát, amikor egy objektumorientált nyelven fejlesztünk ki kritikus rendszereket.
- 20.4. Fejtse ki, hogy milyen fejlesztési és karbantartási problémák adódhatnak a folyamatosan működő rendszereknél, mint például egy telefonközpont szoftvere. Hogyan alkalmazhatjuk a kivételeket az ilyen típusú rendszerek fejlesztésénél?
- 20.5. Magyarázza el, hogy miért szükséges explicit módon kezelni az összes kivételt egy hibatűrő rendszerben.
- 20.6. Röviden mutasson be előreható és visszaható hiba-helyreállítási stratégiákat. Miért használják a visszaható hiba-helyreállítást gyakrabban, mint az előreható? Adjon meg két példát rendszerek olyan osztályára, ahol a visszaható helyreállítás alkalmazható.
- 20.7. Milyen előfeltételek teljesülése esetén lehet előreható hiba-helyreállítást implementálni egy rendszerhez? Lehetséges-e ez interaktív rendszerekben?
- 20.8. Tervezzon egy absztrakt adattípust vagy objektumosztályt `RobustList` néven, amely egy láncolt listával kapcsolatosan előreható hiba-helyreállítást implementál. Vegyen be olyan műveleteket, amelyek a listát helyesség szempontjából megvizsgálják, és újra felépítik, ha az elromlott. Tételizzük fel, hogy a meghibásodás ellenőrzését a lista szomszédos tagjai előre- és visszamutató referenciáinak kezelésével végezzük.
- 20.9. Egy szoftveralapú irányítórendszer kivitelezésénél milyen körülmények között lehet hibatűrő architektúrát használni? Magyarázza meg, miért ezt a módszert kell használni.
- 20.10. Felmerült az a javaslat, hogy egy (rákos betegek kezelésére használt) sugárterápiás gépet irányító szoftvert többverziós programozással kell kivitelezni. Mi a véleménye erről a javaslatról?
- 20.11. Mondjon két okot, hogy miért hibázhat ugyanolyan módon egy többverziós rendszernél az összes különböző rendszerverzió.
- 20.12. Az itt tárgyalt technikák alkalmazásával létrehozott biztonságos szoftverek nyilván jelentős többletköltségeket igényelnek. Milyen pótköltségek igazolhatók, ha 100 életet mentünk meg a rendszer 15 éves élettartama alatt? Igazolhatók-e ugyanezek a költségek 10 élet megmentése esetén? Mennyit ér egy élet? Az érintett emberek keresőképessége okoz-e különbségeket a megítélésben?