1.tétel

a) Bevezetés az informatikába: Az információ fogalma, útja, mérése. Entrópia. Kódolással kapcsolatos alapfogalmak. Kódolási eljárások, hatásfok. A gépi információ (adat, utasítás) ábrázolása, számábrázolás (fix- és lebegőpontos), karakterkódolás.

Információ fogalma

Alapvető szükséglet. Az adat az információ hordozója, amelynek önmagában nincs jelentése, az az értelmezéstől függ, tehát interakció szükséges: például el kell olvasnunk. Az információ jelentéssel bíró adat, egyértelműen elfogadott definíciója nincs.

Információ útja

adó → kódolás → csatorna (zaj) → dekódolás → vevő

Kódolás-dekódolás

Egy nyelv véges ábécéjéből képzett szavakat kölcsönösen egyértelműen hozzárendel egy másik nyelv meghatározott szavaihoz.

Dekódolás: a fordítottja.

Információ mérése

Információ alakjától és tartalmától függetlenül kell mérni.

Bit: két egyenlően valószínű jel egyikének kiválasztásához tartozó információmennyiség.

Redundancia: élő nyelveknél jó, mert zaj esetén is lehet dekódolni. Terjengősség: 1-H/h ()

Definíció: ha a forrás A_1, A_2, A_3 jeleket bocsát ki, $n=2^m$ és a jelek kibocsátásának valószínűsége egyenlő, vagyis $p_i=1/n$, akkor egy n elemű halmaz konkrét elemének kiválasztásához m kérdésre van szükség, vagyis az 1 jelre jutó információ m. $\log_2 n=\log_2 2^m=m$

Entrópia

A rendszer bizonytalanságának mértéke. Revolveres példa. Valószínűségekkel súlyozott középértékekkel jellemezhető. Jele: H. A valószínűségek összege mindig 1.

Ha egy revolverben 3 golyó van, nehéz megjósolni, hogy meghalunk vagy nem. Ha 5 golyó van benne, akkor könnyebb, mert a halálnak nagyobb a valószínűsége. Persze a halálnak alapból 100% a valószínűsége. Az utóbbi esetben kisebb a bizonytalanság, vagyis az entrópia.

Definíció: adó A_1, A_2, A_n jeleket rendre p_1, p_2, p_n valószínűségekkel bocsátja ki, ahol sum i…n, $p_i=1$. H = - sum i…n $p_i*\log_2 p_i$

Kódolási hatásfok

A kódolt jelek jelenkénti átlagos információjának (entrópia) ($H(A)/h_{átl}$) és a kódábécé lehetséges maximális jelenkénti információjának a hányadosa. Az entrópia és az átlagos hossz hányadosa. %

Átlag hossz

Hány karakteres a jelek kódja és mennyi van belőlük. sum p_i*h_i. Minél kisebb a hossz, annál hatásosabb a kódrendszer.

Kódolási eljárások

Shannon-Fano kódolás

A leggyakrabban előforduló jeleket a lehető legkevesebb adattal reprezentáljuk.

1. A jelek előfordulásának mennyisége szerint csökkenő sorrendbe helyezzük őket.
2. Egyenlő valószínűségeket keresünk a rendezett előfordulások közt, felosztjuk két részre a táblázatot, felül 0, alul 1.
3. Így folytatjuk, amíg nem kapunk végeredményt (egyes jelekre jutó kód).
4. Kiszámíthatjuk az átlagos hosszt, az entrópiát és a hatásfokot.

Huffman kódolás

A leggyakrabban előforduló jeleket a lehető legkevesebb adattal reprezentáljuk.

1. Fentről lefelé csökkenő sorrendbe rakjuk az elemeket előfordulás szerint.
2. Kiválasztjuk a legkisebb előfordulásokat és 0-1-et rendelünk hozzájuk.
3. Összehúzzuk őket vonallal egy új számba, ami a kettő összehúzott összesített valószínűsége.
4. Ezt addig csináljuk, amíg nem jutunk el a jelek számáig.
5. A kódokat visszafelé kell leolvasni.

Adat, utasítás ábrázolása

Adat: objektum vagy egyed tulajdonsága: tanuló lakhelye, neve is adat, de ugyanaz a lakhely tartozhat több tanulóhoz is.

Fixpontos számábrázolás

8, 16, 32, 64 bit hosszú bitsorozatban ábrázoljuk binárisan. Legmagasabb bit: előjel (0+,1-). Összeadás esetén túlcsordulás keletkezhet.

Pozitív számok

16 biten legmagasabb bit 0, aztán a szám 15 biten. Legnagyobb ábrázolható szám: 32767. 1 bájt: 0-255

Negatív számok

A negatív szám abszolútértékének a komplementerét képezzük. Negáljuk a számot: végéről kezdve az első 1-esig ugyanaz, utána minden az ellenkezője. Számítógépben csak összeadás van, így egy + és egy – szám összeadása jelenti a kivonást. 1 bájt: -128-+127

Lebegőpontos számábrázolás

Előjelbit (1), karakterisztika (8), mantissza (23). Szorzatalak. Valós számokat egy közelítő értékkel helyettesítve racionális számokká alakítja.

$N = m * A^k$, ahol N a szám, m a mantissza, 'A' a számrendszer alapja, k a karakterisztika.

Törtre normalizálás: 0,10011, egészre normalizálás: 1,0011

IEEE 754 szabvány példa

-3.75 = -11.11b

Egészre normalizálás (eltolás): -1.111b, az implicitbitet elhagyjuk.

Karakterisztika: eltolás mértéke +127 → 1+127 = 128 → 10000000b

Mantissza: 111b és még 20 darab 0

Negatív szám volt: előjelbit 1 lesz.

-3.75 = 1|10000000|11100000000000000000000

Hexadecimális reprezentáció: C070 0000h

Karakterkódolás

Karakterek megjelenítésre használatos, minden karakterhez egy-egy egyedi számot rendelünk hozzá. Célja, hogy a számítógép is megértse az emberi nyelvet és meg tudja jeleníteni.

Legismertebb kódolási modellek: ASCII, Unicode

Az ASCII 128 szövegkaraktert képez le 0…127 előjel nélküli egész számokra. A kibővített jelkészlet már 256-féle állapotot tárol és különböztet meg.

b) Az informatika logikai alapjai: Elsőrendű logika szintaxisa és szemantikája. Normálformák elsőrendű logikában, KNF-re hozás algoritmusa, skolemizáció. Rezolúció elsőrendű logikában, unifikáció. Lineáris és SLD rezolúció. Prolog alapok.

Bevezetés

Szintaxis: logikai jelek, jelsorozatok, formulák jellemzése, az elsőrendű logika helyes nyelvtana.

Szemantika: a jelek értelmezése, hogyan lehet tartalommal megtölteni őket és összefüggően használni, állításokat megfogalmazni.

Elsőrendű logika szintaxisa

Lehetőség van „van olyan X, amelyre teljesül" és „minden X-re teljesül" típusú állítások megfogalmazására. Ezeket a Létezik (∃) és a Minden (∀) kvantor teszik lehetővé.

Az állítások objektumokra (termekre) vonatkoznak. Nulladrendűben nem lehetett tárgya egy állításnak! A állítások paraméterezhetőek termekkel. Paraméterezhető állítás = predikátum.

Példa:

0. Szeretem a sört → S atom
1. Sz(s), több paraméterrel: Sz(én, s)

Nevesített term: konstans (sör)

Elsőrendű logikában a szintaxisát tekintve használunk predikátumokat, kvantorokat, zárójeleket, változókat, függvényjeleket.

Definíció: azt mondjuk, hogy a $P(t_1, t_2, t_n)$ szimbólum sorozat atomi formula, vagy röviden atom, akkor és csak akkor, ha P egy n paraméteres predikátum szimbólum, azaz egy n paraméteres kijelentés, és t_1, t_2, t_n paraméterek termek, azaz objektumok.

Függvény: termekkel paraméterezett és termet ad vissza: „Mindenki fiatalabb a saját anyjánál": ∀ x F(x, anya(x))

Elsőrendű logika szemantikája

Kötött és szabad változók. Kötött változók átnevezése. Kvantorok hatásköre.

Interpretáció: meghatározza azon értékek körét, melyeket változóink és konstansaink felvehetnek. Ezek már nem csak logikai értéket tárolhatnak, hanem bármilyen objektumot, ezért az interpretáció először definiálja az objektumtartományt, az ún. domaint.

Konstansok interpretálása: mindhez konkrét értéket rendel a domainből.

Predikátumok interpretálása: mire ad vissza igazat, mire hamisat. Összes paraméterezést le kell fedni!

Függvények interpretálása: minden függvényhez megadunk egy olyan visszatérési érték táblázatot, ami a domainnek megfelelő összes lehetséges paraméterezést lefedi.

Példa: minden hallgató teljesítette a logika vizsgát. Az igazságérték függ attól, hogy melyik egyetem hallgatóiról van szó (melyik egyetem hallgatói a domain részei).

Szemantikai tulajdonságok

Bizonyításuk sokkal nehezebb.

Logikai törvény: egy formula tautológia, ha minden interpretációban igaz.

Logikai ellentmondás (kontradikció): egy formula ellentmondásos, ha minden interpretációban hamis.

Kielégíthetőség: egy formula kielégíthető, ha legalább egy interpretációban igaz.

Logikai következmény: A P1…Pk formulák logikai következménye C, ha P1&Pk&-C-ből levezethető az üres klóz.

Formula kiértékelés

Formula: **[∀x (⅂P(x) V ∃y Q(x , y)]**
Kérdés ennek a formulának mi az értéke?

Ehhez meg kell adnun a domaint azaz h a formulában a változók, (konstansok)
D = {1, 2, 3, 4} milyen értékeket vehetnek fel.

Ehhez meg kell adnunk a formulában szereplő predikátumok interpretációját

Predikátum	Interpretáció
[P(x)] =	1, ha x páros 0, egyébként
[Q(x,y)] =	1, ha x*y = 4 0, egyébként

Akkor igaz a formulánk a fenti domain elemekre és predikátum interpretációkra, ha
[∀x (⅂P(x) V ∃y Q(x , y)] = 1 teljesül minden domainbeli elemre

Mivel van bennek egy vagy (diszjunkció) ezért érdemes esetszétválasztást alkalmazni.
A ⅂P(x) -> vagyis nem páros és páros elemekre

1 x páratlan ⅂P(x) = 1 (hisz ha valami nem páros akkor páratlan tehát ez a tagrész igaz)
Mivel diszjunkció van és ha annak egyik fele teljesül az egész teljesül ezért ha x páratlan a formulánk igaz lesz

2 x páros ⅂P(x) = 0 Tehát a másik tagrész dönti majd el
∃y Q(x , y) Ha ez a tagrész igaz mindre akkor a formulánk igaz.
Kérdés: ∃y (eleme)D[Q(x , y)∃y Q(x , y)] = 1
x = 2 y = 2 Tehát ez igaz
x = 4 y = 1 Ez is igaz

Tehát minden esetben van olyan y elem ami egy páros x elemmel 4-et ad szorzatként.
Ezért kijelenthető, hogy a formula igaz!

Normálformák elsőrendű logikában

De Morgan:

- ⅂ ∀ x A ⇔ ∃ x ⅂ ∀ (nem minden x A, létezik olyan x, ami nem A)
- ⅂ ∃ x A ⇔ ∀ x ⅂ ∀ (nem létezik olyan, aki A, minden x nem A)

Kvantorkiemelés: O x A o B ⇔ O x (A o B)

Zárt formula: nincs benne szabad változó

KNF-re hozás algoritmusa

1. Változótiszta alakra hozás

-∃y (∃x P(x, y) => Q(x) & Q(y))
-∃y (∃z P(z, y) => Q(x) & Q(y))

2. Implikáció eltüntetése
   -∃y (-∃z P(z, y) | (Q(x) & Q(y)))
3. Negáció bevitele (de Morgan azonosságok)
   -∃y (∀z -P(z, y) | (Q(x) & Q(y)))
   ∀y -(∀z -P(z, y) | (Q(x) & Q(y)))
   ∀y (-∀z P(z, y) & (-Q(x) | -Q(y)))
   ∀y (∃z -P(z, y) & (-Q(x) | -Q(y)))
4. Prenexizálás
   ∀y∃z (-P(z, y) & (-Q(x) | -Q(y)))
5. Skolemizálás
   ∀y (-P(f(y), y) & (-Q(x) | -Q(y)))
6. KNF-re hozás

Prenexizálás: kvantorkiemelés, ha a kvantor által kötött változó nem szerepel a másik tagban paraméterként

Skolemizálás: ∃ eltüntetése, ha van előtte ∀, akkor függvénnyel, aminek a paramétere a ∀ által kötött változó, ha nincs előtte ∀, akkor konstanssal.

Rezolúció elsőrendű logikában

Nulladrendű véges levezetést garantál, elsőrendűben előfordulhat, hogy végtelen lesz.

Cáfoló eljárás, ahol a bizonyítandó állítás tagadásából indulunk ki és bizonyítjuk, hogy levezethető belőle az üres klóz, azaz logikai ellentmondás (UNSAT). A tagadásról bizonyítjuk, hogy kielégíthetetlen.

Az A formula tautológia, ha -A-ból levezethető az üres klóz.

A P1…Pk formulák logikai következménye C, ha P1&Pk&-C-ből levezethető az üres klóz.

Tekintsük $C_1 = P \vee C'_1$ és $C_2 = \neg P \vee C'_2$ klózokat, ahol P ítéletváltozó!

A $C'_1 \vee C'_2$ klózt a C_1 és C_2 **rezolvensének** nevezzük. Az egymással ellentéteseket vesszük ki belőle, és ami marad az a rezolvens.

1. Válasszunk két eddig még nem rezolvált klózt: C_1 és C_2
2. Állítsuk elő C_1 és C_2 **rezolvensét** (ha van) és adjuk a klózhalmazhoz!
3. Ha az **üres klóz** (□) előáll, akkor a klózhalmaz UNSAT. Állj!
4. Vissza az 1. pontra!

Rezolúció algoritmusa:

1. Válasszunk két eddig még nem rezolvált klózt: C1 és C2
2. Állítsuk elő C1 és C2 rezolvensét (ha van) és adjuk a klózhalmazhoz
3. Ha az üres klóz előáll, akkor a klózhalmaz UNSAT. Állj!
4. Vissza az 1. pontra.

Unifikáció

Rezolvens képzésekor a két literál egyforma kell legyen karakterről karakterre. Ezt elsőrendűben nehéz összehozni, ezért lehet illeszteni konstansok segítségével.

Ugyanolyan változót nem helyettesíthetünk olyan termmé, amiben szerepel: x → f(x), occur check, ilyet nem lehet

Különböző függvény szimbólumok nem unifikálhatóak egymással: g(x) → f(a) nem lehet, mert g != f

∀x(x, c) és ∀y(d, y) esetén:

- x és d közül legalább az egyik változó kell legyen
- d és c konstansok
- ha x és d konstansok, akkor nem lehet unifikálni
- x: amit unifikálok, s: amivel helyettesítem

x → d: x-et d-vel helyettesítem

y → c: y-t c-vel helyetessítem

x

Q(*x*, c)

Q(*d*, y)

s

Q(d, c), Q(d, y), majd Q(d, c), Q(d, c) és így már unifikálható

Lineáris rezolúció

A következő lépés mindig épül az aktuálisra, vagyis az előző lépésben előállított rezolvensre.

Bármely kielégíthetetlen klózhalmazból garantáltan le lehet vezetni az üres klózt.

SLD rezolúció

Célirányosabb a lineárisnál is.

A mellékklózoknak minden esetben az input klózhalmazból kell érkezniük. SLD-vel nem biztos, hogy levezethető az üres klóz. Az egyik klóz legalább az eredetiek közül való. Horn-klózok esetén működik.

Horn-klóz: legfeljebb 1 + literált tartalmaz (nem negált). Ha valamelyik nem ilyen, akkor az eredeti állítást kell átfogalmazni.

Prolog

Első logikai programozási nyelv. Egy megadott logikai formuláról képes eldönteni, hogy logikai következménye-e formulák egy adott halmazának. A formulák és a célformula a bemenet, a kimenet pedig az a válasz, hogy logikai következménye-e vagy nem.

2. tétel

a) Magasszintű programozási nyelvek I: Tömbök és listák. Rekordok, class és struct. Felsorolásos típusok. Metódusok. Paraméterátadás módjai, változó paraméterszám. Programozási nyelvek fordítási és futtatási megoldásai. .NET keretrendszer felépítése, más programozási nyelvek és keretrendszerek.

Tömbök és listák

Tömb

- elemei indexeléssel közvetlenül, tetszőlegesen érhetőek el
- dimenziószámmal rendelkeznek. Vektor 1, mátrix 2 dimenziós, de több is lehet
- hossza statikus (előre kell foglalni a memóriában fix helyet)
- memóriafoglalás folytonos, az elemek egymás mellett vannak

Lista

- szekvenciális elérésű, de indexeléssel is lehet
- hossza dinamikus (futási időben változik)
- listák listája: belseje és külseje is változhat: tanulók listájában tanulók jegyei

Rekordok

- összetartozó adatokat egy csoportban tárol
- összetett (több fajta adat lehet benne)
- inhomogén (tárolt adatok típusa eltérő lehet)
- random elérésű
- fix méretű (deklaráláskor kell megadni az adatok számát és mennyiségét)
- folytonos tárolású (memóriában egymás után következnek a tárolt értékek)

A rekord egy osztály példánya. Mezői lehetnek, értéket adhatunk neki stb. A rekordok referencia objektumok, ami azt jelenti, hogy a példány a heap memóriában fog allokálni és a stackben csak egy mutató jön létre, ami erre az objektumra fog rámutatni.

Struct

- érték típusú
- nem lehet paraméterezetlen konstruktora vagy destruktora
- példányosítható „new” nélkül, mert értéktípus és egyből lefoglal helyet a memóriában
- kisebb logikájú, rövidebb életű modellekhez használják
- nincs öröklődés
- nem lehet abstract
- tud implementálni interface-t
- lehet az értéke null
- stackben allokál
- a memória felhasználás pontosan a típus méretével egyezik meg
- Mivel érték típus, ezért paraméter átadásnál vagy értékadásnál lemásolódik (int a = 5, int b = a után ha **b** értékét megváltoztatjuk, akkor **a** értéke nem változik

Class

- referencia típusú

- komplexebb viselkedés leírásához használjuk
- egységbe zárja az adatokat és a rajtuk végzett műveleteket
- int a = 5, int b = a esetén a referencia másolódik nem az érték, ezért ha b módosul, a is módosul, mert ugyanoda mutatnak
- csak „new”-val foglalható neki memóriaterület

Felsorolás: enum

Konstans értékek adategységét jelenti. Például ételek. Csak eljárásokon kívül, de osztályon belül szabad deklarálni. Kordában tartja a usert, hogy mit írhat be, mert meg vannak határozva az értékek, amikből választhat. Az értékek readonly-k.

A felsorolás elemeihez alapértelmezetten egy index társul, de a könnyebb olvashatóság miatt neveket látunk. Int-té konvertálva lekérhetjük az indexet is.

Példa: ConsoleColor, ConsoleKey C#

Érdekesség: Swiftben az enum típusba lehet property-ket és metódusokat tenni

Metódusok

Logikai egységekre szabdalják a programot. Többször felhasználhatóak, mert hívni lehet őket. Visszatéréseket és hívásokat a stack kezeli. LIFO

Mindkettőnek lehet tetszőleges számú paramétere.

Eljárás

- visszatérési érték: void
- lehet használni a return-t kilépésre, de ez elég csúnya

Függvény

- visszatérési érték: programozótól függ

Paraméterátadás módjai

Formális paraméterlista: metódus kifejtésénél a fejlécben vannak, általánosak, érték nélkül

Aktuális paraméterlista: metódus hívásánál értéket adunk a fejlécben felsoroltaknak

Kimenő paraméterátadás: formális paraméterlista egyik paraméterét az „out” kulcsszóval látjuk el, és a meghívás előtt létrehozzuk azt a változót (csak deklarálni), amibe belerakjuk a kimenő paraméter értékét.

Érték szerinti paraméterátadás

Primitív típusok esetén, mert azok a stackben vannak tárolva. Amikor átadjuk őket a meghívott függvénynek, akkor egy másolat keletkezik róla, emiatt akármit csinálunk vele a függvényben, az nem lesz kihatással a változóra.

Referencia szerinti paraméterátadás

Az objektumok heapben vannak tárolva, stackben csak egy mutató van, ami erre mutat. Amikor átadjuk őket a meghívott függvénynek, akkor a referenciát adjuk át, ami a függvényben ugyanarra mutat, mint a változóban. Ezért a függvényben történő dolgok

kihatással lesznek az objektumra. Primitív típusú változókat is használhatunk referencia szerint a ref/in kulcsszavakkal.

Változó paraméterszám

Ha változhat egy függvény paraméterszáma, akkor ugyanazt a nevet több függvénynek is választhatjuk, a fordító el tudja dönteni, hogy melyiket kell használnia. Hátránya, hogy minden változatra meg kell írni ugyanazt a függvényt. Ezért lehet adatszerkezetet bekérni (amit pl. saját magunk hozunk létre a szükséges paraméterekből). Így nem 4-5 paraméter lesz, hanem csak 1-2.

Object: minden más típus őse, úgyhogy azt is csinálhatjuk, hogy objecteket kérünk be, vagy object listát/tömböt.

params: akárhány ugyanolyan típusú paraméter lehet. Deklarálás: params int[] list, és ezután további paraméterek nem megengedettek

Programozási nyelvek fordítási és futtatási megoldásai

Amint a lefordult kód elkészült, háromféleképpen futtahatjuk

Direkt futtatás

A generált kód az adott mikroprocesszor gépi kódú utasításait tartalmazza. Az utasítássorozatot az OS átadja a mikroprocesszornak és megadja a program kezdőpontját. Innentől indul a végrehajtás, követi az ugrásokat, írja, olvassa a memória hivatkozott területeit anélkül, hogy tudná mit csinál a program az adott ponton. A CPU megbízik a programkódban és végrehajtja. Ezért a sebesség nagy, de nem lehet eldönteni, hogy az utasítás a program feladata szempontjából helyes-e, szükséges-e, hibás-e. Van compiler, ami a szöveges forráskódot gépi kódra fordítja, amiből a linker az OS-ben működőképes programot készít.

Interpreteres futtatás

A fordítóprogram nem generál közvetlenül végrehajtható gépi kódú utasításokat, hanem egy köztes kódot, ahol az eredeti programozási nyelv utasításai vannak számkódokká fordítva, a paraméterei már feldolgozott, egyszerűsített formában kódoltak. Ezt a tárgykódot futtatja az interpreter, ami futás közben utasításonként elemzi és hajtja végre az utasításokat. A futtató rendszer végrehajtás előtt ellenőrzi, korrigálja a kódot (pl. változóhivatkozások). Megoldható az is, hogy a változókat nem kell deklarálni, hanem kitalálja magától, hogy milyen változó kell és csak akkor hozza létre. A típussal is megoldható, hogy csak használat előtt derüljön ki, vagy akár változzon is.

Az interpreteres rendszer hátránya, hogy lassú, és a generált tárgykód az adott nyelv lenyomatát tartalmazza, ezért nem univerzális. Tárgykód csak a memóriában van, addig van, amíg fut a program.

Egy adott nyelv interpretere nem képes futtatni más interpreteres nyelv fordítóprogramja által generált tárgykódot.

Virtuális futtatás

A szintaktikai és szemantikai elemzés és a fordítás elkülönült fázis, így a programhiba már ekkor kiderül (futtatás előtt). A virtuális futtató rendszer bemenete a lefordított tárgykód. A

fordító egy virtuális CPU virtuális gépi kódjára generál programot. Ez eltérhet a gépi kódtól, ismerhet több típust, tartalmazhat magasabb szintű utasításokat. A generált kód már nem feltétlenül hasonlít az eredeti nyelv szintaktikájára, ezért univerzális.

A virtuális futtatás előnye, hogy a kód más CPU-n is futtatható lesz, ha arra a CPU-ra is létezik a futtató rendszer.

A linker futtatható állományt generál, de ezt nem lehet direkt módon futtatni. Ehelyett az elején lévő gépi kódú rutin betölti a virtuális futtató motort és a program maradék részét ez futtatja.

Legismertebb ilyen nyelv a JAVA, aminek a futtató rendszere a Java Virtual Machine (JVM).

.NET keretrendszer

Egy virtuális futtatórendszerrel felszerelt programozási platform. Tartalmaz egy virtuális gépi kódú programozási nyelvet, amelyre a lefordított programokat a Framework képes futtatni. Ezt a fordítást erre a célra fejlesztett fordítóprogramok végzik. OOP-re épül. Lehet Windows interface-szel rendelkező programokat, WEB alkalmazásokat is fejleszteni. Programozási nyelve a C#.

b) Adatszerkezetek és algoritmusok: Algoritmus fogalma, tulajdonságai, megadásának módjai, eszközei, a strukturált algoritmus szerkezete. Programozási tételek: sorozathoz elemi értéket, sorozathoz sorozatot és több sorozathoz egy sorozatot rendelő (kiválogatások, rendező algoritmusok és hatékonyságuk, visszalépéses keresés) algoritmusok. Elemi algoritmusok alkalmazása, a halmaz adatszerkezet különböző konstrukciói (elemek rendezetlen, rendezett sorozatban, karakterisztikus függvény szerepe).

Algoritmus fogalma

Adott szinten elemi lépések sorozata, amely adott probléma megoldását célozza.

Algoritmus tulajdonságai

- univerzális: nem 1 adott problémára jó, hanem minden hasonló problémára
- egyértelmű: minden lépés végrehajtása után egyértelműen megmondható, hogy mi lesz a következő lépés
- determinisztikus: a bemenet mindig meghatározza a kimenetet. Ugyanarra a bemenetre ugyanazt a kimenetet adja

Algoritmus hatékonysága

- végrehajtási idő: adatok időigénye, algoritmus időigénye; függ az adatok típusától és méretétől
- tárigény: adatok/kód tárigénye; célszerű kisebb tárigényű típusokat választani
- algoritmus bonyolultsága: minden vezérlési szerkezethez hozzárendeljük 2 annyiadik hatványát, ahány másik vezérlési szerkezet tartalmazza őt. Ha a program alprogramban van megvalósítva +1-et adunk hozzá.

Algoritmus megadásának módjai

Az algoritmusleíró eszközök célja a feladatok megoldásának leírása programozási nyelvtől függetlenül az eltérő szintaxisok stb. miatt. Ismert eszközök: pszeudokód, folyamatábra, gráf.

Strukturált algoritmus szerkezete

- Csomópont: végre kell hajtani a benne lévő utasítást /Téglalap/
- Döntéscsomópont: végre kell hajtani a benne lévő eldöntendő utasítást. Ennek eredménye alapján az i vagy h ágon kell tovább haladni /Rombusz/
- Gyűjtőcsomópont: nem változtatja meg a program állapotát /Pont/

Strukturált szerkezetnek nevezzük azt a kódot, amely csak a három algoritmikus szerkezetet (szekvencia, elágazás, ciklus) tartalmazza.

Hátrány: terjedelmes, javítása nehézkes, áttekinthetetlen lesz egy idő után.

Programozási tételek

Sorozathoz elemi értéket

Sorozathoz elemi értéket rendelnek a következő algoritmusok: összegzés, megszámlálás, minÉrték, minHely, eldöntés, lineáris keresés, kiválasztás, strázsás keresés, bináris keresés

Összegzés: egy sorozat elemeinek értékét összeadja és ezt visszaadja

Megszámlálás: megszámlálja hány elemű egy sorozat és ezt visszaadja

minÉrték: egy sorozat elemei közül visszaadja a legkisebb értékűt

minHely: egy sorozat elemei közül visszaadja a legkisebb értékű elem indexét

Eldöntés: visszaadja, hogy egy sorozatban létezik-e a feltételnek megfelelő elem vagy sem

Lineáris keresés: visszaadja, hogy egy sorozatban létezik-e a feltételnek megfelelő elem vagy sem. Ha igen, visszaadja az utolsó ilyen elem indexét (alapesetben). Legrosszabb esetben N lépést igényel, vagyis a hatékonysága lineáris.

Kiválasztás: egy olyan sorozatból, amiben biztosan szerepel a feltételnek megfelelő elem visszaadja az utolsó ilyen elem indexét (out)

Strázsás keresés: egy olyan sorozatot, amelyben nem biztosan szerepel a feltételnek megfelelő elem olyanná teszünk, amiben biztosan szerepel az utolsó+1. helyen. Ha az eredeti sorozatban szerepelt a feltételnek megfelelő elem, visszatér igaz értékkel és az indexszel. Ha nem szerepelt, akkor visszatér hamis értékkel és a strázsaelem indexével.

Bináris keresés: egy rendezett sorozatban az első és utolsó index alapján meghatározzuk az aktuális középső elemet, és ehhez viszonyítva keressük az elemet. Ha a keresett elem kisebb a középsőnél, akkor a baloldali félben lesz, az utolsó indexet beállítjuk K-1-re. Ezt ismételjük, amíg meg nem találjuk az elemet. Legrosszabb esetben $\log_2 N$ lépést igényel, vagyis a hatékonysága logaritmikus.

Sorozathoz sorozatot

Sorozathoz sorozatot rendel a kiválogatás.

Kiválogatás: egy sorozatból kiválasztja a feltételnek megfelelő elemeket és ezeket egy új sorozatba rendezi, majd visszaadja hány elemű az új sorozat

Több sorozathoz egy sorozatot

Több sorozathoz egy sorozatot rendel az összefuttatás, visszalépéses keresés.

Összefuttatás: két rendezett sorozat elemeiből kialakít egy harmadik rendezett sorozatot.

Rendező algoritmusok és hatékonyságuk

Rendezés közvetlen kiválasztással: tárigény N+1, azaz O(N), összehasonlítások száma N*(N-1)/2, azaz $O(N^2)$ (minden i. elemet n-i darab elemmel kell összehasonlítanunk), mozgatások száma 0-3*N*(N-1)/2, azaz $O(N^2)$, azaz négyzetes

Rendezés minimum kiválasztással: tárigény N+1, azaz O(N), összehasonlítások száma N*(N-1)/2, azaz $O(N^2)$, mozgatások száma 3*(N-1)/2, azaz O(N)

Érdekesség: az előző, az előző előttinek a javítása, de előfordulhat, hogy rosszabb lesz a mozgatások száma, mert a külső ciklusban mindenképpen cserélünk. Ha megvizsgálnánk, hogy érdemes-e cserélni, az meg növelné a futási időt.

Buborékrendezés: tárigény N+1, azaz O(N), összehasonlítások száma N*(N-1)/2, azaz $O(N^2)$, mozgatások száma 0-3*N*(N-1)/2, azaz $O(N^2)$. Minden elemet összehasonlít a rákövetkezőjével, és ha nem megfelelő a sorrend, akkor megcseréli őket. Ez n-1 összehasonlítás és ugyanennyi csere. Aztán az utolsó elem kihagyásával n-1 elemmel ismételjük meg a műveletet, és n-2 összehasonlítást és legfeljebb ugyanennyi cserét végzünk el.

BuborékJ1: ---

BuborékJ2: tárigény N+1, azaz O(N), összehasonlítások száma N-1 - N*(N-1)/2, azaz $O(N^2)$, mozgatások száma 0-3*N*(N-1)/2, azaz $O(N^2)$

Beszúró rendezés: tárigény N+1, azaz O(N), összehasonlítások száma N-1 - N*(N-1)/2, azaz $O(N^2)$, mozgatások száma 0-3*N*(N-1)/2, azaz $O(N^2)$

Visszalépéses keresés

Olyan problémáknál jó, ahol analitikus megoldás nem igazán van, így szisztematikusan próbálgatunk. Ezen kívül kis bemenetek esetén, vagy ha olyan többlettudásunk van, amivel csökkenthető a futási idő (nem kell minden ágat bejárni). Exponenciális lépésszámú.

Adott n darab véges, de nem üres sorozat, amihez hozzá kell rendelnünk egy n-elemű sorozatot úgy, hogy ennek a sorozatnak az i. elemét az i. adott sorozat elemei közül választhatjuk, de az elem megválasztását befolyásolja az, hogy a többi sorozatból korábban mely elemeket választottuk.

8 királynő probléma példa

Előfordulhat, hogy az i. számára nem találunk olyan helyet, ahol nincs ütésben. Ekkor az i-1. számára kell olyan másik helyet keresni, ahol nincs ütésben. Ha ilyen nincs, addig lépegetünk vissza i mentén, amíg valamelyik sorozat eleme elhelyezhető így. Ekkor ismét haladhatunk tovább előre.

Halmaz

Bizonyos egymástól különböző dolgok összessége.

Komplementer: A komplementere azon elemek halmaza, amelyek B-ben igen, de A-ban nincsenek benne. NEM

Unió: összes elem, VAGY

Metszet: közös elemek, ÉS

Különbség: A/B, minden, ami A-nak eleme, de B-nek nem. NEM ÉS

Diszjunkt halmaz: metszetük az üres halmaz

Algoritmusok esetén ki kell jelölnünk egy U-val jelölt Univerzumot, ami minden szóba jöhető elemet tartalmaz. Egy tetszőleges A halmaz, mely részhalmaza U-nak, minden eleme megtalálható az U-ban, ezért nem kell újból eltárolni az elemeket, hanem egy logikai sorozattal reprezentáljuk az A halmazt.

Karakterisztikus függvény szerepe

Az a függvény, amely megmondja, hogy az alaphalmaz i. eleme szerepel-e annak egy A részhalmazában.

3. tétel

a) Adatbázisrendszerek I.: Hierarchikus, hálós és relációs modellek. Kulcsok a relációs modellben. Kapcsolatok: egy-egy, egy-sok, sok-sok. Anomáliák. Funkcionális függőségek, tranzitivitás. Normálformák

Hierarchikus modell

Az egyedeket a köztük lévő kapcsolat alapján hierarchiába rendezi. Egy-egy és egy-sok kapcsolatok megvalósítására alkalmas. Az alul lévőknek egyetlen ősük lehet, de a felül lévőknek több gyereke lehet, azaz egy fára hasonlít, így gráffal adjuk meg. Az egyed a csomópont, a kapcsolat az él. Csak azokat az egyedeket kötjük össze, amik között van kapcsolat. Több független fából is állhat az adatbázis. Példa: családfa, főnök-beosztott viszonyok.

Hálós modell

A hierarchikus modell kiterjesztése. Tetszőleges gráf lehet, nem csak fa. Egy egyedtípusnak több őse is lehet, kezelhető a sok-sk kapcsolat, amiket egy-sok kapcsolatokra bontanak fel.

Halmaztípus: kapcsolatok leírásának reprezentálására szolgál, jellemzői: név, tulajdonos, tag.

A tulajdonos és a tag egyedtípusok, amelyek egyértékű és többértékű attribútumokat is tartalmazhatnak. A halmaztípus az egy-sok kapcsolat megvalósítására való. Megadása szokásos grafikus formában, ahol nyíllal kötjük össze őket.

Relációs modell

Matematikai alapja a reláció, ami kétdimenziós táblázatos halmazt jelent. Ez alapján készült el az SQL nyelv is, melynek használatával lekérdezéseket és adatbázis kezelési műveleteket lehet végrehajtani

Tulajdonságai:

- minden táblázathoz tartozik egy azonosító
- a sorok és oszlopok metszetében lévő adatok egyértelműek, nem tartalmazhat adatcsoportot, neve rekord
- egy oszlopban található adatok azonos jellegűek
- minden oszlopnak egyedi neve van
- minden sorban ugyanannyi adat található (ugyanannyi oszlop)
- nem lehet két azonos sor
- sorok és oszlopok sorrendje lényegtelen

Egy adatbázis több összekapcsolt relációból áll. Egy reláció egy tábla, melynek sora a rekord, oszlopa az attribútum.

Kulcsok

Elsődleges kulcs: minden relációban kell legyen. Egyértelműen megkülönbözteti egymástól a rekordokat.

Egyszerű kulcs: a kulcs egyetlen attribútum

Összetett kulcs: a kulcs több attribútum

Elsődleges attribútum: a kulcshoz tartozik

Másodlagos attribútum: nem tartozik a kulcshoz

Idegen kulcs: olyan attribútum a relációban, ami egy másik relációban kulcs

Funkcionális függőség: az egyik attribútum meghatározza a másikat (szig. szám → ember neme)

Szuperkulcs: olyan attribútum halmaz, amelytől az összes többi attribútum funkcionálisan függ.

Kapcsolatok

Egy-egy: egy egyedhez pontosan egy másik egyed tartozik. Az egyik egyednél van egy kulcs attribútum, amelynek értékeit felveszi a másik és így kapcsolódnak egymáshoz.

Egy-sok: egy egyedhez több másik egyed tartozik, de a másik csoport minden egyedéhez pontosan egyet társítunk. Az egyik egyednél van egy kulcs attribútum, amelynek értékeit ezen az egyeden kívül a másik csoport összes egyede felveszi. Minden oktatónak több hallgatója lehet, de egy hallgató csak egy oktatónál készíthető el a szakdolgozatát.

Sok-sok: egy egyed több másikkal áll relációban és ez fordítva is igaz. Létre kell hozni egy speciális relációt (kapcsolótábla), amely két attribútumból áll. Az egyik az egyed, a másik a másik egyed kulcsának egy lehetséges értékét veszi fel. Egyetemi kurzusfelvétel. Egy hallgató több kurzusra jelentkezik és egy kurzust többen is felvehetnek.

Anomáliák

Beszúrási: egy rekord beszúrása egy másik, hozzá logikailag nem kapcsolód adatcsoport beszúrását követeli meg. Az is lehet, hogy úgy kell beszúrni egy relációba, hogy az egy másikban is megjelenik, amit nem akarunk. Ilyenkor a két relációt logikailag el kell választani egymástól.

Módosítási: egy rekord módosítása több helyen történő módosítást igényel. Felesleges logikai összetartozásra utal két reláció között.

Törlési: egy rekord törlésekor egy másik, hozzá logikailag nem kapcsolódó rekord is törlődik. Logikailag nem összetartozó adatokat szerveztünk egy struktúrába.

Tranzitivitás: A függ B-től, B függ C-től, akkor A is függ C-től.

Normálformák

1NF: ha a reláció nem tartalmaz adatcsoportot. Az oszlopok és sorok metszetében csak egy darab elemi adat található.

2NF: ha 1NF és a kulcstól nem függ egyetlen más oszlop sem, ami nem a kulcs része.

3NF: ha 2NF és egyetlen másodlagos attribútum sem függ tranzitíven a kulcstól.

Boyce-Codd normálforma (BCNF): ha létezik A → B függés A és B attribútum halmazok esetén, akkor A a reláció szuperkulcsa.

b) Adatszerkezetek és algoritmusok: Algoritmus hatékonyságát befolyásoló algoritmizálási és adatkonstrukciós szempontok. Dinamikus adatszerkezetek (verem, sor, lista, hash-tábla) kezelésének modellje, a kapcsolódó adatszerkezetek implementációi, műveletei és alkalmazásai. Kereső algoritmusok és hatékonyságuk. Programozási tételek értelmezése különböző homogén adatszerkezetek esetében. Rekurzió: rekurzió és iteráció, a fa adatszerkezet és műveletei.

Algoritmus hatékonysága

- végrehajtási idő: adatok időigénye, algoritmus időigénye; függ az adatok típusától és méretétől
- tárigény: adatok/kód tárigénye; célszerű kisebb tárigényű típusokat választani
- algoritmus bonyolultsága: minden vezérlési szerkezethez hozzárendeljük 2 annyiadik hatványát, ahány másik vezérlési szerkezet tartalmazza őt. Ha a program alprogramban van megvalósítva +1-et adunk hozzá.

Dinamikus adatszerkezetek

Verem

Homogén, szekvenciális adatszerkezet, ahol a tárolt elemekhez csak az egyik végén férünk hozzá. Bővíthetünk (adatot helyezünk a verem tetejére) vagy kiolvashatunk (kivesszük a verem tetején lévő adatot). A leghamarabb behelyezett adathoz férünk hozzá legkésőbb. LIFO

VeremMéret, Elemtípus, SP

SP=0, üres verem

VeremÜres: üres-e a verem, ha SP<1

VeremTeli: teli-e a verem, ha SP=VeremMéret

push: VeremTeli? Nem → SP++, Verem[Verem.SP]. elem legyen a beillesztendő elem

pop: VeremÜres? Nem → kimenő paraméter megkapja a Verem[Verem.SP]. elemet, SP—

Veremhasználat példák

- Assembly szubrutin hívásánál regiszterek, flag és IP elmentése
- Posztfixes kiértékelés: balról jobbra haladva olvasunk. Operandus: push. Operátor: 2 operandus pop. Az elsőként kiolvasott operandus lesz a 2., a másodikként kiolvasott pedig az első. Műveletvégzés. Ha jól csináltuk, az érték a verem tetejéről kiolvasható. Ha a popolt operandusok közé rakjuk az operátort, akkor infixes lesz.

Sor

Homogén, szekvenciális adatszerkezet, ahol a tárolt elemekhez két oldalon férünk hozzá. Egyik oldalon írunk, másikon olvasunk. A leghamarabb behelyezett adathoz férünk hozzá leghamarabb. FIFO

Műveletek: elem beillesztése a sor végére, elem kinyerése a sor elejéről, elemszám kinyerés, ürítés, elem megtekintése eltávolítás nélkül.

Lista

Homogén, dinamikus adatszerkezet, reprezentációja szétszórt, nem folytonosan vannak tárolva a memóriában az elemek. Elérésükre mutatókat használunk, amelyek az adott elemre mutatnak.

Láncolt lista

egyirányban láncolt lista: adaton kívül tartalmaz egy olyan mutatót, amely a következő elemre mutat: egyféleképpen járható be a lista. Az utolsó elem Köv mutatója a végjelre mutat.

Végjel: nem mutat sehová a mutató, a lista vége

Listafej: az első elemre mutat, a lista üres, ha a listafej értéke a végjel

kétirányban láncolt lista: Köv mellett van egy Előző mutató is. Az első elem előzője, és az utolsó elem következője a végjel.

Listaműveletek

```
FÜGGVÉNY ListábaElől(Fej: MUTATÓTIP, Adat: ÉRTÉKTIP): LOGIKAI;
VÁLTOZÓK P: MUTATÓTIP;
ALGORITMUS
        HA Lefoglal(P) AKKOR
                Elem[P].Érték <- Adat;
                Elem[P].Köv <- Fej;
                Fej <- P;
                ListábaElől <- IGAZ;
        KÜLÖNBEN
                ListábaElől <- HAMIS;
        HVÉGE;
FVÉGE;

FÜGGVÉNY ListaTörölElől(Fej: MUTATÓTIP): LOGIKAI;
VÁLTOZÓK P: MUTATÓTIP;
ALGORITMUS
        HA Fej != Végjel AKKOR
                P <- Fej; Fej <- Elem[Fej].Köv;
                Felszabadít(P);
                ListaTörölElől <- IGAZ;
        KÜLÖNBEN     ListaTörölElől <- HAMIS;
         HVÉGE;
FVÉGE;
```

Elem beszúrása megadott elem után:

```
FÜGGVÉNY ListábaElemUtán(EzUtán: MUTATÓTIP, Adat: ÉRTÉKTIP): LOGIKAI;
VÁLTOZÓK
P: MUTATÓTIP;
ALGORITMUS
        HA ( EzUtán != Végjel ) ÉS Lefoglal(P) AKKOR
                Elem[P].Érték <- Adat;
                Elem[P].Köv <- Elem[EzUtán].Köv;
```

```
            Elem[EzUtán].Köv <- P;
            ListábaElemUtán <- IGAZ;
        KÜLÖNBEN
            ListábaElemUtán <- HAMIS;
        HVÉGE;
FVÉGE;
```

Hash-tábla

Asszociatív tömb: kulcs-érték párokat tartalmaz, ahol egy kulcshoz pontosan egy érték tartozik.

Hash függvény állapítja meg, hogy melyik kulcshoz milyen érték tartozik, így implementál egy asszociatív tömböt. A hash függvény segítségével a kulcsot leképezzük az adatokat tároló tömb egy adott indexére, ahol a keresett érték fellelhető.

Ideális esetben minden értelmezett kulcsra egyedi hash-t állít elő a függvény, de számolni kell azzal, hogy két különböző kulcsra ugyanazt a hash-t kapjuk és ezt kezelni kell.

Hash függvény: megmondja, hogy adott kulccsal azonosított elem a tömb melyik indexén található. Segítségével bármilyen hosszúságú adatot adott hosszúságra képezhetünk le. Mivel végtelenből végesbe képzünk le, ezért előfordulhat ütközés, de ezt a hash kulcs kellően nagyra választásával minimalizálni lehet. Ha bármilyen változás áll elő a kiindulási adatban, annak hash-e megváltozik.

Kereső algoritmusok hatékonysága

Lineáris keresés: visszaadja, hogy egy sorozatban létezik-e a feltételnek megfelelő elem vagy sem. Ha igen, visszaadja az utolsó ilyen elem indexét (alapesetben). Legjobb esetben 1 lépést igényel, legrosszabb esetben N lépést igényel, **vagyis a hatékonysága lineáris.** Rendezett sorozatban nincs értelme folytatni, ha a vizsgált elem már nagyobb, mint a keresett érték.

Kiválasztás: egy olyan sorozatból, amiben biztosan szerepel a feltételnek megfelelő elem visszaadja az utolsó ilyen elem indexét (out). Hatékonysága lineáris.

Strázsás keresés: egy olyan sorozatot, amelyben nem biztosan szerepel a feltételnek megfelelő elem olyanná teszünk, amiben biztosan szerepel az utolsó+1. helyen. Emiatt hatékonyabb, mint a lineáris, mert nem kell vizsgálni a végén, hogy a sorozat végére értünk. Ha az eredeti sorozatban szerepelt a feltételnek megfelelő elem, visszatér igaz értékkel és az indexszel. Ha nem szerepelt, akkor visszatér hamis értékkel és a strázsaelem indexével. Futási ideje 2/3-a a lineáris keresésnek.

Bináris keresés: egy rendezett sorozatban az első és utolsó index alapján meghatározzuk az aktuális középső elemet, és ehhez viszonyítva keressük az elemet. Ha a keresett elem kisebb a középsőnél, akkor a baloldali félben lesz, az utolsó indexet beállítjuk K-1-re. Ezt ismételjük, amíg meg nem találjuk az elemet. Legjobb esetben 1, legrosszabb esetben $\log_2 N$ összehasonlítást igényel, vagyis a hatékonysága logaritmikus.

Kiválogatás: egy sorozatból kiválasztja a feltételnek megfelelő elemeket és ezeket egy új sorozatba rendezi, majd visszaadja hány elemű az új sorozat

Rekurzió

Rekurzió: végrehajtásakor a saját maga által definiált műveletet, műveletsort hajtja végre, önmagát ismétli.

Iteráció: egy megadott lépéssorozatot hajtunk végre újra és újra, változó értékekkel.

Rekurzió előnyei:

- bizonyos matematikai függvények rekurzívan vannak definiálva, így ezzel a módszerrel könnyebben megvalósíthatók (faktoriális)
- fa adatszerkezet rekurzívan épül fel, érdemes vele használni
- csökkenti a program összetettségét

Rekurzió hátrányai:

- bizonyos esetekben lassabb, mint az iteráció
- ha túl mély a rekurzió, akkor megtelhet a verem és a program leáll stack túlcsordulás miatt
- memóriaigényesebb az iterációnál
- nehéz logikájú függvényeket nehéz átírni iteratívról rekurzívra

Fa

Homogén, dinamikus, szétszórt reprezentációjú adatszerkezet. Az elemeket hierarchiába szervezzük, tetején a gyökérelem, alatta a csomópontok, azok alatt a levélelemek vannak.

Bináris fa: minden csomópontnak legfeljebb két gyermeke lehet.

Szigorúan bináris fa: minden csomópontnak pontosan két gyermeke lehet.

Bejárások: részfákat rekurzív függvény meghívásával

- inOrder: bal, gyökér, jobb
- postOrder: bal, jobb, gyökér
- preOrder: gyökér, bal, jobb

Bármely fát megadhatunk a gyökérelemével, ebből bármelyik további elemét elérhetjük. A bináris fa elemeinek rekordjai három mezőt tartalmaznak:

- bal: elemtől balra található elem
- jobb: elemtől jobbra található elem
- adat: érték, amit az elem tárol

Általános fa esetében nincs ilyen, ott tömb vagy lista tartalmazza a csomópont gyermekeit.

A fa rekurzív adatszerkezet, mert a csomópontok rekordjai tartalmazzák a gyermekek elérését is, és így tovább.

Kiegyensúlyozott fa: minden csomópontjára igaz, hogy az egyik részfájának elemszáma legfeljebb eggyel tér el a másik oldali részfa elemszámától.

Keresőfa: olyan bináris fa, amelynek minden csomópontjára igaz, hogy a baloldali részfában a nála kisebb, a jobb oldali részfában a nála nagyobb elemek vannak, és kiegyensúlyozott. Itt gyorsan lehet bináris kereséssel keresni.

4. tétel

a) Magasszintű programozási nyelvek I: Alaptípusok, változók, konstansok, literálok. Operátorok. Szelekciós vezérlési szerkezetek. Ciklusok. Érték- és referenciatípusok memóriamenedzsmentje, stack és heap. Változók hatásköre és élettartama. Programozási nyelvek generációi, imperatív és deklaratív nyelvek.

Alaptípusok (elemi típusok)

Elemi típus: logikailag felbonthatatlan típusok, a nyelv eleve tartalmazza őket

C#

- byte: 1 bájtos, előjel nélküli egész
- sbyte: 1 bájtos, előjeles egész
- short: 2 bájtos, előjeles egész
- ushort: 2 bájtos, előjel nélküli egész
- nt: 4 bájtos, előjeles egész
- uint: 4 bájtos, előjel nélküli egész
- long: 8 bájtos, előjeles egész
- ulong: 8 bájtos, előjel nélküli egész
- string: szöveges típus, immutable
- char: 1 karakter
- bool: logikai típus, igaz/hamis
- loat: előjeles, lebegőpontos típus, 4 bájtos, egyszeres pontosság
- double: előjeles, lebegőpontos típus, 8 bájtos, dupla pontosság
- decimal: 16 bájtos, lebegőpontos típus

Értéktípus: konkrét értéket tárol, a stack-en allokál. Értéke nem lehet null (kivéve, ha Nullable típust használunk.) A fentiek értéktípusok. A string valójában referenciatípus (karakterek tömbje), de értéktípusként viselkedik, azaz immutable (belső állapotat nem változtatható).

Referenciatípus: memóriacímet tárol, ahol valamilyen érték(ek) van(nak). A heap-ben allokál. Értékük lehet null.

Literálok: bedrótozott értékek a forráskódban (értékek, nem változók). Van típusok és értékük: 2, 18, stb.

Konstansok: névvel ellátott literálok. Amikor konstansra hivatkozunk, a fordító behelyettesíti oda a konkrét értéket. Kezdeti értékadás kötelező, később nem módosítható. Osztályok esetében a konstansok alapból statikusak (nem lenne értelme, ha példányszintű lenne), ezért nem írhatjuk eléjük a static kulcsszót.

Változók: azonosító, amivel adatra vagy objektumra hivatkozunk, a memória egy bizonyos része. Lehet lokális, globális, statikus (élettartama a program futása), dinamikus (élettartama a program futása csak részben)

Operátorok

Műveleti jelek, melyek operandusokat kapcsolnak össze. Operandus lehet változó, konstans, függvényhívás.

- Egyoperandusú aritmetikai operátorok: +, -, !, ++, --
- Kétoperandusú aritmetikai operátorok: +, +=, -, -=, *, *=, /, /=, %, %=
- Bitshiftelő operátorok: <<, >>, <<=, >>=
- Összehasonlító operátorok: ==, !=, <=, >=, <, >
- Bitmanipulációs operátorok: &, &=, |, |=, ^, ^=, ~
- Logikai operátorok: &&, ||
- Értékadás: =
- Háromoperandusú feltételes operátor: feltétel ? ha_igaz : ha_hamis

Precedencia: az operátorok műveleti sorrendje. Minél kisebb, annál később kell elvégezni a műveletet. Szorzást előbb, mint az összeadást. Zárójelekkel felülírható.

Típusmódosító operátorok

- explicit: típuskényszerítés (casting), vagy az *as* operátor
- implicit: a környezet elvégzi helyettünk, nem kell tenni semmit: int → float, de visszafelé már nem

Szelekciós vezérlési szerkezetek

Egy logikai feltétel hatására a program végrehajtása két vagy több irányba folytatódhat.

If-else vagy switch.

Ciklusok

- Elől tesztelő: do while, ami egyszer mindenképpen lefut, utána csak akkor, ha a while feltétele igaz
- Hátul tesztelő: while, for, foreach

Hatáskör

Egy adott változót a program mely részeiben érhetünk el. Blokkok befolyásolják. Blokkon belül létrehozott változó hatásköre a blokk végéig tart: for ciklus i-je.

Élettartam

Szoros kapcsolatban áll a hatáskörrel. Ha a vezérlés eléri a hatáskör végét, véget ér az élettartam is. Ilyenkor a Garbage Collector törli a memóriából a változó értékét, de nem biztos, hogy azonnal, és nem minden programnyelvben.

Stack

- futási időben tárolja az értékeket, a művelet után, ha az adat már nem kell, felszabadít.
- metódusokat, lokális változókat, referenciákat tárolunk itt, amik a heap-ben lévő objektumra mutatnak
- gyorsabb, mint a heap
- lineáris
- OS menedzseli
- limitált méret
- nincs random elérés

Heap

- globális változók vannak itt
- lassabb, mint a stack
- hierarchikus
- nem az OS menedzseli
- RAM a max méret
- van random elérés

Generációk

Első generáció

- utasítások a memóriában
- regiszter: mikroprocesszoron belüli tároló
- programozás: elemi lépésekben, nehézkes, bonyolult a nyelv, nem áttekinthető a kód
- futtatás és hibakeresés nehéz
- processzor-specifikus gépi kód
- nagy futási sebesség
- jó memóriakihasználás

Második generáció

- Assembly: gépi kód megérthetőbb nyelve, mnemonikok, processzor-specifikus
- compiler: Assembly-ből gépi kódot fordít, ez volt az assembler. Fordít, ellenőriz. Kialakult az igény a fordítóprogramokra.
- változók: táblázatban felírták a használt memóriacímeket.
- alapvető típusok megjelenése: szám, szöveg
- feltétel nélküli és feltételes vezérlésátadás, ugrások
- szubrutinok szintén ugrással
- hosszú programokat modulokra tagolták
- linker: szerkesztő, ami sok apró tárgykódból elkészíti a működő programot. Újra felhasználható kód alapja.

Harmadik generáció

- eljárásorientált nyelvek
- eljárás: névvel ellátott, rögzített paraméterű kódrészlet, amelyben több utasítás van.
- megjelentek a névvel, típussal ellátott rendes változók
- elágazások, ciklusok megjelenése, nem ugrással
- hardver függetlenség: nem processzor-specifikusak a programnyelvek. A fordító az adott processzor gépi kódjára fordította a forráskódot.

Három és feledik generáció

- OOP

Negyedik generáció

- nyelvek kifejezőerejének növelése, szintaktikai hibák kiküszöbölése.
- speciális nyelvek: SQL, ami csak adatbázissal foglalkozik

Ötödik generáció

- mesterséges intelligencia nyelvek, folyamatban van

Imperatív nyelvek

- a probléma megoldását fogalmazzuk meg.
- procedurális nyelvek és néhány OOP tartozik ide, de azok többnyire multiparadigmásak: C, Pascal
- a program állapotát a memóriában lévő értékek határozzák meg, ezek manipulálásával változtatjuk az állapotot
- az értékadás a leggyakoribb művelet
- aritmetikai műveletek, függvény kiértékelések jellemzik, mellékhatásokkal járó utasításokat eljárásokba szervezik
- szekvencia, szelekció, iteráció
- feltétel nélküli ugrás: goto, adott sorra ugrik, nem minden nyelveben van ilyen
- blokkok, hatáskör, élettartam, switch utasítás

Deklaratív nyelvek

- a problémát magát fogalmazzuk meg, amire eredményt szeretnénk. Az, hogy hogyan kapjuk meg, nem fontos.
- SQL, Mathlab
- könnyű szintaxis, angol nyelv szerű
- bizonyos funkcionális nyelvek is támogatnak deklaratív lehetőségeket
- mesterséges intelligencia nyelvek ide tartoznak

b) Operációs rendszerek: Az operációs rendszer fogalmai, kernel, processz stb. A virtualizáció. Az operációs rendszerek fájl- és könyvtárkezelése, fájlrendszerek. Diszk kezelés, RAID tömbök. Átirányítások és szűrők. Jogosultsági rendszerek: működés, azonosságok és különbségek az egyes rendszerekben. Processz kezelés. Szignálok és kezelésük. Adatmentés és archiválás módszerei és eszközei. Shell-scriptek.

OS

Számítógépek alapprogramja, mely közvetlenül kezeli a hardvert és egy egységes környezetet biztosít a számítógépen futtatandó alkalmazásoknak. Mint a folyó a halaknak. Ütemezi a programok végrehajtását, elosztja az erőforrásokat, biztosítja a felhasználó és a számítógépes rendszer közötti kommunikációt.

Kernel

Az OS alapja, a hardver erőforrásainak kezeléséért felelős. Többfeladatos rendszer esetén ő szabja meg, hogy melyik program mennyi ideig használhatja a hardver egy adott részét. Háttérben futó, alapvető feladatokat ellátó program.

Processz

Folyamat, egy program memóriabéli futó példánya, amihez kapcsolódik a példány végrehajtásával kapcsolatos OS tevékenység. Egy processzt a címtartománya (memóriatérkép) és a processzustáblabéli információk alkotnak. A processznek szüksége van erőforrásokra, hogy elvégezze a feladatát.

OS három fő tevékenysége a processzek felügyeletével kapcsolatban:

- processz létrehozása és törlése
- processz felfüggesztése és újraindítása
- eszközök biztosítása a szinkronizációhoz és kommunikációhoz

Ütemező program: processzek ütemezését végzi, ellátja a processzort processzekkel. Megszakítás esetén a várakozási sorból kiválaszt egy másik processzt és megóvja a rendszert attól, hogy egy processz kisajátítsa a processzoridőt. Egy rendszerben az összes processz végrehajtódik, de egy időpillanatban egyszerre csak egy fut.

Virtualizáció

A fizikai gépen több virtuális gépet hozunk létre, mindre telepítünk OS-t és önálló gépként kezeljük őket. Meghatározhatjuk a memóriafelhasználást, CPU-teljesítményt, megosztoznak az erőforrásokon.

Típusai:

Bare-Metal: szerverek esetében. A fizikai hardverre először a virtualizációt végző szoftver a hypervisor kerül, ez kezeli a teljes erőforráskészletet. A virtuális gépek létrehozását és működtetését a hypervisor felügyeli.

Kliens oldali virtualizáció: a hardver egy hétköznapi OS-t használ, amin elindítunk egy virtualizációs szoftvert.

Virtualizáció megvalósítása

A virtuális gépekben futó OS-ek nem feltétlenül tudják, hogy virtualizált környezetben futnak, ehhez a virtualizálást végző szoftvernek alkalmazkodnia kell.

Erőforráskezelés módjai:

- szoftveres megoldás: a virtualizációs szoftver oldja meg
- paravirtualizációs megoldás: a virtuális gépben futó OS tudja, hogy nem fizikai gépen fut, és ennek megfelelően működik
- hardveres megoldás: a hardvert úgy fejlesztették, hogy képes legyen a megosztott működésre és több gép egyidejű kiszolgálására.

HALT: leállítja a fizikai gépet. Nem szerencsés, ha a virtuális gép ezt eredeti formájában alkalmazza. Megoldások:

- szoftveres: a szoftver vizsgálja a virtuális gép által kiadott utasításokat, és lecseréli olyan kódra, ami a fizikai helyett a virtuális gépre vonatkozik, ebben az esetben leállítás.
- paravirtualizációs: vendég OS érzékeli, hogy virtuálisan fut és a HALT helyett a hypervisor függvényét hívja meg
- hardveres: a CPU elkülönítetten kezeli a virtuális gépeket és képes ennek megfelelően kezelni a HALT-olt.

Fájlkezelés

- touch: a fájl utolsó módosításának idejét változtatja meg, de a mellékhatása, hogy ha nem létezik a fájl, akkor létrehozza
- cp: másolás
- mv: mozgatás
- rm: törlés
- cat: fájl tartalmának kilistázása
- file: megmondja, hogy egy adott fájlban milyen típusú adatok vannak

Könyvtárkezelés

- ls: aktuális könyvtár tartalmát kilistázza
- pwd: kiírja, hogy melyik könyvtárban vagyunk
- tree: külön telepíteni kell, rekurzívan végigjárja a könyvtárat és egy fa szerkezetben jeleníti meg a könyvtár- és fájlstruktúrát
- cd: paraméterben megadott könyvtárba ugrunk, cd .. eggyel felettünk lévő
- mkdir: könyvtár létrehozása
- rmdir: könyvtár törlése

Fájlrendszerek

FAT: régi, viszont kis tárolókon gyors, Windows 95, 98. FAT32-őn nem lehet 4 GB-nál nagyobb fájlokat tárolni.

NTFS: Microsoft, alapértelmezett Windows-on. Merevlemezekkel kapcsolatos bizonyos hibákat automatikusan helyre tud állítani. Nagyobb merevlemezeket is támogat, lehetőség

van engedélyek és titkosítás használatára. A fájlok elérhetősége a jóváhagyott felhasználókra korlátozható.

ext4: naplózó fájlrendszer, Linux

AFPS (új) és HFS+ (régi): Mac, elsősorban SSD-re és flash memóriára lett tervezve, nagy hangsúly a titkosításon.

RAID tömbök

Tárolási technológiák, amelyekkel az adatok elosztása vagy replikálása több fizikailag független merevlemezen, egy logikai lemez létrehozásával lehetséges. Minden RAID az adatbiztonság növelését vagy az adatátviteli sebesség növelését szolgálja.

RAID0: összefűzés. Nem redundáns, ezért nincs hibatűrés. Sebessége a leggyorsabb, ideális esetben a lemezek sebességének összege. A1A2A3A4-A5A6A7A8

RAID1: tükrözés. Ugyanaz az adat van mindkét lemezen. Olvasási sebesség párhuzamos, ezért gyorsabb, de az írás normál. Bármelyik meghajtó meghal, folytatódhat a működés. A1A2A3A4-A1A2A3A4

RAID2: használja az összefűzést, illetve egyes meghajtókat hibajavító kód tárolására tartanak fenn, vagyis az adatbitekből redundáns biteket képeznek. Megnövekedik a tárigény. Ezen meghajtók egy-egy csíkjában a különböző lemezeken azonos pozícióban lévő csíkokból képzett hibajavító kódot tárolják. Képes a lemezhiba detektálására és kijavítására. Manapság nem használják. A1B1C1D1-A2B2C2D2-A3B3C3D3-A4B4C4D4-Ap1Bp1Cp1Dp1-Ap2Bp2Cp2Dp2-Ap3Bp3Cp3Dp3

RAID3: teljes hibajavító kód helyett csak egy lemeznyi paritásinformáció tárolódik. Egy paritáscsík a lemezeken azonos pozícióban elheyezkedő csíkokból XOR művelettel kapható meg. Ha egy meghajtó kiesik, nem gond, mert a rajta lévő info az előző módon megkapható. Kisméretű csíkok vannak, ezért az írás és olvasás párhuzamosan történik. A1A4A7B1-A2A5A8B2-A3A6A9B3-Ap(1-3)Ap(4-6)Ap(7-9)Bp(1-3)

RAID4: RAID3, de nagyméretű csíkok vannak, egy rekord egy meghajtón van, ezért egyszerre több rekord párhuzamos írását, olvasását teszi lehetővé. De a paritás-meghajtó adott csíkját minden íráskor frissíteni kell, így ez lesz a szűk keresztmetszet. Ha egy meghajtó kiesik, az olvasási teljesítmény csökken. A1B1C1D3-A2B2C2D2-A3B3C3D3-ApBpCpDp

RAID5: paritás információ egyenletesen van elosztva a meghajtók között. Legalább 3 meghajtó kell, de az írási sebesség csökkenésének megakadályozása miatt 4 az ajánlott. Írás és olvasás párhuzamos. Hiba esetén az adatok sértetlenül visszaolvashatóak, a hibás meghajtó adatait a vezérlő ki tudja számolni. A1B1C1Dp-A2B2CpD1-A3BpC2D2-ApB3C3D3

RAID6: RAID5 kibővítve, nem csak soronként, hanem oszloponként is kiszámítják a paritást. Kétszeres meghajtómeghibásodást tudnak kiküszöbölni. Legalább 4 meghajtó, de 6 az ajánlott.

RAID01: RAID0 sebessége és a RAID1 biztonsága. Legalább 4 eszköz kell, melyekből 1-1-et összefűzve, majd páronként tükrözve kell felépíteni, ezért a kapacitás fele használható.

RAID10: először tükrözünk és utána fűzzük őket össze. Biztonság szempontjából jobb, mert a kiesés csak az adott tükrözött tömböt érinti. Sebességben azonosak.

Átirányítások

A parancsok/programok írhatnak a standard outputra, a bemenetet pedig a standard inputról kaphatják. Ezek azok a folyamatok, amiket átirányíthatunk.

Kimenet: >>, pl. fájlba

Bemenet: <<, pl. fájlból

Szűrők

Olyan átirányítások (parancsok), amelyeket pipe létrehozásra hoztak létre. A bemenet sorait olvassák, feldolgozást végeznek rajta és ennek eredményét a kimenetre írják.

grep: bemenetről érkező sorokból azokat írja kimenetre, amelyek a paraméterként megadott szöveget tartalmazzák vagy nem (-v kapcsoló).

cut: bemenetről érkező sorokból a paramétereinek megfelelően kiválogat darabokat és továbbítja a kimenetre. Ha csak a sor részletére van szükség.

wc: szavak, sorok, karakterek megszámolására képes. Ha darabszámra van szükség.

head és tail: fájl első vagy utolsó néhány sora

sort: bemenet sorainak rendezése abécé szerint.

uniq: bemenetről érkező azonos sorokból csak egyet ír kimenetre

tr: bemenetről érkező sorokban keresést és cserét végez: karakterek cserélhetőek.

Jogosultsági rendszerek: működés, azonosságok és különbségek az egyes rendszerekben

Minden többfelhasználós rendszerbe kell.

passwd fájl: felhasználókról tárol információkat. Login név, jelszó, userID, groupID, egyéb, home könyvtár, shell

shadow fájl: kódolt jelszavak, utolsó csere óta eltelt napok száma, figyelmeztetés jelszó lejáratról, inaktivitás

Csoportok kezelése: azért, hogy ne egyesével kelljen mindenkire.

chmod: jogosultságok beállítása egy fájlhoz

chown: fájl tulajdonosának beállítása, cseréje

chgrp: fájl csoportjának beállítása, cseréje

Jogosultságok leírása 3 oszlop: jogosultság, tulajdonos csoport. A jogosultságos 9 karakterből áll, felhasználó, csoport és más felhasználók 3-3-3.

Olvasás (2) (r) Írás (4) (w) Végrehajtás (7) (x)

Fájlokra és könyvtárakra vonatkoznak. Könyvtárnál a végrehajtás a könyvtárba való belépést és a fájlok elérését jelenti.

Példa: rwxr-x---: a tulajdonos olvashat, írhat, futtathat. A csoport olvashat, futtathat. Többi felhasználó semmit nem tehet vele.

Szignálok

Processzek kommunikációjára való Unix rendszereknél. A szignál egy üzenet, amelyet az a processz tud fogadni, akinek küldték. Feldolgozása a signal handler függvényben történik, ha a program készítője definiálta azt.

Használatának okai: processz terminálása (futás megszakítása), konfig fájl újraolvasása.

Szignál küldése: kill par1 (küldendő szignál neve), par2(processz ID)

killall: a program összes példánya megkapja a szignált.

Ctrl-c: INT szignál

Ctrl-d: STOP szignál

SIGTERM: processz lépjen ki szabályosan

SIGSTOP: a processz blokkolva lesz és nem lesz CPU-ra ütemezve

SIGINT: billentyűzetről megadott megszakítás, Ctrl+C

SIGKILL: nem értesül róla a processz, az OS fogadja és terminálja, csak akkor, ha a SIGTERM-re nem szakad meg a futás.

Adatmentés és archiválás módszerei és eszközei

Adatmentés: célja másolat készítése azért, hogy az adatvesztés elkerülhető legyen.

Archiválás: célja az adatok állapotának rögzítése, tartós megőrzése.

Stratégiák

- napok végén külön a napot elmenteni, x nap után rámenteni a base-re
- egymásra menteni a napokat, 1, 1-2, 1-3, 1-4 majd rámenteni a base-re

Snapshot: a fájlrendszer pillanatnyi állapotának rögzítése. A snapshot pillanatától a megváltozott rekordok egy átmeneti területre kerülnek, a snapshot megszűnésekor íródnak a helyükre.

Tömörítés: Huffman kód, Shannon-Fano kód

Tömörítő programok: gzip, bzip2, zip, rar, arj, 7zip, tar

Shell-scriptek

Shell parancsok sorozata, amit egy file-ba írunk, legtöbbször .sh-ba.

Filozófia: az adatfolyam különböző hasznos szűrőkön való átfolyatásán alapszik.

Változó értékadás: valtozo=ertek, aztán használni $valtozo.

Számított kifejezés: $()

read valtozo: olvasás a standard inputról

echo: írás a standard inputra

Vezérlési szerkezetek

if feltétel

then

fi

for i in *.txt

do

done

while true

do

done

5. tétel

a) Magasszintű programozási nyelvek II: Az OOP alapelvei, alapfogalmai. Mezők, metódusok, property-k, adatrejtés. Konténerosztályok használata és indexelők. Osztályszint és példányszint. Névterek. Bővítő metódusok. Operátor overloading.

OOP alapelvek

Egységbezárás: az adatstruktúrát és az adatokat kezelő metódusokat egységként kezeljük és elzárjuk őket a külvilág elől. Az objektum belső állapotát védjük. Az így kapott egység az objektumosztály.

Öröklődés: a meglévő objektumokból levezetett újabb objektumok öröklik a definiálásukhoz használt alap objektumok egyes adatstruktúráit és függvényeit, de újabb tulajdonságokat is definiálhatnak vagy régieket újraértelmezhetnek. Öröklődés során a private mezők és metódusok kivételével átvesszük a dolgokat. Az öröklődés miatt csökken a kód redundanciája és olvashatóbb, elszeparálhatóbb lesz a kód.

Sokalakúság: egy metódus azonosítója közös lehet egy adott objektumosztály-hierarchián belül, de a hierarchia minden egyes objektumosztályában a metódus implementációja az adott objektumosztályra nézve specifikus lehet.

OOP alapfogalmak

objektumosztály: olyan egység, amely mezőket és metódusokat tartalmaz. Típusként jelenik meg.

mező: adattárolási feladatot lát el, sokban hasonlít a változóhoz. Egy mező élettartama az objektum élettartamával jellemezhető.

metódus: eljárás vagy függvény

objektum: változó, amelynek típusa egy objektum-osztály

inicializálás: az objektum alaphelyzetbe állítása. Mezőknek valamilyen kezdőértékük van.

adatrejtés: a mezők közvetlen hozzáférés elleni védelme

konstruktor: speciális metódus, mely az inicializálás során kezdőértékeket állít be. Több is lehet belőle

példányosítás: objektumosztály egy példánya számára memóriafoglalás és inicializálás. Legalább egy konstruktor meghívásra kerül.

destruktor: speciális metódus, mely az objektum változó memóriából eltűnése előtt bizonyos nagytakarítás jellegű utasításokat hajt végre.

Property (setter-getter)

A vele megegyező változó védelméért, ellenőrzésért felel. Lehet csak setter, csak getter. Lehetnek private-ok (getternél nincs értelme), vagyis csak az osztályon belül lehet írni vagy olvasni.

Set: beállítás, validálható a bekért érték a value kulcsszóval

Get: visszaadja a property-ben a hozzá tartozó változó értékét

Adatrejtés

Változót és metódust védhetünk vele. Ne lehessen kívülről megváltoztatni az objektum állapotát, vagyis a változók értékét. Alapértelmezetten minden változó private.

- private: csak az adott osztályon belül lehet látni
- protected: csak az adott osztály, és a gyerekosztályai láthatják
- public: minden osztály számára látható, aki referenciát kap hozzá
- private protected: tartalmazó osztályon belül, illetve az adott assembly-n belüli gyerekosztályokban érhető el
- protected internal: az adott assembly-n (összetartozó dolgok, project, DLL) belül és az objektumot tartalmazó osztály gyerekeiben érhető el

Konténerosztály

Több saját osztály adatát tároljuk benne. Például játékban vannak orkok, akkor lehet egy Horda osztály, amiben az összes ork benne van. Az orkok különbözhetnek egymástól, lehet sámán, harcos, stb. Léteznek heterogén (különböző típusú objektumokat tároló) és homogén (azonos típusú adatokat tároló) konténerosztályok.

Indexelők

Speciális property, aminek a neve this. Paraméterezhető, ezt []-ben adjuk meg. Segítségével indexelést végezhetünk a saját osztályunkra, mint a tömbökre.

Példányszint

Olyan mezők, amelyek adott osztály példányaiban eltérőek lehetnek: kutya neve. Lehet readonly, a mező értékét csak konstruktor vagy kezdőérték adás állíthatja be. Meghívásához létre kell hozni egy példányt. This-szel hivatkozhatunk rá.

Osztályszint

Olyan adatokat kell bele rakni, amelyeket program szinten elég egyszer tárolni. Nem a példányokat jellemzi, hanem az összes példányt. Static kulcsszó kell hozzá. This-szel nem lehet rá hivatkozni, mert nem köthető konkrét példányhoz. Osztálynév.mezőnévvel hivatkozunk rá. Readonly itt is használható: kezdőérték megadásával, vagy osztályszintű konstruktorral állíthatjuk be az értékét.

Névtér

Segítségükkel csoportokba rendezhetjük az osztályokat és enumokat. Névtér eléréséhez usingolni kell azt. Osztály: egységbe zárjuk az összetartozó funkcionalitásokat. Névtéren belül 2 ugyanolyan nevű osztály nem lehet, de másik névtérben már igen. Névtér tartalmazhat névteret.

Bővítő metódusok

Meglévő osztályt anélkül bővítsünk saját metódussal, hogy az osztályt módosítanánk. Ezek a metódusok statikusok és első paraméterük az osztály egy példánya a this kulcssózval az elején (például: public static korEgyenlete(this Kor k…)). Példányon keresztül el fogjuk tudni érni ezt a metódust is.

Operator overloading

Saját osztályra újraértelmezhetjük az operátorokat. Megadhatjuk, mint jelentsen az, hogy kutya1>kutya2. Megmondhatjuk, hogy hogyan kell meghatározni azt, hogy az egyik kutya nagyobb, mint a másik. Ehhez írni kell egy public static metódust, aminek a visszatérési értékét mi választjuk feladattól függően, utána az operator kulcsszó következik, majd az operátor megnevezése: ++, >, < stb.

b) Architektúrák: A cache használat. Lokalitási elvek ismertetése. A memória hierarchia szintjeinek összehasonlítása, a cache(ek) helye a szintek között. Felépítése, az elérési változatok. Címbitek számának meghatározása az elérési módoktól függően. Visszaírási módok. A dirty és a valid flag-ek szerepe.

Cache

Átmeneti tároló, célja az adatokhoz való hozzáférés gyorsítása. A gyorsítás alapja, hogy a gyorsítótár gyorsabb tárolóelem, mint a hozzá kapcsolt gyorsítandó tárolók. Ha a CPU a központi tár egy rekeszét olvasni akarja, akkor a gyorsítótár vezérlője bemásolja ezt a cache-be. A cache elérése rövidebb, mint a háttértáraké vagy a memóriáé.

cache hit: a CPU olyan adatot kér, ami benne van a cache-ben

cacse miss: a keresendő adat nincs a cache-ben

Lokaliási elvek

A gyorsítótár gyorsaságának alapja a lokalitási elv, mely szerint a CPU hajlamos ugyanazokat a memóriaterületeket többször is felhasználni.

Időbeli lokalitás: ha egyszer hozzányúltunk, akkor többször is hozzá fogunk

Térbeli lokalitás: ha hozzányúltunk, a környezetéhez is hozzá fogunk

Memória hierarchia

1. szint: CPU, regiszterek (1 nanosec)
2. szint: cache
    a. level 1 ~4-6 ns
    b. level 2 ~8-12 ns
    c. level 3 ~16-24 ns
3. szint: operatív tár, DRAM ~40-130 ns
4. szint: háttértár, SSD ~0.05-0.1 ms, HDD ~4-8ms
5. szint: archív tár, ODD ~80-160ms

Cache elérési módja, címbitek meghatározása

Teljesen asszociatív cache

A gyorsítandó memóriaegység beolvasott blokkjának tartalma a gyorsítótár bármely blokkjának területére kerülhet. Az elhelyezés sorát a helyettesítési algoritmus határozza meg. Amikor a CPU adatot keres a cache-ben, akkor a memóriabéli cím felső 28 bitjét összehasonlítja a cache-beli blokkszámokkal az összes sorban egyidejűleg. Ha sikeres a keresés, kijelöli az adott sorbéli byte-ot, egyébként folytatja a keresést a memóriában. Előnye, hogy rugalmas, de a visszakereséshez annyi áramkör kell, ahány találati sor van. Hátránya a helyettesítési eljárás (az algoritmus)

Címbitek száma: n bites címbusz, m byte-os adattár esetén n – $\log_2 m$

Közvetlen leképezésű cache

A gyorsítandó memóriaegység beolvasott blokkjának tartalma a gyorsítótár meghatározott blokkjára kerülhet. A blokk helyét a cache-ben a blokksorszám alsó 8 bitje határozza meg. A blokk csak abba a sorba kerülhet, amelyet a sorindexe meghatároz. Az adat 16 byte, a

lapsorszám 20 bites. Ezek mellett a jelzőbitek kerülnek tárolásra. Kereséskor a memóriacímből előállított sorindex alapján keresi a sort, majd a felső 20 bitet összehasonlítja a cachebéli lapsorszámmal. Előnye, hogy rövid a lapsorszám és a keresés gyors és olcsó. Hátránya, hogy alacsony a találati arány.

Címbitek száma: n bites címbusz, m byte-os adattár, k soros cache esetén $n – \log_2 m - \log_2 k$

Csoport asszociatív cache

A gyorsítandó memóriaegység beolvasott blokkjának tartalma a gyorsítótár egy blokkcsoportjába képződhet. Átmenet az előző két típus között, mert adott blokk csak egy adott csoportra képződhet, de ezen belül a blokkcsoport bármely blokkjára. Itt a cache nagyobb csoportokra (n db) van osztva, amelyek önmagukban teljesen asszociatívak. 22 bit lapsorszám, 23-28 bit csoportszám (alsó 6 bit), 16 byte adat, 2 jelzőbit. Kereséskor a memóriabéli címből képzett csoportindex alapján jelöljük ki a csoportot, majd a felső 22 bit alapján a cachebéli lapsorszámot. Előnye, hogy kevés összehasonlítást igényel.

Címbitek száma: n bites címbusz, m byte-os adattár, k soros cache, 1 soros blokkok esetén $n – \log_2 m – (\log_2 k – \log_2 l)$

Szektor leképezésű cache

Ritkán használt, ahol a beolvasott blokk szintén egy-egy blokkcsoportba képződik, de fordítva: a beolvasott blokk tetszőleges blokkcsoportra képződhet, de azon belül csak meghatározott blokkra. A CPU a csoport helyét jelöli ki asszociatív módon, és azon belül a blokk helye a lapon belüli elhelyezkedésének megfelelően kötött.

Címbitek száma: n bites címbusz, m byte-os adattár, k soros cache, 1 soros blokkok esetén $n – \log_2 m – (\log_2 k – \log_2 l)$

Visszaírási módok

write through: a CPU által módosított adatot kiírja a memóriába és a cache-be is egyszerre (lassú)

write back: a CPU által módosított adatot csak a cache-be írja ki, a dirty bitet 1-re változtatja, a cache sor törlése előtt visszaírja az értéket a memóriába

Dirty flag

1 biten tárolja, hogy a cache adott sorában lévő adatot a CPU módosította-e, vagyis vissza kell-e írni. Ha módosította az érték 1 lesz. Ha visszacsináljuk a módosítást, akkor is 1 marad. Visszaírás után lesz 0.

Valid flag

1 biten tárolja, hogy az adott sorban lévő adat érvényes-e. A gép bekapcsolásakor minden valid bit 0. Ha egy sorba adatot másolunk, az értéke 1-re változik. Ha fel akarunk szabadítani egy sort (mert megtelt a cache) az értékét 0-ra változtatjuk.

6. tétel

a) Numerikus matematika: Hibák típusa, hibaterjedés. Nemlineáris egyenletek megoldása. Numerikus integrálás. Lineáris egyenletrendszerek megoldása. Függvény kiértékelés. Interpoláció. Legkisebb négyzetek módszere.

Numerikus matematikai eljárásokra azért van szükség, hogy a számítógépeknek értelmezhető módon tudjunk átadni olyan problémákat megoldásra, melyekre egyébként nem lennének alkalmasak. A végtelen fogalmát nem ismerik. A komputeralgebrai rendszerek a numerikus matematikára építve végzik a számításaikat.

Hibák típusa, hibaterjedés

Hibák típusai:

- kiküszöbölhetetlen hibák (pl.: input adatok hibái)
- matematikai modell hibája: a modellek csak közelítenek a valósághoz
- módszer-hiba: a kiválasztott numerikus módszertől függ
- számítógép alkalmazása kapcsán keletkezett hibák: kerekítés

Abszolút hiba

Legyen **a0** az adat pontos értéke, és **a** az **a0** közelítő értéke, akkor **a – a0** lesz az adat közelítő értékének hibája. A közelítő érték hibabecslését (hibakorlátját) **Δa**-val jelöljük.

$|a–a_0| \leq \Delta a$

Kiszámításnál a pontos értékből kivonni és ahhoz hozzáadni is kell a **Δa**-t, így kapunk egy hibaintervallumot.

Relatív hiba

Az **a0** adat az **a** közelítő értékének a relatív hibája:

$$\delta_a = \frac{\Delta_a}{|a_0|},$$

$$\boxed{\delta_a = \frac{\Delta_a}{|a|}} \quad (a \neq 0)$$

Mivel általános esetben **a0** értéke ismeretlen, a relatív hibát formában tudjuk meghatározni. Megadása %-ban történik.

Egy adat relatív hibabecslése és a biztos számjegyeinek számának kapcsolata:

ha 10%, akkor az adat egy biztos számjegyet tartalmaz

ha 1%, akkor kettőt

ha 0.1%, akkor hármat

Hibaterjedés

Mennyiben függenek az alapul szolgáló mennyiségek hibáitól az azokból származtatott mennyiségek hibái. Ha van egy x mennyiségem és ennek egy **Δx** hibája, akkor az ebből származtatott y mennyiség **Δy**-ja mekkora lesz? Azért fontos, mert az x mennyiségem lehet mért, pl. fizikában, és ott bizony előfordulhatnak hibák. Gyakorlatilag a származtatott mennyiségek hibáit jelenti a hibaterjedés.

Nemlineáris egyenletek

Nemlineáris egyenlet: a változók legalább a második hatványon vannak. A kérdés, hogy az egyenlet megoldása mikor lesz 0, vagyis az *f(x) = 0* megoldásait közelítjük numerikus módszerekkel.

Zérushelyközelítő eljárások (?, Y_0). Feltételeik:

- a függvény folytonos: az ÉT. bármely pontjához pontosan 1 értéket rendel
- a vizsgált [a, b] intervallumon pontosan 1 zérushelye van, egyszer metszi az x tengelyt
- a vizsgált a és b pontok ellentétes előjelűek (szorzatuk negatív)

Intervallumfelező eljárás

Az új pont a 2 régi pont számtani közepe. Az eredmény 2 új intervallum, megvizsgáljuk rájuk a feltételeket, amelyikre mind teljesül azzal folytatjuk. Így a pontos gyökhöz konvergáló sorozatot kapunk.

Az eljárás:

Kijelölünk egy intervallumot, amely tartalmazza a zérushelyet (pontosan 1-et) és a végpontjairól tudjuk azt, hogy az azokban vett függvényértékek szorzata negatív. Elvégezzük a felezést, majd a vizsgálatot, hogy melyik irányba kell folytatnunk. Ha hibahatáron belülre kerültünk, leállunk.

A módszer két alappontra támaszkodik (a, b). A megoldás gyorsasága sok mindentől függ, pl.:

- a függvénytől
- hogyan válasszuk meg ε-t
- hogyan válasszuk meg a-t és b-t

Húrmódszer

2 ponton áthaladó egyenes egyenlete. Kiszámítása időigényesebb, nagyobb számítási kapacitást igényel, de gyorsabban konvergáló sorozatot ad, mint az intervallumfelező eljárás.

Az eljárás:

Kijelölünk egy intervallumot, kiszámoljuk f(a)t és f(b)-t (y-on lévő értékek), majd ezeket összekötjük egy egyenessel. Megnézzük, hogy hol metszi ez az egyenes az x-tengelyt. Ez az érték lesz a c. Kiszámoljuk f(c)-t. Ha c-t elfogadjuk közelítő értéknek, akkor végeztünk, egyébként kössük össze egy egyenessel f(a)-t és f(c)-t és ismételjük meg az eljárást.

Érintő módszer (Newton-Raphson eljárás)

Csak 1 kezdőpontra van szükség. Feltétel: az ÉT. bármely pontjában legyen a függvény egyszeresen deriválható.

Az elsőfokú Taylor-polinom egy érintője lesz az f függvénynek az f(a) pontban. Ezt kihasználhatjuk gyökkeresésre.

Az eljárás:

Az ÉT. egy adott értékéhez tartozó pontban érintőt húzunk (elsőfokú Taylor-polinommal). Mivel ismerjük az érintő függvényét, könnyen megnézhetjük, hogy az hol metszi az x-tengelyt. Ebben a metszéspontban vesszük a függvény értékét, majd ehhez húzunk egy érintőt. Az i.

pontot a következő formula adja (itt metszi az x-tengelyt az egyenes). Ismételjük ezt addig, amíg a metszéspont első koordinátája kellően közel nem kerül x*-hoz.

Az, hogy a Taylor-polinomot erre felhasználhatjuk, könnyen bizonyítható úgy, hogy az érintőpontban meghatározzuk az azon áthaladó egyenest. Ezt megtehetjük az adott ponton áthaladó, adott irányú egyenes egyenletének segítségével. A meredekség az első derivált helyettesítési értéke lesz az adott pontban (mivel egy adott pontban a függvény deriváltja megadja az azon áthaladó érintőt).

A módszer előnye, hogy elég hozzá egy pont, hátránya, hogy nem mindig deriválható a függvény az ért. tart bármely pontjában.

Szelő módszer

Előfordulhat, hogy az érintőmódszert nem tudjuk alkalmazni, mert a feltétele nem teljesül. Ekkor, ha legalább az első érintőpontban differenciálható a függvény, akkor megtehetjük, hogy elindulunk az érintőmódszer elvei szerint, de az első után nem érintőket húzunk, hanem az első pont érintővel megegyező meredekségű egyeneseket. Tehát a létrehozott egyenesek párhuzamosak lesznek. Ennek előnye, hogy elég, ha csak 1 adott pontban deriválható a függvény, viszont lassabban konvergáló sorozatot ad.

Érintő- és húrmódszer együttes alkalmazása

Ha balról az érintőmódszerrel közelítünk, jobbról pedig a húrmódszerrel, akkor gyorsabban konvergálhatunk a zérushelyhez.

Inverz interpoláció

A módszer alapja, hogy egy függvény zérushelye pontosan az az x érték, amelyet az inverze rendel a 0-hoz. (x annyi, mint y az origó0-ban.) Ezt a kapcsolatot használjuk ki gyökkeresésre.

Feltételei:

f(x) = x - g(x) alakra hozható legyen az egyenlet. Az f(x) = 0 mindig felírható x = g(x) alakban.

A függvénygörbe meredeksége nem lehet 1-nél nagyobb vagy -1-nél kisebb. Ha ezen belül van, akkor konvergens és lehet közelíteni, ha nagyobb, akkor divergens és távolodnánk a zérushelytől.

Az eljárás:

45 fokos egyenest állítunk, majd vesszük a függvény egy fix pontjának y koordinátáját és behelyettesítjük a 45 fokos egyenesbe. Ekkor megkapunk egy x értéket, melynek segítségével a keresett függvény egyenletébe behelyettesítve megtaláljuk a hozzá tartozó y-t. Ezt az y-t újra a 45 fokos egyenesbe helyettesítjük és kapunk egy x-et. Az eljárással közelítjük a 2 függvény metszéspontját.

Numerikus integrálás

Függvény: rendezett párok halmaza, ahol a rendezett pár első eleme az ÉT.-ból, a 2. eleme az ÉK.-ből kerül ki és a halmazban nem szerepel kétszer ugyanaz az első elem.

Numerikus integrálás: közelítő eljárás az integrált kiszámításához

- Határozatlan integrál eredménye: primitív függvény.
- Határozott integrál eredménye egy valós szám.

- integrál: görbe alatti terület.

Ha egy függvényt integrálunk, megkapjuk a primitív függvényét: a függvény integráltja is egy függvény. Ha a primitív függvényt deriváljuk, megkapjuk az eredeti függvényt.

Vannak olyan függvények, amelyeknek nincs primitív függvényük. Ettől függetlenül van grafikonjuk, tehát van értelme beszélni az az alatti területről. Ilyenkor csak közelítő integrálással tudunk eredményhez jutni.

Felosztjuk egyenlő részekre a függvény integrálandó intervallumát. Az intervallum két végpontja a és b, valamint n részre szeretnénk felosztani. Ez lesz a lépésköz. Az intervallum kezdőpontjából elindulunk h lépésközzel. Módszertől függően téglalapokat határozunk meg, amelyek területeit összegezzük és így kapunk egy közelítést a függvény grafikonja alatti területre.

Téglalap módszer:

- alsó téglalap módszer: a téglalap nem x tengely felőli részén, a görbénél $f(x_i)$ és $f(x_{i+1})$ közül a kisebbet választja
- felső téglalap módszer a téglalap nem x tengely felőli részén, a görbénél $f(x_i)$ és $f(x_{i+1})$ közül a nagyobbat választja
- első téglalap módszer: mindig az aktuális $f(x_i)$ lesz a magasság
- második téglalap módszer: mindig az aktuális $f(x_{i+1})$ lesz a magasság
- harmadik téglalap módszer: az alsó-felső pontok felezőpontjában vett téglalap

Trapéz módszer: a részintervallumok két végpontjában lévő pontokat (nem megy végig a téglalap, csak a görbéig, és ezeket a metszéspontokat kötjük össze) összekötjük egy egyenessel, majd az x-tengellyel, így trapéz alakzatot létrehozva.

Valójában interpolálás történik. A téglalap módszernél nulladfokú, a trapéz módszernél elsőfokú polinomokkal közelítünk.

Az interpolálás lehetőséget kínál arra, hogy ha egy függvény nem integrálható, akkor közelítünk vele és azt fogjuk integrálni. Lényegében keresünk egy olyan függvényt, ami közelíti és integrálható.

Simpson formula

Veszünk három egymást követő osztópontot P_{i-1}, P_i, P_{i+1}. Ezekhez határozunk meg interpolációs polinomot (Lagrange) és ehhez rendelünk egy területet. Így két intervallumot használunk fel egy közelítéshez: [x i-1, x i], [x i ,x i+1].

A meghatározott másodfokú polinomot már egyszerű integrálni. Polinomfüggvény primitív függvényét egyszerű meghatározni: a fokszámhoz hozzáadunk 1-et és azzal el is osztjuk. Ezenkívül az összegfüggvények integrálási szabályát kell alkalmazni (tagonként integrálunk).

Számítása:

- Az első és utolsó pontban egyszeres súllyal vesszük a függvényértékeket.
- Páros pontokban négyszeres,
- páratlan pontokban kétszeres lesz.

Feltétele: legalább 5, és ezen felül páratlan számú osztópont.

Lineáris egyenletrendszerek

Több ismeretlen szerepel bennük, de mind legfeljebb az első hatványon.

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m$$

Megoldás: minden egyenletből kifejezünk egy ismeretlent, és behelyettesítjük a többibe, vagy Gauss eliminációt használunk.

Utóbbihoz felírjuk kibővített mátrixos alakban.

$$K = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{bmatrix}$$

Az elimináció során a rendszerünket visszavezetjük valamely háromszög- vagy átlós mátrixszal reprezentálható alakra. Ezt sorozatos, jobb és bal oldalon egyaránt alkalmazott lineáris transzformációk segítségével tesszük. Ezek elemi sorekvivalens átalakítások: sorok felcserélése. egy sor elemeinek nullától különböző számmal történő szorzása, egy sor konstansszorosának másikhoz való elemenkénti hozzáadása.

Függvény kiértékelés

Horner-módszer: polinom-függvények kiértékelésére használható.

Mivel a polinom hatványokból áll, ezért a számítógéppel való kiértékelése időigényes. A módszer úgy rendezi a polinomot, hogy abban csak összeadás és szorzás legyen, így gyorsabban kiértékelhető.

Taylor-polinom: függvények közelítésére használható.

Minden függvény előállítható vele egy végtelen polinomként. Annál pontosabb a közelítés, minél több tagot veszünk. Adott az f függvény és az ÉT. egy pontja, mely pontban a függvény végtelen sokszor deriválható.

A polinomok jók, mert sok formula bemenetei és könnyű velük számolni, mert nincs bennük hatványozás.

Az *a* ÉT. béli pont környezetében közelítünk. Ebben a pontban a Taylor-polinom és a közelíteni kívánt függvény értéke pontosan megegyezik. Ahogy távolodunk ettől az *a* ponttól, a

pontosság csökken. Minél több tagot veszünk, annál közelebb leszünk a közelítendő függvényhez és annál távolabb is még közelíteni tudjuk.

Interpoláció

Matematikai közelítő módszer, amely egy függvény nem ismert értékeire az ismert értékek alapján ad közelítést.

Ennek sok valós felhasználása van, főleg méréseknél. Ilyenkor bizonyos értékpárokat tudunk, amelyeket felrajzolhatunk, majd ezekre illeszthetünk egy polinomot, hogy meg tudjuk becsülni a lehetséges értékeket olyan értelmezési tartománybeli elemekre, amelyekhez nem tartozik mérés.

Az interpolációs polinom fokszáma mindig az eredeti interpolálni kívánt függvény megadott pontjainak száma – 1.

Lagrange interpoláció

Ismerünk n db alappontot. Ezekre a pontokra pontosan n polinomot kell meghatároznunk, majd ezeket összegeznünk. Így megkapjuk a Lagrange-polinomot ezekhez a pontokhoz. X_n-ben y_n-t vesz fel.

Az $y(x) = L_0(x)y_0 + L_1(x)y_1 + L_2(x)y_2 + \ldots + L_n(x)y_n$ egy interpolációs polinom, amelyben $L_i(x)$ i=0, 1, 2, …, n az úgynevezett Lagrange-féle polinomok. Ezek n-ed fokúak, és így az interpolációs képlet is n-ed fokú. Az interpolációs elv értelmében, ha $x=x_k$, akkor $L_i(x_k) = \begin{cases} 0, & \text{ha } i \neq k \\ 1, & \text{ha } i=k. \end{cases}$

tehát $L_i(x) = \frac{(x-x_0)*(x-x_1)*\ldots*(x-x_{i-1})*(x-x_{i+1})*\ldots*(x-x_{n-1})*(x-x_n)}{(x_i-x_0)*(x_i-x_1)*\ldots*(x_i-x_{i-1})*(x_i-x_{i+1})*\ldots*(x_i-x_n)}$

alakú (i=0, 1, 2,…, n). I számú Lagrange polinom összege:

$L_i(x) = \Sigma \frac{(x-x_0)*(x-x_1)*\ldots*(x-x_{i-1})*(x-x_{i+1})*\ldots*(x-x_{n-1})*(x-x_n)*y_i}{(x_i-x_0)*(x_i-x_1)*\ldots*(x_i-x_{i-1})*(x_i-x_{i+1})*\ldots*(x_i-x_n)}$

Ez a végleges Lagrange formula. Az adott n pont nem elegendő a pontosság szempontjából, akkor további pontokat veszünk, és a L(x) értékét át kell számolni.

Newton interpoláció???????????

A Lagrange akkor jó, ha van n db fix pontunk és erre szeretnénk polinomot illeszteni. Ha viszont új pontokat akarunk ezek után felvenni, akkor az egész Lagrange-polinomot újra kell számolni minden új pontra. Ez nagyon számításigényes.

A Newton-interpoláció esetében nem számoljuk újra az egész polinomot, hanem a meglévő Lagrange-polinomot felhasználjuk kisebb kiegészítéssel.

Legkisebb négyzetek módszere

A lineáris regresszió egy statisztikai módszer, amellyel előre megadott pontokhoz keressük az azokra legjobban illeszkedő egyenest. Nincs köze az interpolációhoz, csak hasonlóak. Van egy mintahalmazunk, ehhez keressük a legjobban illeszkedő egyenest. Feltételezzük, hogy két változó között lineáris kapcsolat van. A síkon a pontokra keressük a legjobban illeszkedő egyenest a legkisebb négyzetek módszerével. Gauss nevéhez fűzhető. A használatához

feltételezni kell, hogy a pontok között van kapcsolat. Az eltérések négyzetösszegét igyekszik minimalizálni.

A legkisebb négyzetek módszerénél x_1-x_n pontokban adva vannak a függvény értékei y_1-y_n, amelyek hibákat tartalmaznak. A célunk az, hogy ezeket a hibákat minimalizálva találjuk meg a legjobban illeszkedő függvény paramétereit.

b) Operációs rendszerek: Az operációs rendszerek funkciói, alapfogalmai. A virtualizáció. Processz kezelés, fájlrendszerek és szolgáltatásaik, hibatűrő diszk rendszerek. Jogosultsági rendszerek az operációs rendszerekben.

OS

Számítógépek alapprogramja, mely közvetlenül kezeli a hardvert és egy egységes környezetet biztosít a számítógépen futtatandó alkalmazásoknak. Mint a folyó a halaknak. Ütemezi a programok végrehajtását, elosztja az erőforrásokat, biztosítja a felhasználó és a számítógépes rendszer közötti kommunikációt.

Kernel

Az OS alapja, a hardver erőforrásainak kezeléséért felelős. Többfeladatos rendszer esetén ő szabja meg, hogy melyik program mennyi ideig használhatja a hardver egy adott részét. Háttérben futó, alapvető feladatokat ellátó program.

Processz

Folyamat, egy program memóriabéli futó példánya, amihez kapcsolódik a példány végrehajtásával kapcsolatos OS tevékenység. Egy processzt a címtartománya (memóriatérkép) és a processzustáblabéli információk alkotnak. A processznek szüksége van erőforrásokra, hogy elvégezze a feladatát.

OS három fő tevékenysége a processzek felügyeletével kapcsolatban:

- processz létrehozása és törlése
- processz felfüggesztése és újraindítása
- eszközök biztosítása a szinkronizációhoz és kommunikációhoz

Ütemező program: processzek ütemezését végzi, ellátja a processzort processzekkel. Megszakítás esetén a várakozási sorból kiválaszt egy másik processzt és megóvja a rendszert attól, hogy egy processz kisajátítsa a processzoridőt. Egy rendszerben az összes processz végrehajtódik, de egy időpillanatban egyszerre csak egy fut.

Virtualizáció

A fizikai gépen több virtuális gépet hozunk létre, mindre telepítünk OS-t és önálló gépként kezeljük őket. Meghatározhatjuk a memóriafelhasználást, CPU-teljesítményt, megosztoznak az erőforrásokon.

Típusai:

Bare-Metal: szerverek esetében. A fizikai hardverre először a virtualizációt végző szoftver a hypervisor kerül, ez kezeli a teljes erőforráskészletet. A virtuális gépek létrehozását és működtetését a hypervisor felügyeli.

Kliens oldali virtualizáció: a hardver egy hétköznapi OS-t használ, amin elindítunk egy virtualizációs szoftvert.

Virtualizáció megvalósítása

A virtuális gépekben futó OS-ek nem feltétlenül tudják, hogy virtualizált környezetben futnak, ehhez a virtualizálást végző szoftvernek alkalmazkodnia kell.

Erőforráskezelés módjai:

- szoftveres megoldás: a virtualizációs szoftver oldja meg
- paravirtualizációs megoldás: a virtuális gépben futó OS tudja, hogy nem fizikai gépen fut, és ennek megfelelően működik
- hardveres megoldás: a hardvert úgy fejlesztették, hogy képes legyen a megosztott működésre és több gép egyidejű kiszolgálására.

HALT: leállítja a fizikai gépet. Nem szerencsés, ha a virtuális gép ezt eredeti formájában alkalmazza. Megoldások:

- szoftveres: a szoftver vizsgálja a virtuális gép által kiadott utasításokat, és lecseréli olyan kódra, ami a fizikai helyett a virtuális gépre vonatkozik, ebben az esetben leállítás.
- paravirtualizációs: vendég OS érzékeli, hogy virtuálisan fut és a HALT helyett a hypervisor függvényét hívja meg
- hardveres: a CPU elkülönítetten kezeli a virtuális gépeket és képes ennek megfelelően kezelni a HALT-olt.

Fájlrendszerek

FAT: régi, viszont kis tárolókon gyors, Windows 95, 98. FAT32-őn nem lehet 4 GB-nál nagyobb fájlokat tárolni.

NTFS: Microsoft, alapértelmezett Windows-on. Merevlemezekkel kapcsolatos bizonyos hibákat automatikusan helyre tud állítani. Nagyobb merevlemezeket is támogat, lehetőség van engedélyek és titkosítás használatára. A fájlok elérhetősége a jóváhagyott felhasználókra korlátozható.

ext4: naplózó fájlrendszer, Linux

AFPS (új) és HFS+ (régi): Mac, elsősorban SSD-re és flash memóriára lett tervezve, nagy hangsúly a titkosításon.

RAID tömbök

Tárolási technológiák, amelyekkel az adatok elosztása vagy replikálása több fizikailag független merevlemezen, egy logikai lemez létrehozásával lehetséges. Minden RAID az adatbiztonság növelését vagy az adatátviteli sebesség növelését szolgálja.

RAID0: összefűzés. Nem redundáns, ezért nincs hibatűrés. Sebessége a leggyorsabb, ideális esetben a lemezek sebességének összege. A1A2A3A4-A5A6A7A8

RAID1: tükrözés. Ugyanaz az adat van mindkét lemezen. Olvasási sebesség párhuzamos, ezért gyorsabb, de az írás normál. Bármelyik meghajtó meghal, folytatódhat a működés. A1A2A3A4-A1A2A3A4

RAID2: használja az összefűzést, illetve egyes meghajtókat hibajavító kód tárolására tartanak fenn, vagyis az adatbitekből redundáns biteket képeznek. Megnövekedik a tárigény. Ezen meghajtók egy-egy csíkjában a különböző lemezeken azonos pozícióban lévő csíkokból képzett hibajavító kódot tárolják. Képes a lemezhiba detektálására és kijavítására. Manapság nem használják. A1B1C1D1-A2B2C2D2-A3B3C3D3-A4B4C4D4-Ap1Bp1Cp1Dp1-Ap2Bp2Cp2Dp2-Ap3Bp3Cp3Dp3

RAID3: teljes hibajavító kód helyett csak egy lemeznyi paritásinformáció tárolódik. Egy paritáscsík a lemezeken azonos pozícióban elheyezkedő csíkokból XOR művelettel kapható meg. Ha egy meghajtó kiesik, nem gond, mert a rajta lévő info az előző módon megkapható. Kisméretű csíkok vannak, ezért az írás és olvasás párhuzamosan történik. A1A4A7B1-A2A5A8B2-A3A6A9B3-Ap(1-3)Ap(4-6)Ap(7-9)Bp(1-3)

RAID4: RAID3, de nagyméretű csíkok vannak, egy rekord egy meghajtón van, ezért egyszerre több rekord párhuzamos írását, olvasását teszi lehetővé. De a paritás-meghajtó adott csíkját minden íráskor frissíteni kell, így ez lesz a szűk keresztmetszet. Ha egy meghajtó kiesik, az olvasási teljesítmény csökken. A1B1C1D3-A2B2C2D2-A3B3C3D3-ApBpCpDp

RAID5: paritás információ egyenletesen van elosztva a meghajtók között. Legalább 3 meghajtó kell, de az írási sebesség csökkenésének megakadályozása miatt 4 az ajánlott. Írás és olvasás párhuzamos. Hiba esetén az adatok sértetlenül visszaolvashatóak, a hibás meghajtó adatait a vezérlő ki tudja számolni.A1B1C1Dp-A2B2CpD1-A3BpC2D2-ApB3C3D3

RAID6: RAID5 kibővítve, nem csak soronként, hanem oszloponként is kiszámítják a paritást. Kétszeres meghajtómeghibásodást tudnak kiküszöbölni. Legalább 4 meghajtó, de 6 az ajánlott.

RAID01: RAID0 sebessége és a RAID1 biztonsága. Legalább 4 eszköz kell, melyekből 1-1-et összefűzve, majd páronként tükrözve kell felépíteni, ezért a kapacitás fele használható.

RAID10: először tükrözünk és utána fűzzük őket össze. Biztonság szempontjából jobb, mert a kiesés csak az adott tükrözött tömböt érinti. Sebességben azonosak.

Jogosultsági rendszerek: működés, azonosságok és különbségek az egyes rendszerekben

Minden többfelhasználós rendszerbe kell.

passwd fájl: felhasználókról tárol információkat. Login név, jelszó, userID, groupID, egyéb, home könyvtár, shell

shadow fájl: kódolt jelszavak, utolsó csere óta eltelt napok száma, figyelmeztetés jelszó lejáratról, inaktivitás

Csoportok kezelése: azért, hogy ne egyesével kelljen mindenkire.

chmod: jogosultságok beállítása egy fájlhoz

chown: fájl tulajdonosának beállítása, cseréje

chgrp: fájl csoportjának beállítása, cseréje

Jogosultságok leírása 3 oszlop: jogosultság, tulajdonos csoport. A jogosultságos 9 karakterből áll, felhasználó, csoport és más felhasználók 3-3-3.

Olvasás (2) (r) Írás (4) (w) Végrehajtás (7) (x)

Fájlokra és könyvtárakra vonatkoznak. Könyvtárnál a végrehajtás a könyvtárba való belépést és a fájlok elérését jelenti.

Példa: rwxr-x---: a tulajdonos olvashat, írhat, futtathat. A csoport olvashat, futtathat. Többi felhasználó semmit nem tehet vele.

7. tétel

a) Magasszintű programozási nyelvek II: Öröklődés. Korai kötés, késői kötés. Konstruktorok, konstruktorhívási lánc, osztályszintű konstruktor. Típuskompatibilitás, object osztály. Lepecsételt osztályok és statikus osztályok.

Öröklődés

A meglévő objektumokból levezetett újabb objektumok öröklik a definiálásukhoz használt alap objektumok egyes adatstruktúráit és függvényeit, de újabb tulajdonságokat is definiálhatnak vagy régieket újraértelmezhetnek. Öröklődés során a private mezők és metódusok kivételével átvesszük a dolgokat. Az öröklődés miatt csökken a kód redundanciája és olvashatóbb, elszeparálhatóbb lesz a kód.

Kötés

A fordítóprogram a függvényhívást konkrét megírt függvénnyel köti össze. Ez azért fontos, mert lehet több, ugyanolyan nevű függvény, más paraméterekkel. Sőt, olyat is lehet, hogy egy örökölt metódust felüldefiniáljunk ugyanazokkal a paraméterekkel és névvel, a new kulcsszóval.

Korai kötés

A fordítóprogram fordítási időben dönti el, hogy melyik metódust rendeli a metódushíváshoz, ezzel gyorsítva a program működését, mert nem futásidőben számol. Ha egy ősosztály (fegyver) és egy gyerekosztály (gépfegyver) is rendelkezik ugyanolyan nevű metódussal (lő), és írunk egy függvényt, amely egy Fegyver-t kér be, és kiírja a lövés metódusát, minden esetben az ősosztály lő metódusa fog lefutni, hiszen ez már fordítási időben eldőlt. Hiába adunk meg függvényhíváskor egy gépfegyver példányt, akkor is a fegyver.lő fut le. Azért, mert Fegyver-t kért be. Ha GépFegyver-t kért volna be, akkor a gépfegyver.lő futna le.

Késői kötés

A gyerekosztályban az ugyanolyan nevű, megegyező paramterű metódust a láthatósági szintjének megadása után az *ovveride* kulcsszóval kell ellátnunk, illetve az ősosztály ugyanezen metódusát a *virtual* kulcsszóval. Ha maradunk az előző példánál, és Fegyver-t kérünk be, akkor meghíváskor, ha paraméterben már GépFegyver-t adunk át, akkor a gépfegyver.lő metódus fog lefutni.

Konstruktorok, hívási lánc

Példány létrehozásáért felelős metódus. Nincs visszatérési értéke, de mondhatjuk azt is, hogy a visszatérési értéke egy példány. Minden osztálynak van egy publikus, üres konstruktora, ha mi nem írunk bele, akkor is. Ha privát konstruktort írunk, az osztályt nem lehet példányosítani, legfeljebb szingletonnal tudunk belőle elkérni egy példányt, amit létrehozhatunk mohó vagy lusta kiértékeléshez. A konstruktorok leginkább a mezők beállítására valóak. Egy osztálynak több konstruktora lehet.

Szintaxis: <láthatósági szint><osztály neve> (formális paraméterlista) {törzs}

Példa: Kutya kutya = new Kutya();

Konstruktorhívási lánc

Abból indulunk ki, hogy nem elegendő csak a gyerekosztály konstruktorának lefutnia, mert az ősosztálynak lehetnek privát mezői, amiket a gyerek nem tud beállítani, mert nem látja őket. Az ősosztály konstruktorai közül is legalább egynek le kell futnia példányosításkor, több szint esetén mindegyik szintű ősosztálynál igaz ez. Az osztályhierarchiában fentről lefelé haladva futnak le a konstruktorok, az őstől a konkrét gyerek felé haladva. Az ős beállítja a mezőit (privátot is), majd jön egy gyerek, ha akarja felüldefiniálja (privátot is tudja, ha van megfelelően definiált örökölt metódushívás).

Probléma, ha a fordító nem tud választani paraméter nélküli konstruktort a felső szintről automatikusan, mert nem létezik ilyen. (De igen, mert mindig van ilyen.) De ha mégse, akkor a base()-zel lehet jelezni, hogy melyik paraméterezésű ős konstruktort akarjuk választani.

base() akkor hasznos még, ha az ősosztálynak a paraméter nélküli konstruktora private. Ilyenkor a gyerekosztályban ki kell választani base()-zel egy nem private paramétereset az ősosztályból.

base(): explicit módon választunk egy ősosztálybeli konstruktort.

this(): meghívhat egy másik saját konstruktort, ekkor az ősosztályból a konstruktorválasztás problémája elodázódik.

Osztályszintű konstruktor

Static kulcsszó kell, nem lehet paramétere és nincs védelmi szintje. Akkor érdemes, ha osztályszintű mezőket akarunk beállítani. Meghívása automatikus. A rendszer garantálja, hogy az első ráhivatkozás előtt lefut a konstruktor.

Típuskompatibilitás

Az öröklődés következménye is lehet: egy gyerekosztály kompatibilis az ősosztályával, azaz minden mezőt tartalmaz, amit az ős.

Tranzitivitás: ha „A” osztály kompatibilis egy „B” osztállyal, és a „B” osztály kompatibilis a „C” osztállyal, akkor „A” is kompatibilis „C”-vel. C#-ban minden osztály őse az *object* osztály, minden osztály kompatibilis vele. Az elemi típusok is azok vele (int, double, string).

is operátor: típuskompatibilitási vizsgálatra jó

Dog dog = new Dog();

if(dog is Animal) {}

as operátor: ha a késői kötés nem lehetséges a virtual kulcsszó kihagyása miatt, akkor használhatjuk: (dog as Dog)

Object osztály

Támogatja a .NET osztályhierarchia összes osztályát és alacsonyszintű szolgáltatásokat nyújt a származtatott osztályok számára. Ez az összes .NET-osztály végső bázisosztálya, a típushierarchia gyökere.

Lepecsételt osztályok

Zárt osztály, amelyből nem lehet örököltetni. A *sealed* használható property-re és mezőre is, ha késői kötésű. Ekkor ezeket nem lehet felülírni.

Statikus osztályok

Csak statikus metódusokat és mezőket tartalmazhatnak, nem lehet őket példányosítani. Példányszintű konstruktor nem lehet benne. Kiterjesztő metódusokat írunk bele általában.

b) Adatbázisrendszerek I: SQL nyelv. Relációsémák definiálása. Indexek. Táblák módosítása. SELECT parancs. Beágyazott lekérdezések. Több táblára vonatkozó lekérdezések. Privilégiumok. Szerepkörök. Tranzakció kezelés. ROLLBACK, COMMIT

SQL nyelv

Lekérdező nyelv, amelyet több relációs adatbáziskezelő rendszer ismer. IBM dolgozta ki a DB2 relációs adatbáziskezelőhöz.

- nem algoritmikus: nem tartalmaz elágazást, ciklust, stb.
- halmaz-orientált: deklaratív: nem kell definiálni a művelet végrehajtásának lépéseit, hanem a feladat nem eljárásszerű megfogalmazását kell megadni, melyek a reláció(k) kiválasztott sorain hajtódnak végre. Az optimális megoldás megtalálása a nyelvi CPU feladata, nem a programozóé.
- nem rekurzív

Felhasználása

- önálló SQL: csak a nyelv utasításai állnak rendelkezésre
- beágyazott SQL: 3rd gen algoritmikus nyelvbe (C, PL/SQL, stb.) ágyazva alkalmazzuk

Elemei

- **DDL:** adatdefiníciós nyelv, az adatbázisban a sémák definiálásáért és módosításáért felel
    - CREATE, DROP, ALTER, TRUNCATE
- **DML:** adatmanipulációs nyelv, az adatokat kezeli az adatbázisban
    - UPDATE DELETE INSERT
- **QL:** lekérdező nyelv, lekérdezésekért fele
    - SELECT
- **DCL:** adatvezérlő nyelv, szerepköröket és jogosultságokat manipulál
    - GRANT, REVOKE
- **TCL:** tranzakciókezelő nyelv, tranzakciókat kezeli
    - COMMIT ROLLBACK

Relációsémák definiálása

Séma: több adattábla gyűjtésére szolgál. Egy sémában táblát CREATE TABLE paranccsal tudunk létrehozni. A táblán belül mezőket hozhatunk létre (INT, DATE, VARCHAR(max_méret)[nem fix mérete van], CHAR(méret)[fix mérete van], LOB(large object)[bináris típusok](BLOB, TINYBLOB, MEDIUMBLOB, stb.)

Minden táblához kell elsődleges kulcs (egy vagy több mező). PRIMARY KEY mező után írva, vagy tábla alján hozzáadva CONSTRAINT-ként. CONSTRAINT táblaneve_PK PRIMARY KEY(mezőneve).

Megszorítások

- CHECK: adott mező értéke megfelel-e a feltételeknek, ha igen lehet módosítani vagy beszúrni
- UNIQUE: adott mező értéke egyedi minden rekord esetében
- NOT NULL: adott mező értéke nem lehet null

- AUTO_INCREMENT (MySQL) vagy IDENTITY(MsSQL): sorszám, új rekord beszúrásánál ++
- FOREIGN KEY: adott mező idegen kulcs lesz. Ilyenkor ez az érték egy másik táblában egy rekord kulcsának az értéke lesz.
- DEFAULT: ha adott mező nem kap értéket, akkor ez lesz az alapértelmezett értéke

Indexek

A táblában való gyors keresésre való. Ha nem lenne, az egész táblát be kéne járni keresésnél. Oszlophoz rendeljük, lehetőleg azokhoz, amelyek gyakran szerepelnek keresésben.

CREATE [UNIQUE] INDEX IndexNév ON TáblaNév (Oszlop1, Oszlop2, …);

Táblák módosítása

Tábladefiníció módosítása DDL

- ALTER TABLE paranccsal, hozzá tartozó dolgok:
- ALTER TABLE TáblaNév ADD COLUMN Oszlop INT;
- ALTER TABLE TáblaNév MODIFY COLUMN Oszlop INT;
- ALTER TABLE TáblaNév DROP COLUMN Oszlop INT;
- ALTER TABLE TáblaNév ADD CONSTRAINT PK PRIMARY KEY(id);
- ALTER TABLE TáblaNév DROP CONSTRAINT PK;

Adatok manipulálása DML

Új rekordok beszúrása: konkrét érték, vagy lekérdezés (rekord másolása)

INSERT INTO TáblaNév (oszlopnév1, oszlopnév2) VALUES (érték1, érték2)

INSERT INTO TáblaNév SELECT oszlop1, oszlop2 FROM tábla2;

Meglévő rekord módosítása:

UPDATE tábla1 SET oszlop1 = érték1, oszlop2 = érték2 WHERE [feltétel];

Meglévő rekord törlése:

DELETE FROM tábla1 WHERE [feltétel];

Összes rekord törlése: TRUNCATE TABLE

SELECT

Tábla, vagy annak oszlopainak lekérdezésére szolgál.

- SELECT * FROM táblanév: minden adat lekérdezése a táblából
- SLECT oszlop1, oszlop2 FROM táblanév: adott oszlopok adatainak lekérdezése

Oszlopnév után adható alias az AS kulcsszóval, ez lesz az oszlop fejléce: SELECT oszlop1 as név..

WHERE: feltétel megadása, logikai kifejezés lehet benne

Különleges logikai kifejezések:

- <oszlopkifejezés> BETWEEN <alsóérték> AND <felsőérték>
- <érték> IN <lista>

- <oszlopkifejezés> LIKE <karakterlánc> (LIKE: %: több karakter helyettesítése, _: 1 karakter helyettesítése

GROUP BY

Eredménytábla rekordjainak csoportosítása. A megadott oszlopnevek sorrendje a prioritás. Először az 1. oszlop szerint csoportosít, aztán ha van olyan, amelyeknél ez egyezik, akkor a második szerint, stb. Csoportokra feltétel: HAVING, például count > 5

ORDER BY

Eredménytábla rekordjait rendezhetjük valamely mező értéke szerint. A megadott oszlopnevek sorrendje a prioritás. Ezután megadhatjuk a rendezés irányát (növekvő ASC, csökkenő DESC).

Beágyazott lekérdezések

Lekérdezés feltételében is lehet további lekérdező parancsot használni: külső és belső SELECT.

Esetei:

- a belső lekérdezés egyetlen értéket szolgáltat. A feltételként megadott kifejezésben a belső lekérdezés által szolgáltatott értéket használja a rendszer.
- a belső lekérdezés egyoszlopos relációt szolgáltat. Ekkor olyan feltételeket adhatunk meg, amelyek a belső lekérdezés által szolgáltatott oszlop adatait használja fel. Ebben az esetben predikátumokat használhatunk.

Az oszlopkifejezés aggregáló függvényeket tartalmazhat: COUNT, SUM, AVG, MIN, MAX.

Predikátumok

- <oszlopkifejezés> [NOT] IN <belső lekérdezés> Az oszlopkifejezésben megadott érték szerepel-e a belső lekérdezés által előállított oszlop adatai között.
- [NOT] <oszlopkifejezés> <reláció> ALL|ANY <belső lekérdezés> Az oszlopkifejezésben megadott értékre teljesül-e a megadott reláció a belső lekérdezés által előállított oszlop adataira. ALL: akkor lesz a feltétel igaz, ha minden elemre teljesül, ANY: elég egyre teljesülni.
- [NOT] EXISTS <belső lekérdezés>: a belső lekérdezés által visszaadott reláció üres-e vagy sem.

Több táblára vonatkozó lekérdezések

Akkor van, ha kapcsolatok állnak fenn a táblák között.

SELECT * FROM tábla1, tábla2; több táblából minden egyes rekordot lekérdezünk (Descartes-szorzat)

Bonyolultabb esetekben JOIN-t használunk, és valamilyen feltétel alapján kapcsoljuk össze a táblákat.

Fajtái

- <tábla1> INNER JOIN <tábla2> ON <feltétel>: azokat a rekordokat adja vissza, amelyekre a feltétel igaz és a kapcsolathoz használt idegen kulcs mezőjének értéke nem NULL

- <tábla1> LEF/RIGHT JOIN <tábla2> ON <feltétel>: iránytól függően az egyik táblából minden rekordot megkapunk, azokat is, amikre a feltétel nem igaz, vagy NULL a kapcsolt mező értéke. LEFT: bal táblából mindent, jobb táblából csak azokat, amikre a feltétel igaz
- <tábla1> FULL OUTER JOIN <tábla2> ON <feltétel>: mindkét oldali táblából megkapjuk azokat a rekordokat, amelyekre a feltétel igaz

Privilégiumok

A felhasználó számára lehetővé teszi műveletek végrehajtását.

Típusai

- rendszer-privilégiumok: adatdefiníciós és adatvezérlő parancsok végrehajtását, és adatbázisba való belépést tesz lehetővé
- objektum-privilégiumok: az adatbázis objektumaival való műveletekhez adnak jogosultságot

Felhasználó létrehozás után nem rendelkezik privilégiumokkal.

Privilégium megadása:

GRANT <privilégium> TO <user> IDENTIFIED BY <jelszó>

REVOKE <privilégium> FROM <user>

Szerepkörök

CONNECT

- bejelentkezhet az Oracle-be és használhatja
- betekinthet táblákba, amelyekre SELECT jogot kapott
- betekinthet a PUBLIC minősítésű táblákba
- az adatmanipulációs utasításokat használhatja azokra a táblákra, amelyekre annak tulajdonosa adott neki INSERT, DELETE UPDATE jogokat
- nézettáblákat (view) definiálhat

RESOURCE

- minden CONNECT jog
- táblák, indexek létrehozása és törlése
- az általa létrehozott táblákra jogokat adhat más usereknek
- az általa létrehozott táblákra, indexekre használhatja a renszer AUDITING szolgáltatását (diagnosztizálás, pl. megtekintheti, hogy kik milyen műveleteket hajtottak végre a tábláin)

DBA

- minden RESOURCE jog
- bármely felhasználó adataiba betekinthet és lekérdezéseket hajthat végre
- jogokat adhat és vonhat vissza bárkitől
- PUBLIC-nak minősíthet adatokat
- rendszer UDITING
- teljes adatbázis EXPORT/IMPORT (DUMPolás)

Tranzakció kezelés

A tranzakció az adatbázis-műveletek végrehajtási egysége, amely DML-béli utasításokból áll és ACID tulajdonságokkal rendelkezik:

- Atomosság (atomicity): a tranzakció mindent vagy semmit jellegű végrehajtása
- Konzisztencia (consistency): az a feltétel, hogy a tranzakció megőrizze az adatbázis konzisztenciáját, azaz a tranzakció végrehajtása után is teljesüljenek az adatbázisban előírt konzisztenciamegszorítások (integritási megszorítások), az az az adatelemekre és a közöttük lévő kapcsolatokra vonatkozó elvárások
- Elkülönítés (isolation): minden tranzakciónak látszólag úgy kell lefutnia, mintha ezalatt az idő alatt semmilyen más tranzakciót nem hajtanánk végre
- Tartósság (durability): ha egyszer egy tranzakció befejeződött, akkor már soha többé nem veszhet el a tranzakciónak az adatbázison kifejtett hatása

COMMIT

A DBMS véglegesíti a tranzakció által előidézett változásokat.

ROLLBACK

ROLLBACK: visszagörgeti (meg nem történtté teszi) a tranzakció során elvégzett adatmódosításokat. Ha volt COMMIT, nincs rá lehetőség.

8. tétel

a) Magasszintű programozási nyelvek II: Absztrakt metódusok és osztályok. Interface-ek, többszörös öröklődés. Kivételkezelés filozófiája és megvalósítása. Generikusok használata és készítése. Delegate-ek és események. Lambda kifejezések.

Absztrakt metódusok és osztályok

Cél az alá tartozó osztályok feladatainak absztrakt megfogalmazása. Nem példányosítható, nem tartalmaznak konkrét elemeket.

Nem konkrét elemek: abstract metódusok, propertyk. Nincs törzsük, csak szignatúra (proerty-nél megadható, hogy get, set legyen-e). A szignatúra a láthatóság, a visszatérési típus, név és formális paraméterlista együttese, valamint az osztály- vagy példányszintű jelzők is ide kerülnek.

- Ha egy osztálynak van legalább 1 abstract metódusa, absztrakttá kell tenni.
- Ha egy nem abstract osztályt egy abstractból származtatunk le, és szerepelnek abstract elemek az ősben, akkor a gyermeknek azokat ki kell dolgoznia, kivéve ha ő maga is abstract.
- Osztályszintű metódus nem lehet abstract, mert nincs értelme.

Interface-ek

Megmondja, hogy az őt implementáló osztályok milyen felületet valósítsanak meg, de azt nem, hogy hogyan tegyék azt. C#-ban metódus és property szignatúrákat tartalmazhat. Minden tag public, nem is írhatjuk ki.

- Ha egy abstract osztály valósít meg egy interface-t, akkor nem kötelező kidolgozni metódusait, ha azok szintén abstract kulcsszóval vannak ellátva. Ha nem abstract, akkor igen.
- Hasonló az abstract osztályhoz, de nem tartalmazhat kidolgozást (implementációt).
- Egy osztálynak csak egy őse lehet, de bármennyi interface-t implementálhat. Ezzel szimulálható a többágú öröklődés.
- Ha egy osztály implementál egy interface-t, akkor az osztály példányait az interface típusú referenciában is tárolhatjuk, mert kompatibilisek egymással.

Kivételkezelés

Kivétel: esemény, amikor a vezérlés nem megfelelő állapotba kerül. Vezérlés megszakad, visszalép a hívási láncon addig, amíg nem találja meg a hiba lekezelését, ha ilyen nincs, akkor a belépési pontig és a program leáll.

Throw: kivételdobás kulcsszava. Példányosítani kell utána egy kivételt, C#-ban erre vannak külön osztályok: Exception-ből származnak. Sajátot is írhatunk, akkor is ebből kell származtatnunk. Ekkor kötelezően konstruktort kell írni és az ős konstruktorát hívni.

Try-catch-finally

- try: kritikus kód, ahol kivétel lehet
- catach: paraméterben megkapja a kivétel típusát, és ha az fellép a try blokkban, akkor ez fut le. Lehet több catch egyszerre. Ekkor a kivételek hierarchiája és a kivétel típusa

alapján dől el, hogy melyik ág fut le. Ezért a catch-ek sorrendje a legkonkrétabb kivétellel kezdődik és így haladunk az általánosak felé.
- finally: ha volt kivétel, ha nem, lefut, mert pl. erőforrást szabadít fel fájlkezelésnél.

Generikusok

A generikus típus valaminek a megvalósítása anélkül, hogy tudnánk a típusát. Előre megírunk valamit, aminek egy nagyon általános viselkedést szabunk meg, de az, hogy milyen típus lesz az osztály nem tudjuk. Ha nem lenne, minden típusra külön osztályt kellene létrehozni, és ez rengeteg hasonló kódot eredményez. Ráadásul módosítani kell az összeset, ha úgy alakul.

Szintaxis class OsztalyNeve<T>. Lehet több paramétere <T, M>

Generikus osztályok: List, Dictionary, Stack, Queue, LinkedList.

Delegate-ek

Metódusreferencia. Valamilyen kódra vagy metódusra mutat. A mutatókat szokás függvénymutatóknak hívni, de C#-ban külön objektumként kezeljük őket.

Metódust (kódot) tudunk átadni másik metódusnak paraméterként. Ilyenkor a paraméterként átadott kódot callback-nek hívjuk. C#-ban definiálhatunk saját delegate-eket a delegate kulcsszóval, szignatúrával és névvel névtérben vagy osztályban. Ezután használhatjuk t apusként.

Beépített generikus delegate-k:

- Action<T1, T2,…>: eljárást tárolhatunk vele (nincs visszatérési típus), a T1, T2, … a paraméterek típusai
- Func<T1, T2,…, K>: függvényt tárolhatunk vele, a T-k a paraméter típusok, K a visszatérési típus
- Predicate<T>: egy paramétere van, visszatérési típusa logikai

Események

Gyakori feladat, hogy több függvényt kell meghívni egymás után egyszerre egy esemény miatt, ezért csinálták az event-et. Egy ilyen módosítóval ellátott változó nem egy, hanem tetszőlegesen sok függvény memóriacímének tárolására képes, vagyis egy lista.

event Action<int> eseménynév //létrehozás

eseménynév += metodus1 //feliratkozunk a metuds1-gyel

eseménynév(); //lefuttatjuk az összes delegate-et

Lambda kifejezések

Adatbázis vagy adatszerkezetek irányába lekérdezések készítése. Alapja a LINQ, Language Integrated Queryy.

string[] names = {„Tom”, „Dick”, „Harry”}

string query = names.Where(n => n.Contains(’i’);

string query = names.Where(n=> n.Contains(’s’).OrderBy(n=> n.Length);

b) Fordítóprogramok: A fordítóprogramok alapjai, lexikális elemzés, és a reguláris nyelvek, szintaktikai elemzők (Rekurzív leszállás módszere, LR(k), LL(k) elemzők, táblázatos elemzők működése), szemantikai elemzés kérdései. A program fordítás lépései.

Fordítóprogramok szerkezete

- Source-handler: a forrásnyelvű programot a fordítás számára könnyen hozzáférhető karaktersorozattá alakítja (whitespace, új sor, kocsivissza törlése)
- Compiler
    - lexikális elemző: inputja a source-handler ouputja. Reguláris, azaz Chomsky 3-as nyelvet használ. Feladata felismerni az adott nyelv lexikális elemeit, majd helyettesíteni azokat szintaktikai elemző számára érthető jelekkel.
    - szintaktikai elemző: eldönti, hogy a lexikális elemzőtől kapott szimbólumsorozat szintaktikailag helyes-e. Környezetfüggetlen grammatikát használ (Chomsky 2).
    - szemantikai elemző: szemantikai hibákat keres. Környezetfüggő grammatikát használ (Chomsky 1)
- Code-handler: elhelyezi a tárgykódot a háttértáron

Lexikális elemzés

A forrásnyelvű programból a source-handler által készített karaktersorozatban a szimbolikus egységeket felismeri, vagyis meghatározza a szimbólum szövegét és típusát. A nyílt forrásszövegből generál szimbólumsorozatot a szimbólumtábla segítségével operátorok, fenntartott szavak, azonosítók stb. formájában.

Szimbólumtábla: tartalmazza a programszövegben előforduló szimbólumok nevét és jellemzőit. Egy adott szimbólum minden előfordulása esetén a táblában keresést vagy beszúrást kell elvégezni. A táblát legtöbbször láncolt listával valósítják meg.

Egy sor tartalma

- szimbólum neve
- attribútumok
- definíciós adatok: mit azonosít a szimbólum (változó, kontans). Konstansnál az érték, típusnál a típusdescriptorra mutató pointer, eljárásnál a paraméterek és azok száma.
- típus
- programbéli cím (tárgyprogram): ezt kell a tárgyprogramba beépíteni a szimbólumra v aló hivatkozáskor.
- forrásnyelvi sorszám: explicit definíciónál a definíció sora, implicitnél az első előfordulás – hivatkozott sorszám, láncolási cím: a hivatkozási lista elkészítéséhez

Lexikális hiba: az elemző nem ismer fel egy tokent (int 2ab) helytelen. int 2 szintén helytelen, de ez már szintaktikai hiba

Reguláris nyelvek

Reguláris nyelvek, 3: egy nyelv ilyen, ha a nyelvet generáló nyelvtan minden szabálya A → a és A → Ba, vagy A → a és A → aB alakú: jobbreguláris, balreguláris. Lexikális elemzésre alkalmas. A szó felépítése egy irányban halad. Először a szó bal oldala alakul ki és az irány nem változik, mert a csere során újabb terminális szimbólumot teszünk a generált szóba, és egy

újabb nemterminálist a terminális jel jobb oldalára. A köztes állapotokban nemterminális jel csak egy lehet és mindig a köztes szó jobb végén. Nincs kitétel a S → ε szabályra, mert a generálás bármikor abbahagyható.

Szintaktikai elemzők

Eldöntik az adott programozási nyelven írt programról, hogy az nyelvtanilag helyes-e, vagyis, hogy megfelel-e a nyelvi szabályoknak. A szintaktikus elemző feladata valamely jelsorozat levezetési fájának előállítása.

Rekurzív leszállás

Minden szabályhoz eljárást rendel, meghívja, majd ezek sorban meghívják egymást. A szalagon lévő adatot és a szabályt egyszerre lépteti. A függvény megmondja, hogy mi van a szabályban. Ha ugyanaz, ami kell, akkor léptet, ha nem, akkor hibát ad. Nem lehet generálni, ezért nem elterjedt módszer.

Ha az i egy globális változója a programnak és az input szalagon balról jobbra van felírva az elemzendő szöveg és rendelkezésünkre áll az összes szabályhoz tartozó eljárás, akkor csak meg kell hívnunk a nyelvtan startszimbólumához tartozó eljárást, amely rekurzívan, vagy csak szimpla hívással meghívja az összes többi szabályhoz tartozó eljárásokat.

Ha a hívási láncban az összes eljáráshívás lefutását követően visszatérünk a start szimbólumhoz, akkor az elemzett szó helyes, egyébként a hiba helye minden szabály eljárásában adott az i index által.

LR(k)

Az LR elemzők bottom-up elemzők. Az elemzendő szimbólumsorozatból indul ki, és a cél a kezdőszimbólum elérése. Így építi fel a szintaxisfát. A kezdőszimbólum lesz a gyökér, a fa levelein pedig az elemzendő programszöveg terminális szimbólumai vannak.

LL(k)

Az LL elemzők top-down elemzők. A kezdőszimbólumból indul ki, ami a gyökér lesz, és a cél, hogy a szintaxisfa levelein az elemzendő programszöveg terminális szimbólumai legyenek. A szövegben balról jobbra haladunk, arra törekszünk, hogy az elemzéskor kapott mondatformák bal oldalán a lehető leghamarabb megjelenjenek az elemzendő szöveg terminálisai.

Táblázatos elemzők

Az elemzendő terminális sorozat xay, ahol az x szöveget már elemeztük. Felülről lefelé legbaloldalibb helyettesítéseket alkalmazunk, szintaktikus hibát nem találunk, így az elemzendő mondatformánk xYa, azaz xBa vagy xba alakú.

xBa esetén: a szintaxisfa építésekor a B-t kell helyettesítenünk egy B → ß szabállyal, amely re igaz, hogy a eleme Első(ß Követő(B)). Ha van ilyen szabály, akkor pontosan egy van, mert LL a nyelvtan, egyébként szintaktikai hiba van.

xba esetén: a mondatforma következő szimbóluma a b terminális jel, tehát az elemzendő szövegben is b-nek kell szerepelnie. Ha így van továbblépünk, ha nem, akkor szintaktikai hiba van.

Az elemző állapotait (ay#, Xa#, v) hármassal írjuk le, ahol # az elemzendő szöveg vége és a verem alja, ay# a még nem elemzett szöveg, Xa# az elemzendő szöveg mondatformájának még nem elemzett része a veremben, v a szabályok sorszámát tartalmazó lista. Mindig a verem tetején lévő X-et hasonlítja össze a még nem elemzett szöveg következő szimbólumával (a: aktuális szimbólum).

Kezdőállapot: (xay#, S#, ε). Az elemző működése állapotátmenetekkel adható meg. A sikeres végállapot (#, #, w).

Szemantikai elemzés

Meghatározza az elemzendő szöveg szintaxisfáját. A szintaxisfa pontjaihoz olyan attribútumokat rendelünk, amelyek leírják az adott pont tulajdonságait. Ezeknek az attribútumoknak a meghatározása és az attribútumok konzisztenciájának vizsgálata a szemantikai elemzés feladata. Olyan tulajdonságok vizsgálatával foglalkozik, amelyek statikusak, nem írhatók le környezetfüggetlen nyelvtannal.

- változók deklarációja, hatásköre, láthatósága, többszörös deklarációja, deklaráció hiánya
- operátorok és operandusok közötti típuskompatibilitás
- eljárások, tömbök formális és aktuális paraméterei közötti kompatibilitás
- túlterhelések egyértelműsége

Fordítás lépései

Forrás-kezelő (source handler) → lexikális elemző → szintaktikai elemző → szemantikai elemző → belső reprezentáció → kódgenerátor → optimalizáló → kód-kezelő → tárgyprogram

9. tétel

a) Formális nyelvek: Ábécék, szavak, formális nyelvek. Műveletek szavakkal és nyelvekkel. Szintaxisleíró eszközök. Generatív grammatikák, Chomsky-féle osztályozás. Levezetési fák, elemzési stratégiák. A véges és a verem-automaták, a Turing gépek és változataik ismertetése. A delta leképezés tulajdonságai, megadási módjai a különböző automaták esetén. Az automaták és a grammatikák kapcsolata.

Ábécék, szavak, formális nyelvek

ABC: véges, nem üres halmaz, elemei a jelek (karakter, betű, szimbólum).

Szavak: ABC jeleiből alkotott kifejezések. Üres szó: ε

AxA {a,b}: kettő hosszúságú szavak

Lezárt V*: végtelen hosszúságú szavak, az összes kirakható szó

Pozitív lezárt V^+: V*, de nincs benne ε

Formális nyelvek: Legyen $V^{-1} \leftarrow A$. Egy $l \subseteq V^*$ halmazt az „A” ABC fölötti formális nyelvnek nevezünk. Vagyis a formális nyelv egy adott ABC jeleiből alkotott tetszőleges hosszú szavak halmazának részhalmaza.

Egy formális nyelv megadható: felsorolással, szavakat alkotó szabály szöveges vagy matematikai leírásával, generatív grammatikával

Műveletek szavakkal és nyelvekkel

Konkatenáció (szavak szorzása): 2 vagy több szóból egy új szót képzünk, vagyis egymás mellé illesztjük őket. Jele a +. Ha α = „alma” és β = „fa”, akkor α + β = „almafa” β

Tulajdonságok

- asszociatív: (a+b)+c = a+(b+c)
- nem kommutatív: ab != ba
- van neutrális eleme: ε α = α ε

Megállapítások

- $\alpha^0 = \varepsilon$, bármely szó 0. hatványa üres szó
- $\alpha^n = \alpha + \alpha^{n-1}$ (n >= 1): bármely szó n. hatványa a szó n-szeres konkatenációja
- Ha az α a szó prefixuma/suffixuma és α hossza nem nulla, akkor valódi prefixum/szuffixum

Szavak hatványozása: n-szeres konkatenációja

- $\alpha^0 = \varepsilon$, bármely szó 0. hatványa üres szó
- $\alpha^n = \alpha + \alpha^{n-1}$ (n >= 1): bármely szó n. hatványa a szó n-szeres konkatenációja
- α primitív, ha egyetlen másik szónak sem hatványa
- α és β egymás konjugáltjai, ha létezik α = γδ, és β = δγ
- egy α szó lehet periodikus: 123123123123

Szavak tükrözése: palindrom. abc-cba

Műveletek ABC-kel, vagyis nyelvekkel

komplexus szorzás: A * B := {ab|a ∈ A, b ∈ B}: két ABC komplexus szorzata egy olyan ABC, melynek karakterei kettős jelek, egyik az egyikből, másik a másikból

A^0:= ε: minden ABC 0. hatványa az a halmaz, amelynek egyetlen eleme az ε

hatvány: ABC n. hatványa egy olyan ABC, melynek minden eleme n karakter hosszú

Szintaxisleíró eszközök

Szintaxisdiagram: nem precíz eszköz, de látványos és gyorsan megérthető szintaxisfát eredményez. Jellemző felhasználási területe a programozási nyelvek szintaxisának leírása.

EBNF forma: nem grafikus, hanem karakteres jelölésrendszert használ. Programozási nyelvek és parancssori operációs rendszerbeli parancsok definiálására is jó.

Generatív grammatikák

Előállítja az összes lehetséges szót. Nem felsorolás, hanem megnézi, hogy generálható-e.

Egy G(V, W, S, P) formális négyest generatív grammatikának nevezzük, ahol:

- V : a terminális jelek ABC-je
- W : a nemterminális jelek ABC-je
- V ∩ W = , vagyis a két halmaz diszjunkt, nincs közös elemük
- S : S ∈ W egy speciális nemterminális jel, a kezdőszimbólum
- P : helyettesítési szabályok halmaza.

Ha V := a, b és W := S, B, és P := (S, aB),(B, SbSb),(S, aSa),(S, ε, akkor a G := (V, W, S, P) négyes egy nyelvet ír le. Ennek a nyelvnek a szavai "a" és "b" jelekből írhatók fel. Az "S" és "B" jelek nem szerepelnek a végső szavakban, csak a szavak létrehozása közben mintegy "segédváltozók", köztes állapotokban használt jelek. A segédváltozók jelei csupa nagybetű (S, B), míg a nyelv terminális jelei csupa kisbetűvel vannak jelölve (a, b).

Chomsky-féle osztályozás

Két azonos nyelv esetén a terminális jeleknek meg kell egyezniük (ha nem egyeznek meg, akkor ugyanazokkal a szabályokkal nem lehet a szavakat előállítani), a nemterminálisoknak nem kell. A szabályok halmaza a legfontosabb, itt lehetnek a nagy eltérések. A formájuk fontos, mert ezek alapján különböztetjük meg és tudjuk csoportosítani a nyelveket (nyelvtanok és grammatikákat). Ezek alapján 4 nagy csoport létezik:

Mondatszerkezetű nyelvek, 0: egy nyelv ilyen, ha a nyelvet generáló nyelvtan minden szabálya αAβ → γ alakú. A szabályrendszer megengedő. Tagjai a beszélt, élő nyelvek. Szabálytalan szabályok vannak benne. A fordítás sok erőforrást igényel.

Környezetfüggő nyelvek, 1: egy nyelv ilyen, ha a nyelvet generáló nyelvtan minden szabálya αAβ → αωβ alakú, és megengedett az S → ε szabály használata. Szemantikai elemzésre alkalmas, mert ha jelentést akarunk elemezni, ismernünk kell a környezetet. Fontos, hogy a nemterminális szimbólum helyettesítése után a szimbólum környezete ne változzon meg.

Környezetfüggetlen nyelvek, 2: egy nyelv ilyen, ha a nyelvet generáló nyelvtan minden szabálya A → ω alakú, és megengedett a S → ε szabály használata. Szintaktikai elemzésre

alkalmas. A szabályok bal oldalán kizárólag egy nemterminális jel állhat, amely lehetővé teszi, hogy a programozási nyelv szabályait definiálhassuk. Környezetfüggetlen, ezért egyszerű és gyors. Amit elemeztünk és helyes volt, eldobhatjuk, mert nem kell többé.

Reguláris nyelvek, 3: egy nyelv ilyen, ha a nyelvet generáló nyelvtan minden szabálya A → a és A → Ba, vagy A → a és A → aB alakú: jobbreguláris, balreguláris. Lexikális elemzésre alkalmas. A szó felépítése egy irányban halad. Először a szó bal oldala alakul ki és az irány nem változik, mert a csere során újabb terminális szimbólumot teszünk a generált szóba, és egy újabb nemterminálist a terminális jel jobb oldalára. A köztes állapotokban nemterminális jel csak egy lehet és mindig a köztes szó jobb végén. Nincs kitétel a S → ε szabályra, mert a generálás bármikor abbahagyható.

Levezetési fák, elemzési stratégiák

Környezetfüggetlen nyelvtan segítségével eldönthető az adott szóról, hogy generálható-e az adott nyelvtannal. Ennek eldöntéséhez használjuk a levezetési fát. Ha balról jobbra, fentről lefelé olvasva megjelenik a szó, akkor generálható.

A levezetés kanonikus, ha mindig a legbaloldalibb nemterminális szimbólumot helyettesítjük tovább.

A levezetés top-down, ha fentről lefelé, a start szimbólumból kiindulva konstruáljuk.

A levezetés bottom-up, ha alulról felfelé elemezzük a szabályok behelyettesítésével.

Hossz nemcsökkentő stratégia, mert a mondatforma folyamatosan bővül. Ha egy szóhoz konstruáljuk és több a levélelemek között szereplő terminális jelek száma, mint a szó hossza, akkor sikertelen a levezetés. Visszaléphetünk és próbálhatunk más helyettesítéseket. Visszalépéses elemzés.

Kanonikus levezetés esetén a szó eleje folyamatosan kialakul. Ha a szó eleje a mondatformában a terminális szimbólumokon eltér a generálandó szótól, akkor a levezetés sikertelen.

Példa: környezetfüggetlen-e az adott nyelv?

- Ha L1 és L2 környezetfüggetlen, akkor L1&&L2 nem biztos, hogy az, mert a metszetük lehet véges, és minden véges nyelv reguláris nyelv.
- Ha L1 és L2 környezetfüggetlen, akkor L1||L2 is környezetfüggetlen.
- Ha L környezetfüggetlen nyelv, akkor L* is környezetfüggetlen.

Arra, hogy egy nyelv 2-es vagy 3-as a szabályok alapján lehet következtetni, de a szabályok 0→4 irányban szigorodnak, tehát egy reguláris nyelv beilleszthető a többibe.

Véges és verem-automaták

Helyes-e egy szó vagy nem. 0-ás nyelvnél erre nincs algoritmus. Minden Chomsky osztályhoz tartozik egy saját automata: olyan konstrukció, amely egy input szóról el tudja dönteni, hogy helyes-e vagy nem.

- input szalag: cellákra van osztva, minden cellában egy-egy karakter van. Általában véges, hossza az input hosszával azonos.

- olvasó fej: mindig egy cella fölött áll, és ebből olvas értéket. A fej lépked a szalagon balra-jobbra.
- output szalag: cellákra van soztva, minden cellában egy-egy karakter lehet. Véges vagy végtelen. Ahová még nem írt az automata, ott BLANK van.
- író-olvasó fej: output szalaghoz tartozik, ezzel lehet jelet írni az aktuális cellába vagy olvashat belőle. A fej lépked a szalagon balra-jobbra.
- belső állapotok: az automata lépked közöttük egy megadott séma alapján, egyszerre csak egy állapotban lehet.

Véges automata

nincs output szalag, nincs író-olvasó fej

input szalag véges, inputhosszú

Egy G(K, V, δ, q0, F) formális ötöst véges automatának nevezünk, ahol

- K: az automata belső állapotainak halmaza (véges!)
- δ: állapotátmeneti függvény, δ ⊆ KxV -> K, K: aktuális állapot, V: input jel
- q0 ∈ K: speciális belső állapot, a kezdőállapot
- F ⊆ K: befejező állapotok halmaza

Induláskor:

- az automata q0-ban van
- az input szalagon a szó jelei vannak felírva folytonosan balról jobbra
- az olvasó-fej az input szalag legbaloldalibb cellája fölött ál

Működés:

- az olvasó-fej beolvassa az aktuális jelet az input szalagról
- a jel és az aktuális belső állapot ismeretében a δ alapján új belső állapotba vált
- az olvasó fejet lépteti eggyel jobbra

Megálláskor:

- az automata megáll, ha az olvasó-fej lelép a szalagról
- ha megállt, meg kell vizsgálni az aktuális állapotot (elfogadó, nem elfogadó)
- n lépés után fix megáll az automata

Verem automata

van output szalagja, van író-olvasó fej, de nem mozoghat, csak a legjobboldalibb cellába írhat/olvashat. Írás után jobbra mozdul. Olvasásnál előbb bemozdul balra, kiolvas.

Egy G(K, V, W, δ, q0, z0, F) formális hetest verem automatának nevezünk, ahol:

- K: az automata belső állapotainak halmaza (véges!)
- V: az input ABC
- W: az output ABC
- δ: állapotátmeneti függvény δ ⊆ Kx(V&&{ ε })xW → KxW, K aktuális állapot, V input jel, nem olvasás esetén ε, W input jel a veremből
- q0 ∈ K: speciális belső állapot, a kezdőállapot
- z0 ∈ W: speciális veremABC-béli jel, verem üres szimbólum

- F ⊆ K: befejező állapotok halmaza

Induláskor:

- az automata q0 állapotban van
- a veremben csak egy jel van, a z0
- az input szalagon a szó jelei vannak felírva folytonosan balról jobbra
- az olvasó-fej az input szalag legbaloldalibb cellája fölött áll

Működés:

- az olvasó-fej vagy olvas az input szalagról, vagy nem
- a veremből mindig olvassa a verem tetején lévő jelet
- ezen input jelek és az aktuális belső állapot ismeretében a δ alapján új belső állapotba vált, és a verembe egy szót ír
- ha olvasott az input szalagról, az olvasó-fejet lépteti eggyel jobbra

Megálláskor:

- az automata megáll, ha az olvasó-fej lelép a szalagról
- ha megállt, meg kell vizsgálni az aktuális állapotot (elfogadó, nem elfogadó)
- az automata nem minden lépésben olvas jelet az input szalagról, ezért nem mindig lép, tehát nem biztos, hogy megáll n lépés után. Az se biztos, hogy egyáltalán megáll valaha.

Turing gépek

Olyan automaták, amelyeknek nincs külön output szalagjuk, de az input szalagra írni is tudnak. Az input szalag mindkét irányban végtelen.

Egy G(K, V, W, δ, q0, B, F) formális hetest Turing automatának nevezünk, ahol:

- K: az automata belső állapotainak halmaza
- V: input ABC
- W: output ABC
- δ: állapotátmeneti függvény δ ⊆ KxW → KxWx{←,→}, parciális kell legyen (nem teljesen definiáltak a szabályok, nincs minden kidolgozva)
- q0 ∈ K: speciális belső állapot, a kezdőállapot
- B: blank, B ∈ W
- F ⊆ K: befejező állapotok halmaza

Induláskor:

- az automata q0 állapotban van
- az input szalagon a szó jelei vannak felírva folytonosan balról jobbra, többi helyen BLANK
- író-olvasó fej a legbaloldalibb cella felett áll

Működés:

- az olvasó fej beolvas egy jelet
- ezen input jelek és az aktuális belső állapot ismeretében a δ alapján új belső állapotba vált

- egy jelet ír vissza a szalag aktuális cellájába
- lépteti a fejet balra vagy jobbra

Megálláskor:

- az automata megáll, ha δ függvény nincs értelmezve az aktuális input jel és belső állapot esetén

Egyirányú végtelen szalagos Turing gép

egy irányban végtelen, másik irányban le lehet róla lépni. Ekvivalens a mindkét irányba végtelen szalagossal.

Több szalagú Turing gép

Több, esetleg mindkét irányban végtelen szalagú output szalag van. Ekkor a δ input adatai nem csak egy szalagról származnak, hanem mindről, és mindegyikre ír minden lépésben, és minden fejre külön megmondja, hogy merre lépjen. Ekvivalens az egyszalagossal.

Delta leképezés

A δ leképezése verem automatáknál

δ	K	V∪{ε}	W	K	W*
1.	q_0	ε	z_0	q_0	z_0
2.	q_0	1	z	q_1	zxy
3.	q_0	1	z	q_0	ε
4.	q_1	1	z	q_1	Z

1.sor: Az automata nem olvas az input szalagról, és olvasás előtt-után q_0 állapotba kerül, a veremből z_0-t olvas, és ír. Ezt a sort az automata önmagában végtelen sokszor ismételheti (végtelen ciklus).
2. sor: Az automata egy jelet vesz ki a veremből (z), és három jelet tesz vissza (zxy). Ezen sor végrehajtása után a verem aktuális mérete 2-vel növekszik.
3. sor: Az automata egy jelet vesz ki a veremből (z), és nem tesz vissza semmit (ε). Ezen sor végrehajtása után a verem aktuális 1-es csökken.
4. sor: Az automata egy jelet vesz ki a veremből (z), és egy jelet tesz vissza (z). Ezen sor végrehajtása után a verem aktuális mérete nem változik.

A δ leképezése Turing gép esetén

A δ függvény egy (q,a) páron van értelmezve (q∈K, a∈W). δ (q,a) = (q', a', <-) alakú sorokból áll, ahol a (q,a) párhoz hozzárendel egy új állapotot (q'), egy jelet visszaír a szalagra (a'), és megadja, hogy a fejnek balra, vagy jobbra kell lépnie.

Automaták és grammatikák kapcsolata

Egy nyelvet egy grammatika készít vagy egy automata ismer fel. Az automata gyakorlatilag azt takarja, hogy egy grammatikából hogyan lesz algoritmus.

b) Rendszerfejlesztés technológiája: Szoftver életciklus. Tervezés, implementáció, tesztelési iterációk. Rendszerfejlesztés eszközei, feladatkövetés, verziókövetés. Tesztelési technikák, tesztelés szintjei.

Szoftver életciklusa

Szoftver

- egyedi: ismert a megrendelő, az ő számára készül a szoftver (például állami megbízások)
- dobozos: leendő, ismeretlen vásárlók, például számítógépes, konzolos játékok, operációs rendszerek

A szoftver életének állomásait írja le megszületéstől átadásig. Azért ciklus, mert a szoftver állandóan változik, új igények vannak, bővíteni kell. Lépéseit a módszertanok határozzák meg. A módszertan célja, hogy magas minőségű szoftvert fejlesszünk minél kisebb költséggel.

Életciklus állomásai:

- → felhasználói oldalon új igény merül fel
- igények elemzése, meghatározása, követelményspecifikáció elkészítése
- rendszerjavaslat kidolgozása (funkcionális specifikáció, ütemterv, szerződéskötés)
- rendszerspecifikáció (megvalósíthatósági tanulmány, nagyvonalú rendszerterv)
- logikai és fizikai tervezés
- implementáció
- tesztelés (teztterv, tesztesetek, teszt napló, validált szoftver)
- rendszerátadás és bevezetés (felhasználói dokumentáció)
- üzemeltetés és karbantartás
- felhasználó új igénye →

Felhasználói oldalon új igény merül fel

Mindig lesznek újak.

Igények elemzése, meghatározása

félreértések fordulhatnak elő, mert nem ugyanazokat a dolgokat értjük rajta (szakmai kifejezések). Ezért riportot készítünk, aminek célja a megrendelő üzleti folyamatainak és igényeinek felderítése.

- szabad: megrendelő saját szavakkal mondja el az igényt, mi figyelünk és felhívjuk a figyelmét az ellentmondásokra, hiányosságokra, ismeretlen fogalmakra)
- irányított: általánosabb véleménykutatás a cél, előre megírt kérdőívet töltetünk ki

Követelményspecifikáció

Alapja a megrendelővel készített riportok. Részei: jelenlegi helyzet, vágyálomrendszer, rendszerre vonatkozó pályázat, jogszabályi háttér, jelenlegi üzleti folyamatok, igényelt üzleti folyamatok, követelménylista, irányított és szabad szöveges riportok szövege, fogalomszótár.

Rendszerjavaslat

Funkcionális specifikáció → ütemterv, árajánlat → szerződés.

- funkcionális specifikáció: a felhasználó szemszögéből írja le a rendszert. Követelményspecifikáció igényeire válasz.

- ütemterv: árajánlat legfontosabb része, ez határozza meg mi kerül be a rendszer x. verziójába és mi nem
- árajánlat: hány ember hány napig fejleszt, mi a napidíj. 50%-os kihasználtság mellett legyen nullszaldó.

Rendszerspecifikáció

- megvalósíthatósági tanulmány: projekt indításával, finanszírozásával kapcsolatos információk a döntéshozóknak
- nagyvonalú rendszerterv: mit, miért, hogyan, mikor, miből akarunk létrehozni. Reális kell legyen, megvalósítható lépéseket írjon elő, mint tervrajz az építkezésen.

Rendszerterv alapja a funkcionális specifikáció, a fejlesztő szemszögéből.

Részei:

- rendszer célja
- projektterv: projekt szerepkörök, felelősségek, ütemterv, mérföldkövek, üzleti folyamatok modellje, üzleti szereplők, folyamatok, funkcionális, nemfunkcionális követelmények, jogszabályi háttér
- funkcionális terv: rendszerszereplők, rendszerhasználati esetek, menü-hierarchiák, képernyőtervek
- fizikai környezet: szoftverkomponensek, külső rendszerek, hálózati topológia, hardverek, fizikai alrendszerek, fejlesztő eszközök, keretrendszerek
- absztrakt domain modell: megvalósítandó rendszer fogalmai, megvalósítás magas szintű váza 1-2 példán keresztül, domain specifikáció, fogalmak, absztrakt komponensek
- architekturális terv: architekturális tervezési minta (MVC), alkalmazás rétegei, komponensei, változások kezelése, rendszer bővíthetősége, biztonsági funkciók
- adatbázis terv: logikai adatmodell, tárolt eljárások
- implementációs terv: perzisztencia-osztályok, üzleti logika osztályai, kliensoldal osztályai
- tesztterv: tesztelési elvek, folyamat, kontroll, tesztesetek, sikeres teszt kritériumai
- telepítési terv: rendszer telepítésére vonatkozó elvek, megszorítások, fizikai környezet
- karbantartási terv: frissítés módja, folyamata.

Rendszerterv fajtái

- konceptuális (mit, miért)
- nagyvonalú (mit, miért, hogyan, miből)
- részletes (mit, miért, hogyan, miből, mikor)

Logikai és fizikai tervezés

logikai rendszerterv: megvalósítás hogyan része, nincsenek fizikai korlátok, tartalma: üzleti folyamatok modellje, funkcionális, nem funkcionális követelmények, használati esetek, állapotgépek, funkcionális felépítés, menük, felhasználói felületek, logikai adatmodell, adatfolyam diagramok

fizikai rendszerterv: fizikai megszorítások figyelembevétele. Osztály, adatbázis, teszt, telepítési tervek, rendszerspecifikációk (fejlesztési, futtatási környezet), szoftver architektúra

lightweight módszertan: nem kell rendszerterv, mert sok időbe telik

Implementáció

Forráskód, programozói dokumentáció elkészítése.

Tesztelés

Hibák megtalálása üzembe helyezés előtt. Hiba biztos van.

Rendszerátadás és bevezetés

Gondoskodni kell a megfelelő rendszerkörnyezetről: meglévő infrastruktúra, vagy új. A fizikai rendszerterv része.

Átadás dokumentumai: felhasználói dokumentáció, üzemebe helyezési kézikönyv, átadás-átvételi jegyzőkönyv

Üzemeltetés és karbantartás

A rendszergazda feladata.

Elvárások

- rendszeres biztonsági mentés
- meghibásodás esetén utolsó konzisztens állapot visszaállítása
- frissítés
- általános átvizsgálás során észlelt hibák javítása
- bejelentések során érkezett hibák javítása
- biztonsági beállítások felülvizsgálata és korrigálása

Rendelkezésre állás: megbízhatóság, karbantarthatóság, szolgáltatási képesség, biztonság

Tervezés-implementáció-tesztelés ciklus

Tervezés

Ami efelett van, az kb. mind az.

Implementáció

Kifejtve korábban ugyanezen címszó alatt.

Tesztelés

Tevékenységek: teszterv elkészítése, tesztesetek tervezése, tesztek végrehajtása, kilépési feltételek vizsgálata, eredmények értékelése, jelentéskészítés.

Teszterv: tartalmazza a teszteléshez szükséges információkat. Tartalmazza a tesztelés követelményeit, a teszt terjedelmét.

Tesztmodell: megmondja mi és hogyan lesz tesztelve

Teszteset: bemeneti értékek, végrehajtási feltételek és várható értékek halmaza

Teszt szkript: számítógép által végrehajtható utasítások halmaza, amelyek automatizálják egy teszt eljárás végrehajtását.

Unit-teszt: maximum értéket visszaadó metódushoz:

[TestMethod()]

```
public void maxTest()

{

Program target = new Program()

int a = 10; int b = 20; int expected = 20; int actual

actual = target.max(a, b);

Assert.AreEqual(expected, actual);

}
```

Meg kell határozni mikor sikeres egy teszt (hány % sikeres teszteset pl)

Ehhez tudni kell: mi a teszt tárgya (rendszer, vagy csak egy funkció), tesztbázis (teszt tárgyára vonatkozó követelményeket tartalmazó dokumentáció), tesztadat (amivel meghívjuk a teszt tárgyát), kilépési feltétel (mikor zárható le a teszt, pl sikeres lefutás, vagy kritikus részek tesztlefedettsége 100%)

Rendszerfejlesztés eszközei

Feladatkövetés

A módszertanok határozzák meg, pl az agilis módszertan. Scrum, Kanban, XP

Fejlesztési ciklusok vannak, 1-4 hét. Akkor sikeres egy ciklus, ha az elején meghatározott feladatokat bemutatható állapotba hozzuk a végére. A fejlesztő csapatban van egy PO, aki a megrendelő delegáltja, ő prezentálja a megrendelő igényeit. Feladatok időigényét becsüljük, priorizáljuk. Trello: lássuk, hogy ki mit csinál, melyik mennyi időbe/energiába telik, hány ember kell rá, mi a prioritása a feladatoknak.

Verziókövetés

Állományok tartalmi változásait követjük figyelembe véve, hogy ki és mikor módosította azokat. Korábbi állapotokat is elő tud hívni. Meg lehet nézni egy állomány állapotai közötti különbséget. Ha ketten egyszerre módosítják ugyanazt, akkor jelzi a konfliktust és lehet dönteni, hogy melyik maradjon meg. Brancheket hozhatunk létre: külön változatok, pl általános/középiskola, külön fejleszthetőek.

Tesztelési technikák

Feketedobozos tesztelés

Specifikáció alapú. Leggyakoribb formája, hogy egy adott bemenetre tudjuk, milyen kimenetet kellene adnia a programnak.

Fehérdobozos tesztelés

Strukturális teszt, forráskód alapján készülnek a tesztesetek. A lefedettséget kell értelmezni, ami azt mutatja meg, hogy a struktúra hány %-át tudjuk tesztelni a meglévő tesztesetekkel.

Struktúrák: kódsorok, elágazások, metódusok, osztályok, funkciók, modulok.

Tesztelés szintjei

- fejlesztői tesztek:
    - komponensteszt: rendszer egy komponensét teszteli önmagában

        - unit-teszt: metódusokat teszteli. Adott paraméterekre ismerjük a metódus visszatérési értékét, azt teszteljük, hogy ez az elvárt-e.
        - modulteszt: modul nem-funkcionális tulajdonságát teszteli: sebesség, memóriaszivárgás, szűk keresztmetszet.
    - integrációs teszt: kettő vagy több komponens együttműködését teszteli. Az OS és a rendszer közti interfészt, más rendszerek felé nyújtott interfészeket is teszteli. Célszerű minél hamarabb elkezdeni, mert minél nagyobb az integráció, annál nehezebb megtalálni a hibát.
    - rendszerteszt: az egész rendszert, minden komponenst együtt tesztel. Teszteli, hogy a kész rendszer megfelel-e a követelmény specifikációnak, a funkcionális specifikációnak és a rendszertervnek. Feketedobozos tesztelés. A tesztkörnyezetnek hasonlítania kell az éles környezethez.
- átvételi teszt: felhasználók a kész rendszert tesztelik. Fajtái: alfa, béta, felhasználói átvételi, üzemeltetői átvételi.

10. tétel

a) Hálózati architektúrák és protokollok: Csomagkapcsolt hálózatok működése, az OSI és a TCP/IP modell összehasonlítása, forgalomirányítási és IP címzési alapok – Kapcsolódó protokollok: TCP, UDP, ICMP, DNS.

Csomagkapcsolt hálózatok működése

Hálózat: egymással kapcsolatban lévő csomópontok összessége

Csomópont: önálló kommunikációra képes, saját hálózati címmel rendelkező eszköz.

Csomagkapcsolt hálózatokban nincs előre kiépített út a kommunikálni kívánó számítógépek között. Az üzenetet max. méretű adatcsomagokra bontják és ezeket önálló egységenként küldik el, ezért a csomag fejrésze tartalmazza a címzett(ek) címét. Az IMP-k különböző útvonalakon juttathatják el a címzett(ek)nek, ezért a beérkezési sorrend megváltozhat. A címzettnek gondoskodni kell a helyes sorrendbe rakásról. Ha nagy az adatforgalom, akkor az IMP korlátozott kapacitása miatt elveszhet csomag.

OSI

Elméleti modell, gyakorlatban TCP/IP-t használnak helyette. Fontos volt megalkotni, mert egy hierarchikus rendben felépített protokollrendszer könnyebben kezelhető. Könnyen implementálhatóak a változások, a rétegek együttműködhetnek. Különböző gyártók a protokoll betartásával kompatibilis eszközöket gyárthatnak. Az egyes rétegek egymás között saját szabályok szerint kommunikálnak.

Sok kis adatot mozgat, gyorsabb, rugalmasabb, megbízhatóbb.

Beágyazási folyamat: az adatot a továbbítás előtt új protokollfejléccel látják el. Minden réteg új adatrésszel egészíti ki a szétdarabolt, kis adatrészeket. Bitek – fizikai, keret – hálózatelérési, csomag – hálózati, szegmens – transport, adat – alkalmazási.

Fizikai réteg: közvetlen összeköttetést teremti meg, topológiát ír le.

Adatkapcsolati réteg: MAC cím (fizikai) és LLC (közeghozzáférés vezérlés). Megbízható adatátvitelt biztosít egy fizikai összeköttetésen keresztül és a keretek sorrendhelyes kézbesítését.

Hálózati réteg: logikai címzés (IP). Összeköttetést és útvonalválasztást biztosít két csomópont között.

Szállítási réteg: megbízható hálózati összeköttetést létesít két csomópont között. Virtuális áramkörök kezelése, átviteli hibák felismerése, javítása, áramlás szabályozás. TCP/UDP

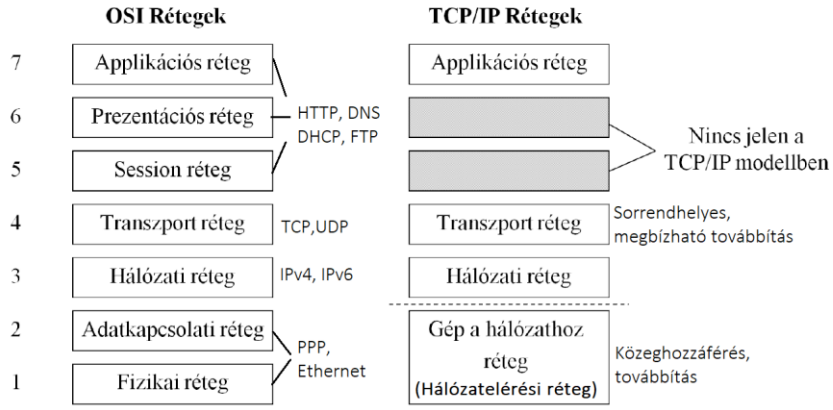
Session/viszony réteg: applikációk közötti dialógusok. Ellenőrzi, hogy véget ért-e már az adatsor vagy még érkeznek részei.

Megjelenítési réteg: különböző csomópontokban használt különböző adatstruktúrákból eredő információ-értelmezési problémák feloldása.

Alkalmazási réteg: applikációk (fájlátvitel, e-mail) működéséhez nélkülözhetetlen szolgáltatásokat biztosítja.

TCP/IP

- Applikációs rétegbéli HTTP protokoll kezdi a HTML formátumú adatok továbbításával a szállítási réteg felé. A felhasználó programja és a szállítási réteg között teremt kapcsolatot.
- A szállítási rétegben az alkalmazás adatai TCP szegmensekre darabolódnak fel. Ezek kapnak fejlécet, ami segít a címzett gép alkalmazását kiválasztani, aminek majd az adatot kell feldolgoznia. (TCP, UDP Továbbá információkat tárolunk itt, ami az eredeti visszaállításához szükségesek. Majd küldi tovább a hálózati rétegbe.
- Hálózati réteg IP protokoll. A teljes TCP szegmenst beágyazzuk IP csomagba. Az IP fejléc tartalmazza a forrás és a cél IP címet és a feldolgozási folyamathoz információkat.
- A kapott IP csomagot beágyazza egy keretbe, amiben van keretfejléc és utótag. A fejlécben a forrás és a cél gép fizikai címe van. Az utótag hibaellenőrzési információkat tartalmaz. Ha nem fér el egy keretben, akkor felbontja és az utolsó keret végére egy tail-t kapcsol, hogy a fogadó oldalon vissza lehessen állítani az eredeti adatot.
- A hálózati kártya bitekre kódolja az üzenetet.



Forgalomirányítás

Csomagok továbbítási irányának meghatározásával kapcsolatos döntések meghozatala a routing tábla alapján. Feladata a csomagok útjának kijelölése és gyors eljuttatása A-ból B-be. Összetettségét a hálózat topológiája határozza meg. Például csillagnál csak a középponti csomópontnak kell rendelkeznie minden szükséges információval, mert minden adat itt megy keresztül.

Egy csomag címzettje lehet: saját maga, egy helyi állomás vagy egy távoli állomás.

Alapértelmezett átjáró: helyi hálózatról távolira küldi az adatot. Legtöbbször router, ami routing táblázatot tartalmaz. A RAM-ban tárolja a távoli hálózatok adatait, ezért reset-kor ezeket újra össze kell gyűjtenie.

Routing táblázat: forgalomirányításhoz szükséges információkat tartalmazza.

- célhálózat: elérhető hálózatok listája
- hálózati maszk: segít meghatározni az IP cím hálózati és állomás részét
- átjáró: a távoli elérésre használt cm (router címe, ami felé küldeni kell)
- interface: a router saját, fizikai interface-ének címe, amit használni kell a csomag tovább küldéséhez.
- mérték: az útvonal költsége, a legoptimálisabb út megállapításához.

Tábla célhálózatainak csoportjai:

- 0.0.0.0: helyi alapértelmezett útvonal. Ide továbbítja azon csomagokat, amik célhálózata nem szerepel az irányítótáblában.
- 127.0.0.0-127.255.255.255: loopback, visszacsatolási cím. Arra használják, hogy visszairányítsák saját maguknak a forgalmat.
- 192.168.10.0-192.168.10.255: helyi címek. 0 általában a hálózat címe, 255 a broadcast cím, amivel az összes helyi címet egyszerre lehet elérni.
- 224.0.0.0-239.255.255.255: multicast címek
- 255.255.255.255: megkereshető a HCP szerver, mielőtt az állomás helyi IP címet kapna a DHCP-től

Forgalomirányítók működése:

- a router az input interfészen érkező csomagot fogadja
- a router a csomag célcímét (IP illesztéssel) illeszti a routing táblázat soraira. Ha a célcím több sorra illeszkedik, akkor a leghosszabb prefixű sort tekintjük illeszkedőnek.
- ha nem létezik illeszkedő sor, akkor a cél elérhetetlen, a csomag nem továbbítható
- ha létezik illeszkedő sor, akkor a csomagot az ebben szereplő kimeneti interfészen továbbítjuk (adatkapcsolati rétegbéli beágyazással) a következő hop-ként megadott szomszédhoz, illetve a célállomáshoz, ha már nincs több hop.

IP címzés

- OSI besorolás alapján a 3. rétegben helyezkedik el.
- IPv4-nél 4 db 8 bites egység, IPv6-nl 6 db
- csomagkapcsolt hálózatot valósít meg (routing)
- nincs hibadetektálás, megbízhatatlan protokoll
- IP cím: eleje a hálózat azonosítója, vége a csomópont azonosítója (a hálózaton belül)
    - a hálózati rész egy hálózaton belül mindig azonos, az állomás rész mindig egyedi
    - hálózati és állomásrész választóvonalát az alhálózati maszk adja meg. Az egyesek (255) jelölik a hálózat részt, a 0-ák pedig az állomás részt.
    - a hálózati előtag (prefix) az alhálózati maszk 1-eseinek db száma. 255.255.255.0 alhálózati maszk esetén 24 (3*8db) 1-es van. Ha növeljük az 1-esek számát, akkor csökkentjük a hálózatban szétoszthtató állomáscímek számát.
- az IP cím alhálózati maszkkal való AND művelete segít eldönteni, hogy az adatot helyi hálózaton kell elküldeni, vagy az alapértelmezett átjárónak kell címezni továbbításért. Összehasonlítja az eszköz és a címzett címének hálózati részét, illetve eltüntetheti az IP cím állomásrészét.
    - Példa:
        - IP: 192.168.10.59
        - alhálózati maszk: 255.255.255.0
        - AND után: 192.168.10.0 → hálózati rész
- Címzés lehet egyedi (konkrét cím) vagy szórásos (mindenkinek a hálózaton, állomásrész 255)
    - irányított szórás: távolról egy másik hálózat minden állomására
    - korlátozott szórás: mind a 4 bájt egyes, az ilyet a router felfelé nem továbbítja, ezért ezt alhálózati szórási tartománynak hívjuk
- Multicast címek: pl.: video adás továbbítására 224.0.0.0-239.255.255.255. Alsó 255 cím tartomány a local hálózatra, a 224.0.1.0-tól a maximumig az interneten keresztüli multicastra.

- IP cím osztás. /24-es hálózatot ketté szeretnénk vágni 2 db /25-ösre. A példában az alsó csoport van kifejtve, a kettévágás után két ilyen egység jön létre.
    - 192.168.10.0-127 (IP cím tartomány marad): 192.168.10.0 /25
    - 255.255.255.128 (alhálózati maszk)
    - 192.168.10.0 (hálózat címe)
    - 192.168.10.127 (broadcast cím)
    - Minden osztás esetén 2 címet elvesztünk a hálózat és a broadcast miatt.
    - Ha /26-os osztást csinálunk 192.168.10.0-64 a tartomány. Alhálózati maszk 255.255.255.192 lesz.

IPv4 címtípusok:

- A: nagy hálózatokhoz. 16 millió vagy több állomás számára. IPv4 címkészlet fele ide tartozik. 0.0.0.0 /8-tól 127.0.0.0 /8-ig. Csak 120 vállalat kap belőle
- B: közepes és nagy vállalatok számára: 65000 állomás. A teljes készet 25%-a. 128.0.0.0 /16-tól 191.255.0.0-ig
- C: legfeljebb 254 állomás számára. /24-es hálózatok. 192.0.0.0 /24-től 223.255.255.255-ig 2 millió kis hálózat

IPv6: egyszerre működik IPv4-gyel. Itt már hexában adjuk meg a címeket. 32 db hexa alkotja a címet. Nincs alhálózati maszk. Az IP cím előtagja 0-128 közé kell essen.

DHCP: automatikus címkiosztás. Az új gép küld egy broadcast-ot DHCP kéréssel. A forgalomirányító feldolgozza, és kioszt neki egy elérhető címet.

Kapcsolódó protokollok

Szállítási rétegben: ez fogadja az alkalmazási rétegtől az adatot és készíti elő azokat a hálózati rétegben való címzésre. Az adatot szegmensekre bontja, felel azért, hogy minden szegmens megérkezzen, ha nem, akkor az újra küldésért.

TCP

IP protokoll hiányosságait javítja ki: nyugtázás, speed control, hibadetektálás. Garantálja a szegmensek célba érkezését, nyomon követi őket, a megérkezett adatot nyugtázza, a nem nyugtázottokat újra küldi. Ott használják, ahol fontos, hogy minden szegmens célba érjen: HTTP, FTP, webböngésző, e-mail, SMTP

Működése:

- a forrás felveszi a kapcsolatot a céllal
- fogad elküldi az ablakméretet
- forrás felad 4x1Kb-nyi csomagot.
- fogadó jelzi, hogy a puffere megtelt
- forrás vár
- vevő jelzi, hogy jöhetnek az adatok
- kapcsolat zárása

UDP

A TCP többletterhelést és késleltetést okozhat. UDP célja a sebességnövelés. Stream, online gaming. Nincs nyugtázás. Datagram alapú: informatikai protokoll, amely nem nyugtázható, így megbízható adatátvitelt nem tesz lehetővé.

ICMP

Internet Control Message Protocol: értesülhetünk a hibákról és azok típusáról, valamint hálózati diagnosztizálásban segít. Datagram-orientált. IP-t használja borítékként (ICMP csomagok az IP hálózaton mennek). Legismertebb a ping és a traceroute. Megfordulási és utazási idő mérése, köztes hálózati csomópontok felderítése két gép között.

DNS

Tartománynév rendszer: egy hierarchikus, nagymértékben elosztott elnevezési rendszer számítógépek, szolgáltatások, illetve az internetre vagy egy magánhálózatra kötött erőforrások számára. A részt vevők számára kiosztott tartománynevekhez különböző információkat társít. Legfontosabb funkciója az emberek számára értelmezhető tartományneveket a hálózati eszközök számára érthető numerikus azonosítókká oldja fel.

b) Szolgáltatásorientált programozás: Az RPC architektúrája, működési elve, jellemzői. A Google RPC működési elve, jellemzői. A protocol buffers szerepe a gRPC-ben. A WEB, egyrétegű, kliens-szerver modell, többrétegű alkalmazások, vastag és vékony kliensek, elosztott rendszerek jellemzői. A REST tulajdonságai. A web service jellemzői. A WCF architektúrája, az ASMX és a WCF közötti eltérések.

RPC

A kliensoldali eljárás futtatási igényeket küld a szervernek, amely azokat megérkezésük után adminisztrálja, elvégzi a kért funkciót, viszontválaszt küld és az eljáráshívás visszatér a klienshez.

Típusai:

- programozási nyelvbe (keretrendszerbe) integrált (.NET RPC)
- speciális célú interfészdefiníciók használata a kliens-szerver közti kapcsolat leírására (GRPC, XLM-RPC)

Jellemzői:

- a kérés/fogadás kommunikáció nyelvi szintű absztrakciója üzenetküldésen és távoli eljáráshíváson alapul
- típusellenőrzött mechanizmus, amely lehetővé teszi, hogy egy adott gépen történő nyelvi szintű hívás automatikusan a megfelelő nyelvi hívássá alakuljon
- a szerver által biztosított eljárások interfészdefiníciója meghatározza az eljárások karakterisztikáját (eljárások nevei, paramétereinek típusa) a szerver kliensei felé
- távoli eljáráshívások egy úgynevezett RPC-csomagba ágyazottak, melyek alacsony szintű rendszerhívások

Architektúra: klienscsonk (client stub) és szervercsonk (server stub). Ezek a csonkok absztrahálják le a kliens és szerver számára a másik féllel történő kommunikációt.

Működési elve:

- a kliens meghívja a klienscsonkot metódushíváson keresztül (úgy hívja meg a metódust, mintha az neki lokális lenne)
- a klienscsonk becsomagolja az eljárás azonosítóját, paramétereit. Ennek neve marshalling. Ezután meghívja az OS-t
- a kliens OS-e átküldi az üzenetet a szerver OS-nek
- a szerver OS-e átadja az üzenetet a szervercsonknak
- a szervercsonk kicsomagolja a z üzenetet (a metódushívást, paramétereket stb.), majd meghívja az eljárást, és ha van, megkapja a visszatérési értéket
- ugyanez visszafelé, csak szervertől kliens felé

gRPC

Google által fejlesztett általános célú RPC infrastruktúra (Stubby)

Jellemzői:

- cross-platform
- nyelvfüggetlen

- nyílt forráskódú
- HTTP/2-őt használ

Működési elve:

Egy kliensalkalmazás közvetlenül meg tudja hívni egy metódusát a szerveren futó alkalmazásnak, mintha egy lokális objektum lenne. Először egy szolgáltatást kell definiálni, meghatározva a metódusokat. A szerveroldalon implementálni kell ezt a definiált szolgáltatást (interfész), majd futtatni azt, amely kezeli a klienshívásokat. A kliensoldalon itt is található csonk, amely egy olyan osztály, amely ugyanolyan funkciókkal rendelkezik, mint a szerveren lévő igazi.

Protocol buffers szerepe a gRPC-ben

Nyelvfüggetlen, platformfüggetlen mechanizmus, mely strukturált adatok szerializálásáért felelős. Minden protokoll buffer üzenet egy kisméretű, logikai rekord, amely név-érték párok sorozatát tartalmazza. A szerializálandó adatok struktúráját egy .proto fájlba kell tenni. Ebben lehet meghatározni a szolgáltatásunk interfészét is.

Példa:

```
message Person {
      string name = 1;
      int32 id = 2;
      string email = 3;
      enum PhoneType {
            MOBILE = 0;
            HOME = 1;
            WORK = 2;
      }
      message PhoneNumber {
            string number = 1;
            PhoneType type = 2 [default = HOME];
      }
      repeated PhoneNumber phone = 4;
}
service SearchService {
      rpc Search (SearchRequest) returns (SearchResponse);
```

A WEB, egyrétegű, kliens-szerver modell, többrétegű alkalmazások, vastag és vékony kliensek, elosztott rendszerek jellemzői

Egyrétegű alkalmazások: helyi programok, nem képesen kommunikációra.

Kliens-szerver modell

Kliens: olyan számítógép, amely hozzáfér egy (távoli) szolgáltatáshoz, amelyet egy számítógép hálózathoz tartozó másik gép nyújt.

Jellemzői:

- kéréseket küld a szervernek
- a választ a szervertől fogadja
- egyszerre (általában) csak kis számú szerverhez kapcsolódik
- közvetlenül kommunikál a felhasználóval

Szerver: olyan (általában nagy teljesítményű) számítógép, illetve szoftver, amely más gépek számára a rajta tárolt vagy előállított adatok felhasználását a kiszolgáló hardver erőforrásainak kihasználását, illetve más szolgáltatások elérését teszi lehetővé.

Jellemzői:

- passzív, a kliensektől várja a kéréseket
- a kéréseket feldolgozza, majd visszaküldi a választ
- nem áll közvetlen kapcsolatban a felhasználóval

Többrétegű alkalmazások

Kliens-szerver architektúra, melyben a megjelenítés, az adatkezelés és az üzleti logika különálló folyamatokra van bontva. Így könnyebben karbantartható és fejleszthető a rendszer. Leggyakoribb a háromrétegű architektúra:

- megjelenítés: a társ rendszer felé nyújt interfészt (ez gyakran felhasználói interfész), és az ehhez kapcsolódó eseményeket kezeli le. Ilyen például egy weboldal.
- üzleti logika: feladata az üzleti folyamatok futtatása, hosszú életű tranzakciók kezelése
- perzisztencia: feladata az adatok tárolása

Nem azonos az MVC-vel. Ott a megjelenítés hozzáfér a modellhez (perzisztencia), itt csak az üzleti logikán keresztül.

Vastag kliens

A kliens a megjelenítésen kívül más feladatokkal is foglalkozik. A szerver központi egységének kihasználtsága kiegyenlítettebb. Önmaga képes végrehajtani adatfeldolgozást, tárolóként viselkedhet.

Vékony kliens

A szükséges erőforrásokat is a távoli gépen veszi igénybe. Feladata legtöbbször a szerver által küldött adatok megjelenítése.

Elosztott rendszerek jellemzői

Önálló számítógépek olyan összessége, amely kezelői számára egyetlen koherens rendszernek tűnik. 4 célja van: távoli erőforrások elérhetővé tétele, átlátszóság, nyitottság, skálázhatóság. Egy program különböző részei több számítógépen futnak párhuzamosan a hatékonyabb működés érdekében. A gépek hálózaton keresztül kommunikálnak egymással.

REST

https://github.com/enpassant/miniatures/tree/master/src/main/scala/rest

A REST egy architekturális stílus, egy tervezési szempont web-szolgáltatások készítéséhez, ami 100%-ban a HTTP-re épül. Amelyek követik, azokat RESTful szolgáltatásoknak nevezzük.

6 kritérium:

1. kliens-szerver architektúra: a kliensek nem foglalkoznak adattárolással, a szerverek nem foglalkoznak a felhasználói felülettel
2. állapotmentesség: a szerveren nem tároljuk a kliensek állapotait. Minden egyes kérés bármely klienstől tartalmazza az összes információ a kérés kiszolgálásához
3. gyorsítótárazhatóság: a kliensek képesek legyenek a válaszokat gyorsítótárazni
4. réteges felépítés: a kliens nem tudja megmondani, hogy direkt kommunikál-e a szerverrel, vagy proxy-n keresztül
5. igényelt kód: opcionális, a szerverek ki tudják terjeszteni saját funkcionalitásukat programrészek átadásával a kliensek számára (Java appletek, JS kódok)
6. egységes interfész

Erőforrás lehet dokumentum, adatbázis egy rekordja vagy akár egy algoritmus eredménye. Bármi lehet, amit el lehet nevezni és URI-val meg lehet címezni: students/1. A HTTP által definiált műveleteket használjuk az URI-k által azonosított erőforrásokon a kérések elvégzésére. Az erőforrásokat egységesen egy technológiával közvetítjük: JSON, XML.

Műveletek:

- GET: visszaad egy erőforrást, nincs mellékhatása, gyorsítótárazható
- POST: létrehoz egy erőforrást
- PUT: módosít egy létező erőforrást
- DELETE: eltávolít egy erőforrást

Ha a szerver állaptát meg kell változtatni egy HTTP művelettel, akkor célszerű idempotens metódusokat használni. Ez azt jelenti, hogy ha egyszer elküldjük az üzenetet a szervernek, akkor ugyanaz lesz az állapota, mintha többször küldjük el ugyanazt. Ezek alapján POST helyett a PUT-ot érdemes használni például. Rendelésfelvételnél POST esetén ha nem érkezik az üzenetre válasz, a kliens nem tudja, hogy a szerver feldolgozta-e az üzenetet. PUT-nál mindegy, legfeljebb elküldjük még egyszer. PUT esetén figyelni kell, hogy ha 2 kliens egyszerre veszi fel pl. a 4-es rendelést, akkor felülírhatja a másikat. Ezt UUID-val pl. ki lehet szűrni.

Web service

Alkalmazások közötti adatcserére szolgáló protokollok és szabványok gyűjteménye.

- a protokollok és szabványok leírása mindenki által hozzáférhető és implementálható. Ezért a webszolgáltatások segítségével eltérő programozási nyelven írt, eltérő OS alatt futó programok is képesek egymással kommunikálni.
- a kommunikáció során küldött-fogadott adatok jellemzően XML-ben kerülnek leírásra.
- az XML dokumentum szerkezetét a SOAP szabvány írja le
- az XML formájú adatokat szintén szabványos protokollok (FTP, HTTP, SMTP, XMPP, HTTPS) segítségével lehet egyik géptől a másikig eljuttatni
- valamely webszolgáltatás által elérhető szolgáltatásokat a WSDL (Web Service Description Language) szabvány által leírt formában kell publikálni.
- a webszolgáltatásról az UDDI szabvány szerint lehet megadni információkat.

WCF

A szolgáltatás-orientált architektúra alapelveire épül, ezért támogatja az elosztott számítást, ahol a szolgáltatásokat a felhasználók hívják meg. WCF-et használva szinkron üzenetként adatokat lehet küldeni egyik végpontból a másikba.

Jellemzői:

- szolgáltatás-orientált alkalmazások készítésére használható (SOA, service-oriented architecture)
- különböző rendszerekkel képes együttműködni
- támogatja a legfontosabb ipari szabványok metaadat formátumait (WSDL, XML, stb.)
- támogatja az adatszerződéseket (data contract)
- többszintű kódolást, üzenetküldési típust támogat
- támogatja a tranzakciókezelést

Minden végpontnak három összetevője van. Egy WCF szolgáltatás több végponttal is rendelkezhet. Megoldható, hogy az egyes végpontokban akár ugyanaz a szolgáltatás (contract) kerül publikálásra, de eltérő protokollokon (binding) vagy az egyes címeken (address) más-más szolgáltatások érhetőek el.

Az ASMX egy egyszerűsített WCF.

ASMX: csak IIS-ben hosztolható, csak HTTP-t támogat, limitált védelem, XmlSerializer-t használ.

WCF: IIS, WAS, WinService, WCF-provieded, konzolos hosztolási lehetőségek, HTTP, TCP, MSMQ, NamedPipes-t támogat, összetett védelmi rendszer, DataContractSerializer-t használ.

11. tétel

a) A rendszerfejlesztés technológiája: Az életciklus dokumentumai. Módszertanok. Hagyományos és agilis fejlesztés összehasonlítása. Prototípus alapú megközelítés. Scrum. Extrém programozás. Kockázatmenedzsment.

Az életciklus dokumentumai

Életciklus: szoftver életének állomásait írja le az igény megszületésétől az átadásig. Azért ciklus, mert többször merülhet fel igény, és ilyenkor kezdődik minden elölről. Az életciklus lépéseit a módszertanok határozzák meg. A módszertan célja, hogy magas színvonalú szoftvert fejlesszünk minél kisebb költséggel.

Módszertanok osztályozása

- Milyen sorrendben követik egymást az életciklus fázisai: lineáris, spirális, iteratív
- Milyen implementációs nyelvet részesít előnyben a módszertan: folyamatorientált, adatközpontú, strukturált (top-down, bottom-up), objektumorientált, szervizorientált, keverék.
- Milyen megközelítést használ a módszertan célja eléréshez: jól dokumentált, prototípus alapú, rapid, agilis, extrém, keverék.
- Mennyire szigorú a dokumentáltság: könnyűsúlyú, nehézsúlyú
- Mit helyez a középpontba a módszertan: adatközpontú, folyamatközpontú, követelményközpontú, használati eset központú, tesztközpontú, felhasználó központú, emberközpontú, csapatközpontú, keverék.

Mindre jellemző az elemzés és tervezés, illetve a logikai és fizikai tervezés szétválasztása.

Strukturált módszertanok: vízesés, SSADM, V modell

A feladatokat modulokra bontják. Az életciklus lépései merev sorrendben követik egymást, lineáris. Nagy, hosszú projektek megvalósítására való. Lineáris, jól dokumentált, nehézsúlyú, adat és folyamat központú.

Vízesés modell: nagy projektekhez való, kevés döntési pontot definiál. Lineáris, nincs a lépések között átfedés, nem térhetünk vissza lezárt fázisba. Rengeteget kell dokumentálni, ezt elfogadtatni a megrendelővel, ezzel biztosítják, hogy nincs félreértés. Lineáris, strukturált, jól dokumentált, nehézsúlyú, nem agilis. Fázisai: szükségletek felmérése, rendszertervezés, megvalósítás, tesztelés, üzembe helyezés és karbantartás.

V modell: vízesés kiegészítése teszteléssel. Végrehajtjuk a fejlesztés lépéseit, majd a tesztelés lépéseit. Ha hibát találunk, vissza kell menni az adott fejlesztési lépésre.

Hagyományos vs agilis szoftverfejlesztés

A résztvevők, amennyire lehetséges megpróbálnak alkalmazkodni a projekthez.

Az agilis szoftverfejlesztés szerint értékesebbek:

- az egyének és az interaktivitás szemben a folyamatokkal és az eszközökkel,
- a működő szoftver szemben a terjedelmes dokumentációval
- az együttműködés a megrendelővel szemben a szerződéses tárgyalásokkal
- az alkalmazkodás a változásokhoz szemben a terv követésével

Az agilis szoftverfejlesztés alapelvei:

- a legfontosabb a megrendelő kielégítése használható szoftver gyors és folyamatos átadásával
- még a követelmények kései változtatása sem okoz problémát
- a működő szoftver/prototípus átadása rendszeresen, a lehető legrövidebb időn belül
- napi együttműködés a megrendelő és a fejlesztők között
- a projektek motivált egyének köré épülnek, akik megkapják a szükséges eszközöket és támogatást a legjobb munkavégzéshez
- a leghatékonyabb kommunikáció a szemtől-szembeni megbeszélés
- az előrehaladás alapja a működő szoftver
- az agilis folyamatok általi fenntartható fejlesztés állandó ütemben

Prototípus

Cél, hogy a felhasználó ne a projekt végén lássa először a terméket. Félreértések elkerülése. Ezért a végső átadás előtt több prototípust is szállítsunk le, hogy minél hamarabb tisztázódjanak a félreértések.

A követelményeket finomítani lehet a prototípusok használatával, mert a user meg tudja fogalmazni, hogy miért nem felel meg az elvárásainak, ha látja. A modern módszertanok többsége alkalmazza, van, hogy minden iteráció végén, minden mérföldkőhöz kötnek egy prototípust.

Akkor érdemes használni, ha a felhasználó és a rendszer között sok lesz a párbeszéd.

Iteratív, nem strukturált, prototípus alapú, gyakran rapid, agilis, vagy extrém, könnyűsúlyú, követelményközpontú.

Scrum

Agilis módszertan. Nagy a jelentősége a csapaton belüli összetartásnak, kommunikációnak. Arra épít, hogy fejlesztés közben a megrendelő igényei változhatnak.

Sprint: megbeszélt hosszúságú fejlesztési időszak, Sprint Planning-gel kezdődik és Retrospective-vel zárul. Addig kell ismételni, amíg a Product Backlogról el nem tűnnek a megoldásra váló sztorik. Minden sprint végén rendelkezni kell egy prototípussal, potenciálisan leszállítható szoftverrel.

Akadály: csak munkahelyi lehet, magánéleti nem.

Sztori: taszkokra bontható egység. A Product Backlog tartalmazza.

Burn down chart: segít megmutatni, hogy az ideális munkatempóhoz képest hogy áll a csapat a sprinten belül.

A Scrum iteratív, objektum vagy szervizorientált, prototípus alapú, rapid vagy agilis, csapatközpontú, könnyűsúlyú.

A módszertan szerepköröket, megbeszéléseket és elkészítendő termékeket ír elő.

A PO létrehozza a Product Backlogot, ahová felveszi a sztorikat (feladatokat) és priorizálja őket. Ezekből választ a csapat az adott sprintre megvalósítandó feladatokat (Sprint Planning). A

csapat a prioritás mellé súlyozást rendel az alapján, hogy kb. mennyi idő elkészíteni. Így azonos prioritás mellett a kevesebb munkát igénylő elemnek nagyobb a megtérülése (Return of Inevestment: ROI). A sprint 2-4 hétig tartó fejlesztési fázis. Naponta van meeting, ahol mindenki elmondja, hogy mit csinált, mit fog csinálni és milyen akadályokba ütközött (Daily Meeting). A sprint végén áttekintik az elkészült feladatokat, vagy elfogadják, vagy nem (Sprint Review). Aztán a felmerült problémákat is áttekintik (Sprint Retrospective).

Utóbbi nagyon fontos, mert ez hozza elő a problémákat, amelyek alapján a következő sprintet jobban meg lehet szervezni.

Szerepkörök

Disznók: elkötelezettek a projekt sikerében, Scrum Master, PO, team

Csirkék: érdekeltek, de haszonélvezői a sikernek, ha nem sikerül, nem az ő felelősségük, üzleti szereplők, menedzsment

SM: felügyeli és megkönnyíti a folyamat fenntartását, segíti a csapatot, megoldja a felmerülő problémákat, akadályokat. Lényegében projekt menedzser.

PO: a megrendelő szerepét tölti be, felelős azért, hogy a csapat mindig azt a részét fejlessze a terméknek, amely éppen a legfontosabb. Nem lehet azonos SM-mel.

Csapat: felelősek azért, hogy a sprintre bevállalt feladatokat elvégezzék, 5-9 fő. Fejlesztők, tesztelők, elemzők.

Üzleti szereplők: megrendelők, tulajdonosokat. Általa jön létre a projekt, aki majd hasznát látja a termék elkészítésének

Menedzsment: feladata a megfelelő környezet felállítása a csapatok számára.

Extrém programozás

Agilis módszertan. Az egyéb módszertanokból a jól bevált technikákat veszi át, minden mást feleslegesnek tekint. 4 tevékenységet ír elő:

- kódolás
- tesztelés: extrém sok tesztelés megtalálja a hibákat, a tesztelés maga a dokumentáció, unit tesztek, nincs kövspec, hanem átvételi teszteseteket fejlesztenek
- odafigyelés: meg kell érteni a megrendelő igényeit
- tervezés: ne ad hoc megoldások legyenek, komponensek legyen függetlenek egymástól, amennyire lehet, OOP

5 értéket vall:

- kommunikáció: dokumentációt helyettesíti, XP szerint azt úgysem olvassa el senki
- egyszerűség: a legegyszerűbb megoldástól induljunk el, emiatt lehet később újra kell írni mindent, de megoldja azt a problémát, hogy lefejlesszünk valami nehezet, amire később nem is lesz szükség
- visszacsatolás: visszacsatolás, ami alapján javíthatjuk a rendszert. Rendszer felől: unit tesztek, integrációs tesztek, elfogadási tesztek (milyen messze vagyunk a kiszállítható megoldástól), megrendelő felől: prototípus bemutatások alkalmával, csapat felől: új igény esetén a csapat elmondja, hogy mennyi a becsült kifejlesztési idő

- bátorság: pl. elég bátrak vagyunk a legegyszerűbb megoldást választani, aztán holnap lehet újra kell írni az egészet
- tisztelet: nem szabad olyan kódot feltölteni, ami nem megy át a unit-testeken, mert ez a többiekkel szemben tiszteletlenség

Technikái:

- páros programozás: az egyik programozó írja, a másik figyeli a kódot. Ha a figyelő hibát lát, vagy nem érti, szól. Folyamatosan megbeszélik, hogy hogyan érdemes megoldani a problémát.
- teszt vezérelt fejlesztés: a metódus elkészítése előtt megírjuk a hozzá tartozó unit-tesztet. Folyamatosan mindig előre írjuk meg az adott bővítéshez szükséges tesztet, és aztán úgy bővítjük a metódust, hogy a tesztet átmenjen.
- forráskód átnézés (review): elkészült nagyobb modulokat egy vezető fejlesztő átnézi. A fejlesztők elmagyarázzák mit miért csináltak. A vezető fejlesztő elmondja hogyan lehet jobban.
- folyamatos integráció: a nap vagy hét végén a verziókezelő rendszerbe bekerült kódokat integrációs teszteljük, hogy kiderüljön, képesek-e együttműködni. Így korán kiszűrhető a programozók közötti félreértés.
- kódszépítés (refactoring): a már letesztelt, működő kódban lehet szépíteni a lassú, rugalmatlan vagy csúnya részeket. Előfeltétele, hogy legyen sok unit-teszt. A funkcionalitást nem szabad megváltoztatni. Minden unit-tesztet le kell futtatni a végén, nem csak a módosított részhez tartozókat.

Akkor működik jól, ha a megrendelő biztosítani tud egy munkatársat, aki átlátja a megrendelő folyamatait és tudja, mire van szükség. Ha a változó követelmények miatt át kell írni elkészült részeket, akkor az XP rossz választás.

Kockázatmenedzsment

Mennyire sebezhető az informatikai rendszer, és ha ez megtörténik mekkora lesz a kár. Két vetület: bekövetkezés valószínűsége, kár mértéke. Csökkentenünk kell mindkettőt, szorzatuk adja a kockázat súlyosságát.

Kockázatcsökkentés példa: vírusirtók, banknál tranzakciók biztonsága

Lépései: kockázat azonosítása, értékelése, csökkentése, kommunikációja

b) Az informatika logikai alapjai: Nulladrendű logika szintaxisa és szemantikája. Igazságtábla. Normálformák nulladrendű logikában, CNF-re hozás algoritmusa, Tseitin transzformáció és Plaisted-Greenbaum kódolás. Rezolúció nulladrendű logikában. SAT, DPLL és DIMACS. SMT és SMT-LIB. Elsőrendű logika szintaxisa és szemantikája.

Nulladrendű logika: minden objektuma igaz/hamis típusú. Logikai változókat kapcsolunk össze logikai műveletekkel.

Elsőrendű logika: tetszőleges típusú objektumokon tudunk logikai állításokat megfogalmazni. A változók objektumokat vehetnek fel értékül, ezeken logikai függvényeket (predikátum) hívunk, majd ezeket kötjük össze logikai műveletekkel.

Szintaxis

Azt írja le, hogy formailag hogyan lehet helyes kifejezéseket csinálni az adott nyelven.

Definíció (Formula):

Adott ítéletváltozóknak egy véges, nem üres V halmaza. Formulának nevezzük a következő kifejezéseket:

1. Ítéletváltozó: Ha $A \in V$, akkor A formula.
2. Negáció: Ha A formula, akkor $\neg A$ is formula.
3. Ha A és B formulák, akkor a következő kifejezések is formulák:
   a. Konjunkció: $(A \wedge B)$
   b. Diszjunkció: $(A \vee B)$
   c. Implikáció: $(A \Rightarrow B)$
   d. Ekvivalencia: $(A \Leftrightarrow B)$
   e. Kizáró vagy: $(A \oplus B)$

Minden formula a fenti szabályok véges sokszori alkalmazásával áll elő.

Példa: Ha Kristóf bejárt órákra és sikerült a dolgozata, akkor megkapja a kettest.

K = „Kristóf bejárt órákra”

D = „Kristófnak sikerült a dolgozata”

E = „Kristóf megkapja a kettest”

(K && D) => E

Zárójelezéssel lehet befolyásolni a kiértékelési sorrendet. Formalizálásánál a lényegtelen dolgokat elhagyjuk, próbáljuk logikai kifejezésekkel megfogalmazni az állítást: nincsenek igeidők pl.

Szemantika

A kifejezések jelentésével foglalkozik, nulladrendű logika esetében a kifejezések igazságértékével. Ennek egyik eszköze az igazságtábla. A szemantika vizsgálja, hogy az egyes interpretációkban mi a kifejezés igazságértéke.

Igazságtábla

Összetett logikai kifejezések kiértékelésében segít, vagy szabályokat fogalmazhatunk meg vele, amelyek mindig úgy történnek, például a konjunkció, implikáció.

Negáció:

A	$\neg A$
0	1
1	0

Konjunkció, diszjunkció, implikáció, ekvivalencia, kizáró vagy:

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$	$A \oplus B$
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	1	1	0

Reprezentáció igazságfüggvényekkel

Az előbbi formula az alábbi háromváltozós igazságfüggvényt *(Boole-függvényt)* reprezentálja.

P	Q	R	$P \vee \neg Q$	$P \wedge Q \vee R$	$P \vee \neg Q \Rightarrow P \wedge Q \vee R$
0	0	0	1	0	**0**
0	0	1	1	1	**1**
0	1	0	0	0	**1**
0	1	1	0	1	**1**
1	0	0	1	0	**0**
1	0	1	1	1	**1**
1	1	0	1	1	**1**
1	1	1	1	1	**1**

Minden formula egy n-változós igazságfüggvényt ad meg. így az n változós függvények száma 2^n. Egy négy változóból áll formula esetén ez a szám 16.

Interpretáció: azt mondjuk, hogy i. függvény az F formula egy interpretációja, ha F formula minden atomjához hozzárendeli az igaz és a hamis állítások közül az egyiket. Pl.: az igazságtábla egy sora.

Tautológia: egy formula tautológia, ha minden interpretációban igaz.

Kontradikció (kielégíthetetlenség): egy formula ellentmondásos, ha minden interpretációban hamis.

Kielégíthetőség: egy formula kielégíthető, ha legalább egy interpretációban igaz.

Normálformák

Literál: atom vagy annak negáltja

KNF: konjunktív normálforma. Klózok konjunkciója, azaz literálok diszjunkciójának konkjunkciója. AVB&(-AVBVC)&(AV-B)

DNF: diszjunktív normálforma. Elemi konjunkciók diszjunkciója, azaz literálok konjunkciójának diszjunkciója. (A&-B&C)V(A&B)

KNF-re hozás algoritmusa

Diszjunktív és konjunktív normálformák előállításának lépései

1. lépés ekvivalenciák és implikációk eliminálása

átírási szabályok:

$$(P \Leftrightarrow Q) \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

$$P \Rightarrow Q \equiv \neg P \vee Q$$

2. lépés negációk atomokhoz kötése

de Morgan azonosságok:

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

3. lépés disztributivitás azonosságaival a kívánt forma előállítása

disztributivitás azonosságai:

$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

Tseitin transzformáció

A KNF/DNF-re hozás algoritmusa elméletben exponenciális, mert vannak részformulák, amik duplikálással járnak, például a disztributivitás. Ezért a klózok csökkentése érdekében használjuk a Tseitin transzformációt, ami lineáris algoritmus.

A formula összes olyan A részformuláját, ami nem literál felülírjuk egy új X ítéletváltozóval majd konjugálunk hozzá egy X ⇔ A ekvivalenciát. Kintről befelé érdemes haladni, és bevezetni a teljes formulára egy X1-et, utána annak részformulájára X2-őt és addig folytatjuk, amíg el nem érjük a literálok szintjét. Kapcsos zárójelek lefelé: ha az egész X1, akkor az például állhat

X2-ből és X3-bó, X2 állhat X4-ből és X5-ből stb. Az eredményben a külső operátorok konjunkciók lesznek, tehát innen már könnyű KNF-et csinálni.

Plaisted-Greenbaum kódolás

A Tseitin transzformáció sok klózt állít elő, ami radikálisan csökkenthető. Felülvizsgáljuk az ekvivalenciákat és eltüntetjük őket, ahol lehet. Az ekvivalencia két egymással szembe mutató implikáció, ha az egyikről sikerül belátni, hogy szükségtelen, akkor azzal spórolunk.

X ⇔ A marad, ha X mindkét polaritással szerepel. Ha csak +, akkor X => A marad, ha csak negatív, akkor A => X marad.

Rezolúció

Cáfoló eljárás, ahol a bizonyítandó állítás tagadásából indulunk ki és bizonyítjuk, hogy levezethető belőle az üres klóz, azaz logikai ellentmondás (UNSAT). A tagadásról bizonyítjuk, hogy kielégíthetetlen.

Az A formula tautológia, ha -A-ból levezethető az üres klóz.

A P1…Pk formulák logikai következménye C, ha P1&Pk&-C-ből levezethető az üres klóz.

Tekintsük $C_1 = P \vee C'_1$ és $C_2 = \neg P \vee C'_2$ klózokat, ahol P ítéletváltozó!

A $C'_1 \vee C'_2$ klózt a C_1 és C_2 **rezolvensének** nevezzük. Az egymással ellentéteseket vesszük ki belőle, és ami marad az a rezolvens.

1. Válasszunk két eddig még nem rezolvált klózt: C_1 és C_2
2. Állítsuk elő C_1 és C_2 **rezolvensét** (ha van) és adjuk a klózhalmazhoz!
3. Ha az **üres klóz** (□) előáll, akkor a klózhalmaz UNSAT. Állj!
4. Vissza az 1. pontra!

SAT

Kielégíthető-e egy KNF-ben szereplő formula? NP-teljes probléma, vagyis nem ismert rá polinom időbonyolultságú algoritmus, csak exponenciális. Ha legalább 1 háromhosszú klóz van, akkor már nincs polinomiális együttható.

3-SAT: (X1v-X2vX3) & (X2v-X1), NP-teljes

2-SAT-ra van polinomiális megoldó algoritmus

DPLL

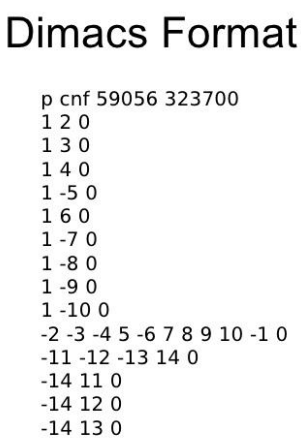
A sima rezolúció nagy klózhalmazon nem elég hatékony. Visszalépéses keresést használ, rekurzív algoritmus. Az egységklózokat kiemelten kezeli, p.: A, -B.

A **unit propagáció** a rezolvensképzés egy speciális esete:

Tekintsük $C_1 = P$ és $C_2 = \neg P \vee C'_2$ klózokat! Az eredményklóz rövidebb lesz, ha unit klózzal rezolváljuk. A C'_2 klóz a C_1 és C_2 rezolvense.

DIMACS

Egységes formátum klózhalmazok megadására. Minden ítéletváltozót sorszám jelöl. Negáció: -, elválasztójel szóköz, klózvégjel 0. Fejléc: p cnf <változó db> <klóz db>



SAT solverek standard input formátuma.

SMT

Magasabb szintű feladatleírásra és megoldásra. Nulladrendű logikát kiegészíthetjük egész/valós számokkal, aritmetikával, tömbökkel, listákkal.

SMT: kielégíthetőség vizsgálata bizonyos elméletekre vonatkoztatva.

SMT-LIB

Az SMT szolverek standard input formátuma.

Komplexebb, emberközelibb, mint a DIMACS. Szöveges utasítások, típusok, aritmetikai operátorok, függvényhívások. Szoftververifikáció, rendezések, logikai fejtörők, aritmetikai problémák.

Szintaxis

Lehetőség van „van olyan X, amelyre teljesül” és „minden X-re teljesül” típusú állítások megfogalmazására. Ezeket a Létezik (∃) és a Minden (∀) kvantor teszik lehetővé.

Az állítások objektumokra (termekre) vonatkoznak. Nulladrendűben nem lehetett tárgya egy állításnak! A állítások paraméterezhetőek termekkel. Paraméterezhető állítás = predikátum.

Példa:

0. Szeretem a sört → S atom
1. Sz(s), több paraméterrel: Sz(én, s)

Nevesített term: konstans (sör)

Elsőrendű logikában a szintaxisát tekintve használunk predikátumokat, kvantorokat, zárójeleket, változókat, függvényjeleket.

Definíció: azt mondjuk, hogy a $P(t_1, t_2, t_n)$ szimbólum sorozat atomi formula, vagy röviden atom, akkor és csak akkor, ha P egy n paraméteres predikátum szimbólum, azaz egy n paraméteres kijelentés, és t_1, t_2, t_n paraméterek termek, azaz objektumok.

Függvény: termekkel paraméterezett és termet ad vissza: „Mindenki fiatalabb a saját anyjánál”: ∀ x F(x, anya(x))

Szemantika

Kötött és szabad változók. Kötött változók átnevezése. Kvantorok hatásköre.

Interpretáció: meghatározza azon értékek körét, melyeket változóink és konstansaink felvehetnek. Ezek már nem csak logikai értéket tárolhatnak, hanem bármilyen objektumot, ezért az interpretáció először definiálja az objektumtartományt, az ún. domaint.

Konstansok interpretálása: mindhez konkrét értéket rendel a domainből.

Predikátumok interpretálása: mire ad vissza igazat, mire hamisat. Összes paraméterezést le kell fedni!

Függvények interpretálása: minden függvényhez megadunk egy olyan visszatérési érték táblázatot, ami a domainnek megfelelő összes lehetséges paraméterezést lefedi.

Példa: minden hallgató teljesítette a logika vizsgát. Az igazságérték függ attól, hogy melyik egyetem hallgatóiról van szó (melyik egyetem hallgatói a domain részei).

12. tétel

a) Adatbázisrendszerek II: A PL/SQL alapjai: típusok, változók, konstansok, vezérlési szerkezetek. SQL utasítások elhelyezésének és használatának lehetőségei a PL/SQL-ben. A PL/SQL program felépítése. Blokkok és alprogramok. Tárolt eljárások, tárolt függvények összehasonlítása, csomagok készítésének célja és lehetőségei. A tárolt alprogramok paraméterezési lehetőségei, az egyes lehetőségek jellemzői. Kivételek kezelés PL/SQL-ben.

PL/SQL

Típusok

- NUMBER(h, t): numerikus adatok, hossza <=38, h=szám hossza, t=tizedesjegyek száma
- DATE: dátum és idő
- VARCHAR2(méret): karakter típus, hossza változó (max hossza <=4000)
- CHAR(méret): karakter típus, hossza fix (max hossza <= 4000)
- NCHAR(méret): azonos a CHAR-ral, de a max hossza függ a karakterkészletétől
- LOB(large objects): bináris vagy szöveges formátum. Lehet kép, hang vagy nagyméretű szöveges állomány. Ezen belül lehetnek:
    - LONG: szöveges adatok, max hossza <= 2 GB
    - LONG RAW: bináris adatok, max hossza <= 2 GB
    - LONG-ot és LONG RAW-t csak egyszerű adatfeldolgozásban alkalmazzák.
    - CLOB, NCLOB, BLOB: belső LOB típusok, az adatbázisban tárolódnak, max hossza <= 4GB
    - CLOB: adatokat az adatbázis karakterkészlete alapján tárolja
    - NCLOB: adatokat a nemzeti karakterkészlet alapján tárolja

Változók

név típus [[NOT NULL {:= | DEFAULT} kifejezés];

v_name VARCHAR2(20) NOT NULL := ’John Smith’;

Konstansok

név CONSTANT típus [NOT NULL] {:= | DEFAULT} kifejezés;

%TYPE: egy korábban már deklarált kollekció, kurzorváltozó, mező, objektum, rekord, adatbázistábla-oszlop vagy változó típusát veszi át és ezzel a típussal deklarálja a változót vagy nevesített konstanst. Használható még tábla oszlopának típusmegadásánál is. Két előnye van: nem kell pontosan ismerni az átvett típust, illetve ha az átvett típus megváltozik, a változó típusa alkalmazkodik hozzá futási időben.

Vezérlési szerkezetek

Elágazások

```
IF condition THEN

statements;

[ELSIF condition THEN
```

```
statements;]

[ELSE

statements;]

END IF;
```

```
CASE [szelektor_kifejezés]

WHEN {kifejezés | feltétel} THEN

utasítás [utasítás]…

[WHEN {kifejezés | feltétel} THEN

utasítás [utasítás]…]…

[ELSE

utasítás [utasítás]…]

END CASE;
```

Ciklusok

Fajtái:

- alapciklus (végtelen), nincs információ az ismétlésre vonatkozóan
- WHILE (előfeltételes)
- FOR (előírt lépésszámú)
- kurzor FOR

Egy ciklus befejeződhet, ha:

- az ismétlődésre vonatkozó információk ezt kényszerítik ki
- a GOTO-val kilépünk a magból
- az EXIT-tel befejeztetjük
- kivétel váltódik ki

Alapciklus

```
[címke] LOOP
utasítás
[utasítás]…
EXIT [WHEN condition];
END LOOP [címke];
```

While

```
[címke] WHILE feltétel
LOOP utasítás [utasítás]…
END LOOP [címke];
```

For

```
[címke] FOR ciklusváltozó IN [REVERSE] alsó_határ…felső_határ

LOOP utasítás [utasítás]…

END LOOP [címke]
```

EXIT: bármely ciklus magjában kiadható, de azon kívül nem. `EXIT [címke] [WHEN feltétel];`

SQL utasítások elhelyezése és használata PL/SQL-ben

SELECT INTO: egy vagy több adatbázistáblát kérdez le és a származtatott értékeket változókba vagy egy rekordba helyezi.

```
SELECT select_list INTO {variable_name[, variable_name] FROM [table_name]
WHERE valami = valamiérték
```

DELETE: egy adott tábla vagy nézet sorait törli

```
DELETE FROM table_name WHERE valami = valamiérték
```

INSERT: új sorokkal bővít egy táblát vagy nézetet

```
INSERT INTO table_name(param1, param2, …) VALUES (value1, value2, …)
```

UPDATE: megváltoztatja egy adott tábla vagy nézet adott oszlopainak értékét

```
UPDATE table_name SET salary = salary + v_sal_increase WHERE job_id =
'ST_CLERK'
```

MERGE: többszörös INSERT és DELETE elkerülésére

```
MERGE INTO tábla [másodlagos_név]

USING {tábla|nézet|alkérdés} [másodlagos_név] ON (feltétel)

WHEN MATCHED THEN UPDATE SET oszlop={kifejezés|DEFAULT}

[,oszlop={kifejezés|DEFAULT}]…

WHEN NOT MATCHED THEN INSERT(oszlop[,oszlop]…)

VALUES ({DEFAULT|kifejezés[,kifejezés]…}];
```

PL/SQL program felépítése

Blokkok és alprogramok

A blokk az alapvető programegység. Lehet önálló vagy beágyazva más programegységbe, megjelenhet bárhol a programban, ahol végrehajtható utasítás állhat. Lehet címkézett vagy címkézetlen (anonim). Utóbbi akkor működik, ha szekvenciálisan rákerül a vezérlés. A címkézettre lehet GOTO utasítással ugrani.

Felépítése:

[címke] [DECLARE deklarációk]

BEGIN utasítás [utasítás]…

[EXCEPTION kivételkezelő]

END [címke];

Deklaráció és kivételkezelő opcionális, végrehajtó kötelező rész.

Tárolt eljárások

Nincs visszatérési érték.

Az alprogramok az absztrakció, újrafelhasználhatóság, modularitás és a karbantarthatóság eszközei. Paraméterezhetőek, hívhatóak a felhasználás helyén. Alapértelmezés szerint rekurzívan hívhatók. Két fő részük van: specifikáció és törzs.

Felépítése:

```
specifikáció

CREATE [OR REPLACE] PROCEDURE procedure_name

[(parameter1 [mode] datatype1,

parameter2 [mode] datatype2, …)]

IS|AS

[local_variable_declarations; …]

BEGIN

-- actions;

END [procedure_name];
```

A blokk működése befejeződik, ha elfogynak az utasítások, ekkor beágyazott blokknál a blokkot követő utasításra, egyébként a hívó környezetbe kerül a vezérlés; kivétel keletkezik, GOTO ugrás, RETURN utasítás

Tárolt függvények

Van visszatérési érték, így a hívókörnyezet számára valamilyen információt közvetít.

Felépítése:

```
CREATE [OR REPLACE] FUNCTION function_name

[(parameter1 [mode1] datatype1, …)]

RETURN datatype IS|AS

[local_variable_declarations; …]

BEGIN
```

```
-- actions;

RETURN expression;

END [function_name];
```

Procedure	Function
PL/SQL utasításként futtatható	**Kifejezés részeként futtatható**
A header-ben nincs RETURN	**A header-ben tartalmaznia kell RETURN-t**
Az output paraméterben visszaadhat értékeket	**Csak egy értéke ad vissza**
Tartalmazhat egy RETURN utasítást érték nélkül	**Kötelező legalább egy RETURN utasítást tartalmaznia**

Mindkettőnek lehet 0 vagy több IN paramétere, amelybe a hívó környezetből kerülhet érték.
Mindkettőnek standard blokk struktúrája van, beleértve a kivételkezelést.

Csomagok

Egy olyan PL/SQL tároló, amely adatbázis-objektumokat és PL/SQL programozási eszközöket (alprogramok, változók, kurzorok, kivételek) tartalmaz. Részei: specifikáció és törzs.

Specifikáció: interfészt biztosít, amelyen keresztül hozzáférhetünk a csomag eszközeihez. Típusdefiníciók, nevesített konstans, változó, kivétel deklarációk, kurzorspecifikáció, alprogramspecifikációk és pragmák vannak benne. Futtatható kódot nem tartalmaz. A hívó környezet csak ezt látja.

Törzs: kurzorok és alprogramok teljes deklarációja és esetlegesen végrehajtható utasítások. Nem kötelező elem. A törzs tartalma a hívó környezetnek nem látható.

Csomagspecifikáció:

```
CREATE [OR REPLACE] PACKAGE csomagnév

[AUTHID {DEFINER|CURRENT_USER}]

{IS|AS}

{típus_definíció|

nevesített_konstans_deklaráció|

változó_deklaráció|

kivétel_deklaráció|

kurzor_specifikáció|

alprogram_specifikáció|

pragma}
```

```
END [csomagnév];
```

Törzs:

```
CREATE [OR REPLACE] PACKAGE BODY csomagnév

{IS|AS}

deklarációk

[BEGIN

végrehajtható_utasítások]

END [csomagnév];
```

Tárolt alprogramok paraméterezése

Háromféle paraméterátadási mód van: IN, OUT és IN/OUT.

- IN: default mód, érték szerinti. A formális paraméter az alprogramra nézve konstans, csak olvasható.
- OUT: eredmény szerinti. A formális paraméter az alprogramon belül csak írható. Az aktuális paraméter visszatéréskor felveszi a formális értékét.
- IN/OUT: az aktuális paraméterről a híváskor másolat készül, amelynek értéke az alprogram futása alatt független az aktuális paraméter értékétől. Visszatéréskor az aktuális paraméter felveszi a formális paraméter értékét.

- Az alprogramok paraméterszáma kötött, változószámú nem lehet.
- A paramétereknek lehet default értéket adni, vagyis hívható kevesebb paraméterrel.
- Az alprogram azonosítók túlterhelhetőek mindaddig, amíg van különbség a paraméterlistában.
- Híváskor a formális és aktuális paraméterlista összerendelése pozíció, név vagy mindkettő szerint történik.

Kivételkezelés

A kivételek futás közben bekövetkező események. Egy kivételhez egy kód és egy üzenet tartozik. A beépített üzenetek kódja negatív – egyet kivéve –, a felhasználói kivételeké pozitív.

Fajtái:

- beépített kivételek, amiket a futtató rendszer vált ki (STANDARD csomag)
- felhasználói kivételek, amelyeket a programegység deklarációs részében határoztunk meg
- nem definiált Oracle-szerver hibák

Példa beépített kivételre:

```
DECLARE
v_country_name wf_countries.country_name%TYPE := 'Korea, South';
v_elevation wf_countries.highest_elevation%TYPE;
```

```
BEGIN
SELECT highest_elevation INTO v_elevation FROM wf_countries WHERE
country_name = v_country_name;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ('Country name, '|| v_country_name ||', cannot be
found. Re-enter the country name using the correct spelling.');
END;
```

Bármely megnevezett kivétel kiváltható: `RAISE kivételnév;`

Felhasználói kivétel deklarálása:

```
DECLARE
sajat_kivetel EXCEPTION;
```

Több kivétel kezelése: bármely programegység végén az EXCEPTION alapszó után. WHEN ágak, amelyek név szerint kezelik a kivételeket, legutós ág az OTHERS, ami minden kivételt kezel.

```
WHEN kivételnév [OR kivételnév]…
THEN utasítás [utasítás]…
[WHEN kivételnév [OR kivételnév]…
[THEN utasítás [utasítás]…]…
[WHEN OTHERS THEN utasítás [utasítás]…]

EXCEPTION
WHEN NO_DATA_FOUND THEN
statement1;
…
WHEN TOO_MANY_ROWS THEN
statement2;
…
WHEN OTHERS THEN
statement3;
```

Beépített kivételek: NO_DATA_FOUND, TOO_MANY_ROWS, INVALID_CURSOR, ZERO_DIVIDE

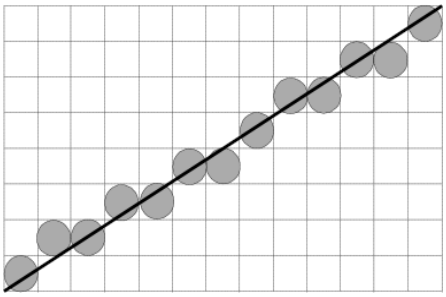
b) Bevezetés a számítógépi grafikába: Raszteres algoritmusok: DDA és MidPoint szakaszrajzoló-, MidPoint körrajzoló algoritmusok, Cohen-Sutherland vágó algoritmus. Paraméteres görbék, Hermit ív, Bézier görbe, B-Spline. Pont-transzformációk síkban, térben, homogén koordináták. Tér leképezése síkra: párhuzamos és centrális vetítés, axonometria. Poliéderek megadása. Wire Frame modell, B-Rep adatstruktúra. Poliéderek megjelenítése: hátsó lapok eltávolítása, Z-Buffer algoritmus, Flat-, Gouraud- és Phong árnyalás.

Raszteres algoritmusok

A modellezés során arra keressük a választ, hogy hogyan lehet folytonos geometriai alakzatokat képpontokkal közelíteni. Azokat az algoritmusokat, melyek erre választ adnak digitális differenciaelemző (DDA) algoritmusoknak nevezzük.

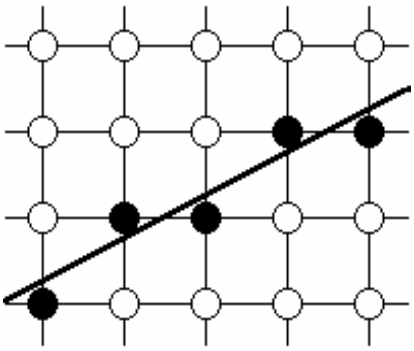
DDA

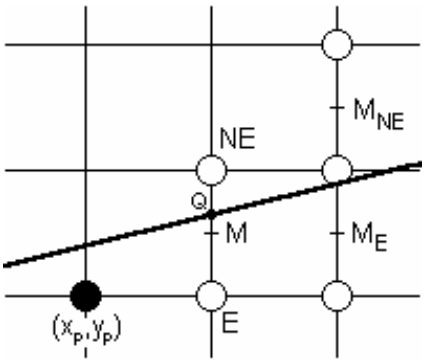
Az egyesen y = mx + b alakú egyenletéből indulunk ki. Ekkor megkapjuk az egyenes y tengelypontját (0,b). Ezután az m = dy / dx segítségével meghatározzuk, hogy mennyit lépjünk balra és mennyit felfelé, hogy megkapjuk az egyenes következő pontját. Nem szép a kép, mert nekünk kell kerekítenünk a kapott (x,y) értékeket, mert az ideális egyenes helyett csak közelítő raszter pontokat kaptunk. Ezen lehet segíteni, ha dx-et és dy-t kedvezőbben választjuk meg. A nagyobb elmozdulás irányában legyen a növekmény egy, tehát Max(dx,dy) = 1.



MidPoint szakasz

Legyenek az egyenest meghatározó pontok P1(x1,y1) és P2(x2,y2), tegyük fel, hogy a meredekség 0 <= m <= 1, tehát kisebb, mint 45 fok. Az egyenest balról jobbra haladva rajzoljuk ki. A kitöltött körök a megvilágított pixeleket jelentik. Legyen az éppen megjelenített pont P(xp,yp), ekkor a következő megrajzolandó raszterpont az E és az NE pontok valamelyike lehet. Közülük kell azt kigyújtani, amelyik közelebb van az elméleti egyeneshez. A két rácspont között lévő M felezőpont segítségével történik.



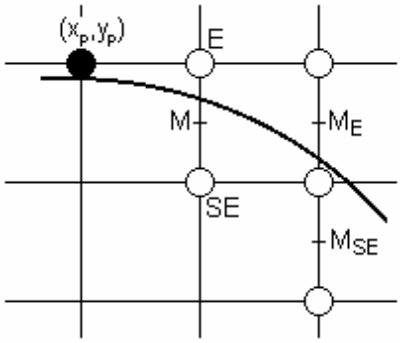


MidPoint kör

Kör: pontok halmaza a síkban, amelyek adott, a síkra illeszkedő C ponttól egyenlő (r>0) távolságra helyezkednek el. Ha rögzítünk egy (x,y) koordináta-rendszert, akkor az origó középpontú, r sugarú kör egyenlete $x^2+y^2=r^2$. Ebből az (u,v) középpontú, r sugarú kör egyenlete $(x-u)^2+(y-v)^2=r^2$.

Nyolcas szimmetria elve: tekintsünk egy origó középpontú kört. Ha egy (x,y) pont rajta van a körvonalon, akkor könnyen meghatározatunk három másik pontot, ami szintén rajta lesz. Ezért, ha meghatározzuk a kör egy nyolcadának pontjait, akkor a teljes kört is meghatároztuk. Ezzel gyorsítható az algoritmus, de feltétel, hogy a középpont az origó legyen. Ha nem az, akkor koordináta transzformációt alkalmazunk. A koordináta-rendszer origóját a kör (u,v) középpontjába toljuk. Kirajzolás előtt a pontokat (u,v) vektorral eltoljuk, és így a jó helyen rajzoljuk ki a kört.

Legyen az aktuális kivilágított pixel P(xp,yp), az elméleti körhöz legközelebb eső pont. A következő megrajzolandó pint E vagy SE lehet. Közülük kell azt kigyújtani, amelyik közelebb van az körívhez. A két rácspont között lévő M felezőpont segítségével történik.



Cohen-Sutherland vágó

A síkot 9 részre osztjuk. A középső kilenced a képernyő. Négyjegyű bináris kódot rendelünk minden kilencedhez az oldalegyenesekhez való viszonyuk alapján:

- az 1. bit 1, ha a végpont a felső vízszintes vonal felett van, egyébként 0
- a 2. bit 1, ha a végpont az alsó vízszintes vonal alatt van, egyébként 0
- a 3. bit 1, ha a végpont a jobb oldali függőleges vonaltól jobbra van, egyébként 0
- a 4. bit 1, ha a végpont a bal oldali függőleges vonaltól balra van, egyébként 0

1001	1000	1010
0001	0000	0010
0101	0100	0110

- Ha a két kódnak van olyan bitje, hogy azonos helyi értéken egyes van, akkor a szakasz kívül esik a képernyőn.
- Ha a szakasz 2 végpontjához rendelt kód csupa 0, akkor a képernyőn belül van.
- Ha a két végpont közül valamelyik kódjában van egyes, akkor az egyes helyiértéknek megfelelően képernyő éllel el kell metszeni a szakaszt, és a metszéspontra módosítani

a szakasz végpontját és az új kódot újból meg kell vizsgálni. Ezt addig csináljuk, amíg a szakasz végpontjaihoz rendelt kódok csupa 0-ák nem lesznek.

- Előnye: hatékony, ha a szakaszok nagy valószínűséggel kívül esnek a képernyőn, jól általánosítható 3D-ben 6 bites kódokkal.
- Hátránya: poligon ablakra nem általánosítható, csak téglalapra működik.

Paraméteres görbék

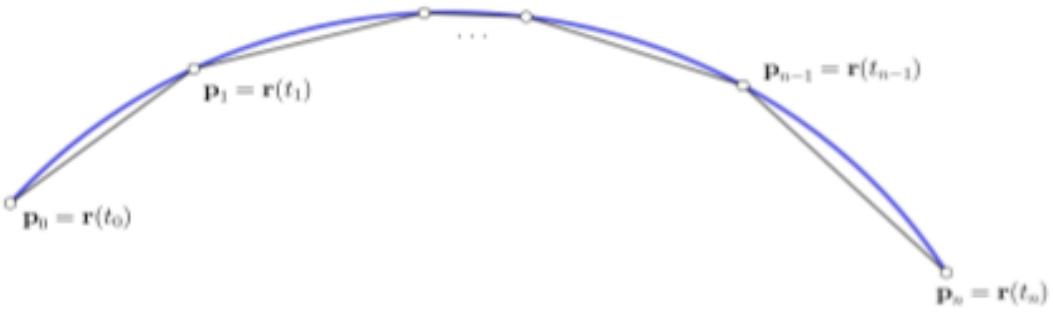
A síkon mozgó pont egy görbét ír le. Ha minden t időpillanatban húzunk egy op vektort r(t), akkor egy I intervallumon értelmezett vektorfüggvényt kapunk. Térben térgörbét kapunk. Az r(t) vektorfüggvényt általában koordinátafüggvényeivel adjuk meg.

- r(t) = (x (t), y (t)) síkban
- r(t) = (x (t), y (t), z (t)) térben

ahol a koordinátafüggvények általában valós változós, valós értékű függvények.

Megjelenítés: A [a,b] ⊂ R intervallumon értelmezett r(t) görbét számítógépen töröttvonallal közelítjük. Ez azt jelenti, hogy kiszámítjuk a p0 = r(t0),p1 = r(t1),...,pn−1 = r(tn−1),pn = r(tn) pontokat, ahol t0 = a és tn = b, majd a pontok által meghatározott szakaszokat megjelenítjük.

A megjelenítés során meg kell határoznunk a szomszédos p_i és p_{i+1} pontok távolságát. Mivel a megjelenítés az [a,b] intervallumon való iterálással történik, így általában azt osztjuk fel egy előre meghatározott mennyiséggel.



Hermite ív, Hermite interpoláció

Interpoláció: adott n darab pont p0,…,pn, melyeket ismerünk, ezek a kontrollpontok. Azt a k-ad fokú polinommal megadott s(u) görbét keressük, ami áthalad ezeken a kontrollpontokon, tehát interpolálja azokat. Módszerek: Lagrange, Newton, stb.

Ha adott egy görbe két végpontja, azzal még nem definiáltuk egyértelműen. Legyenek adottak p0 és p1 pontok, valamint a keresendő görbe végpontjaiban az érintővektorok (p0 és p1 vektorok). Célunk leírni az s(t) görbét a kontroll alakzatok és bizonyos bázisfüggvények lineáris kombinációjaként. A bázisfüggvények: gyorsan számolhatóak, kellően rugalmasak és a velük definiált görbe igazodjon a kontroll adatokhoz.

A Hermite görbénél keresünk egy olyan $s(u)$ harmadfokú polinomot, ahol az input adat két pont és két érintő, de megadható négy ponttal vagy három ponttal és egy érintővel is. Tehát adottak a p_0 és p_1 pontok, valamint a t_0 és t_1 érintő vektorok. Keressük az $s(u) = a_0u^3 + a_1u^2 + a_2u + a_3, u \in [0,1]$ harmadfokú polinommal megadott görbét, amelyre teljesül, hogy

- $s(0) = p_0$
- $s(1) = p_1$
- $s(0)' = t_0$
- $s(1)' = t_1$

Ezután ezekből egyenletrendszert képezve és megoldva azt megkapjuk a Hermite polinomokat:

- $H_0^3(u) = 2u^3 - 3u^2 + 1$
- $H_1^3(u) = -2u^3 + 3u^2$
- $H_2^3(u) = u^3 - 2u^2 + u$
- $H_3^3(u) = u^3 - u^2$

Ezek segítségével felírhatjuk a görbét:

- $s(u) = p_0H_0^3(u) + p_1H_1^3(u) + t_0H_2^3(u) + t_1H_3^3(u)$

Ezek az alappolinomok a görbe súlyfüggvényei. Ha a két végpontbeli érintőket egyre nagyobb mértékben növeljük, miközben irányukat nem változtatjuk, akkor kialakulhat a hurok a görbén.

Hermite-ív hátrányai:

- az érintőkkel való definiálás körülményes
- bizonyos helyzetekben a görbe metszi az érintőt
- bizonyos helyzetekben a görbe metszi önmagát

Bézier görbe

Approximáció: a görbének nem kell áthaladnia, elég ha csak megközelíti a kontrollpontokat tetszőleges mértékben.

A Bézier görbe egy approximációs görbe. Az első és utolsó kontrollpontokon áthalad. Előállítása Bernstein polinom segítségével történik:

$$B_i^n(u) = \binom{n}{i} u^i(1-u)^{n-i}$$

Ezek után a görbe polinomiális alakja:

$$s(u) = \sum_{i=0}^{n} b_i B_i^n(u), u \in [0,1]$$

A Bézier görbe tulajdonságai:

- végpontok interpolációja
- a görbe a kontrollpontok konvex burkán belül marad
- a görbe invariáns a kontrollpontokon végrehajtott affin transzformációkkal szemben, vagyis egy görbe transzformációját úgy is elvégezhetjük, hogy a kontrollpontokat transzformáljuk és újraszámoljuk a görbét. Nem kell minden pontot újra transzformálni.
- a görbe csak globálisan változtatható → ha egy kontrollpont megváltozik, az egész görbe megváltozik (minden pont elmozdul)
- konvex poligon konvex görbét generál, viszont visszafelé nem. Konkáv poligon is generálhat konvex görbét

Bézier görbe esetén a magas fokszám rugalmatlanná teszi a görbét, nem követi kellőképpen a poligont. Nagy mennyiségű kontrollpont esetén a számítási erőforrásigény megnő.

Alacsony fokszámú görbéket csatlakoztatunk egymáshoz úgy, hogy a csatlakozási pontokban automatikusan biztosítjuk a megfelelő szintű folytonosságot.

B-Spline

Bázisfüggvények:

$$N_0(t) = \frac{1}{6}(1-t)^3$$

$$N_1(t) = \frac{1}{2}t^3 - t^2 + \frac{2}{3}$$

$$N_2(t) = \frac{1}{2}t^3 + \frac{1}{2}t^2 + \frac{1}{2}t + \frac{1}{6}$$

$$N_3(t) = \frac{1}{6}t^3$$

Ha felveszünk 4 kontrollpontot (p0,p1,p2,p3) és megrajzoljuk a hozzájuk tartozó görbét, akkor láthatjuk, hogy a görbe nem érinti a végpontokat. Ha felveszünk egy új kontrollpontot, amit jelöljünk p4-gyel, és az előző 4 ponthoz tartozó görbén túl megrajzoljuk a p1,p2,p3,p4 pontokhoz tartozó görbét is, akkor láthatjuk, hogy a végpontokat nem érinti a görbe, de a két kisebb görbe egybeolvadt. Vagyis nem látszódnak a csatlakozási pontok.

A B-Spline görbe tulajdonságai:

- a görbeívek automatikusan csatlakoznak
- nem látjuk a csatlakozási pontot, automatikusan megfelelő a folytonosság

A görbeívek első végpontjai a hozzájuk tartozó kontrollpontok közül az első három által alkotott háromszög középső (p1,p2,p3 esetén p2) pontjából induló súlyvonalának a súlyponttól különböző harmadolópontja. Az érintő a háromszög első pontjából az utolsóba (p1,p2,p3 esetén p1-ből p3-ba) mutató vektor fele. A második végpont ugyanígy számítható, csak ahhoz a görbeívhez tartozó 4 kontrollpont közül az utolsó 3 által alkotott háromszöget kell figyelembe venni.

Homogén koordináták

A tér pontjait olyan rendezett számhármasokkal reprezentáljuk, amelyek arányosság erejéig vannak meghatározva, és mind a 3 koordináta nem lehet egyszerre 0.

- rendezett számhármas: [x1,x2,x3]
- arányosság: [x1,x2,x3] ugyanazt a pontot jelöli, mint a [ωx1, ωx2, ωx3], ahol ω egy 0-tól különböző valós szám.
- [0,0,0] nem létezik

Áttérés Descartes koordinátákról homogén koordinátákra: legyen egy síkbeli pont valós koordinátája [x,y], akkor a homogén alak [x,y,1] lesz. Mivel csak arányosság erejéig vannak meghatározva, most már szorozhatjuk a koordinátákat egy tetszőleges nem 0 számmal.
Visszatérés homogén koordinátákról Descartes koordinátákra: ha egy pont homogén koordinátája [x1,x2,x3] és x3!=0, akkor az első két koordinátát eloszthatjuk az arányossági tulajdonság miatt a harmadikkal: [x1/x3, x2/x3, 1]. Tehát x = x1/x3, y = x2 /x3. Ha x3=0, akkor nincs valós megfelelője a pontnak a síkban.

Pont-transzformáció síkban és térben

A geometriai transzformációk olyan leképezések, melyek a tér minden p pontjához kölcsönösen egyértelműen hozzárendelik a tér valamely p' pontját. Térben 4x4-es mátrixokkal írjuk le, így a tér pontjai homogén koordinátákkal adjuk meg. A p pont transzformáltját úgy kapjuk meg, ha azt balról megszorozzuk a végrehajtandó transzformáció mátrixával.

Eltolás: az eredeti pontot eltoljuk egy adott vektorral. p'=p + v

Forgatás alfa szöggel: a tetszőleges irányvektorú tengely körüli forgatást fel lehet bontani a három tengely körüli forgatások sorozatára. A képernyőn az y tengely állása miatt a szokásos + iránnyal ellentétesen, az óramutató járásával megegyező irányt tekintjük +-nak.

Tükrözés: az egyes koordinátasíkokra való tükrözések során a koordinátasík meghatározásában nem szereplő tengelyhez tartozó koordinátát negáljuk.

Origó középpontú kicsinyítés és nagyítás: az objektumot minden koordinátatengely irányában azonos mértékkel nyújtjuk meg.

Skálázás: az objektumot a koordinátatengelyek mentén eltérő mértékkel nyújtjuk.

Nyírás: a tér pontjait egy adott síkkal párhuzamosan csúsztatjuk el. A sík haladjon át az origón. A csúsztatás mértéke arányos az adott pont egyenestől való d távolságával. A síkot az n normálvektorával adjuk meg, a csúsztatás irányát pedig egy erre merőleges t irányú egységvektorral.

Window-Viewport: egy a képsíkon lévő ablakot transzformál át egy másik, a képernyőn megjelenítendő ablakba. Az eredeti képet eltoljuk az origóba, majd a két ablak oldalarányai alapján skálázást végzünk. Végül a helyes méretű ablakot eltoljuk a végleges pozícióba.

Tér leképezése síkra

Vetítés: térbeli objektumok megjelenítése a sík képernyőn. A vetítés egy n dimenziós objektumot nála alacsonyabb dimenzióba transzformál. Térből síkra a leképezés $R^3 \rightarrow R^2$ transzformáció.

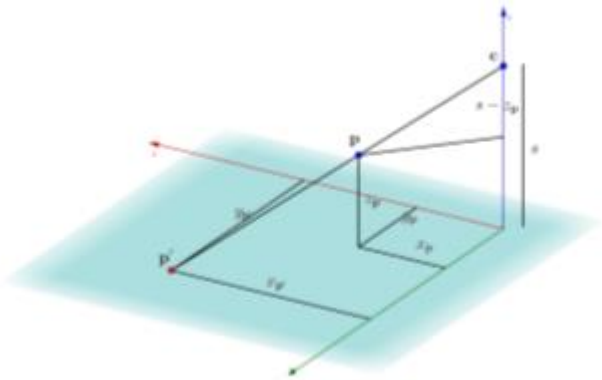
Párhuzamos vetítés

A térbeli p pontot egy adott v irányvektor mentén eltoljuk, míg az a p' pontban elmetszi a képsíkot. Megtartja az egyeneseket, a párhuzamosságot, az arányokat, de nem tartja meg a távolságot és a szöget.

Centrális vetítés

A vetítés centrumát és a vetítés irányát kell definiálni. A vetítés centruma az tengely, a kamera és az origó távolsága s.

Párhuzamos szelők tétele: ha egy szög szárait párhuzamos egyenesekkel metsszük, akkor az egyik száron keletkező szakaszok hosszának aránya egyenlő a másik száron keletkezett megfelelő szakaszok hosszának arányával.



Axonometria

Adott a térbeli Descartes-féle koordinátarendszer az origóval, valamint az e1,e2,e3 egységvektorokkal, továbbá adottak a képsíkon ezek képei, azaz origó', és e1',e2',e3'. A térbeli p pont p' képét úgy kapjuk meg, hogy az eredeti pont koordinátáit megszorozzuk a hozzájuk tartozó egységvektorok képeinek vektoraival, majd összegezzük őket.

Poliéderek

Színes, árnyalt megjelenítéshez a felületeket poligonhálóval kell közelíteni. Általában háromszögeket használunk, mert 4 pont esetén nem biztosított, hogy mindegyik egy síkba esik.

Poliéderek megadása

Wire Frame modell: drótvázmodell, mely egy adott modellnek csak a csúcsait és azok összekötő éleit tárolja. Az így előállított kép sokszor értelmezhetetlen, és a valóban nem látható részek eltávolítása nem lehetséges.

B-Rep adatstruktúra: a Wire Frame továbbfejlesztése. Tárolja a geometriai (csúcspontok koordinátái, élek és lapok egyenlete) és topológiai (lapok, élek és csúcspontok kapcsolata) információkat. Az éleket és háromszögeket legegyszerűbben csúcsokra való mutatókkal adhatjuk meg.

```
STRUKTÚRA ÉL
    VÁLTOZÓK:
        PONT: A, B;
STRUKTÚRA_VÉGE;
```

```
STRUKTÚRA HÁROMSZÖG
    VÁLTOZÓK:
        PONT: A, B, C;
STRUKTÚRA_VÉGE;
```

```
STRUKTÚRA B_REP
    VÁLTOZÓK:
        PONT[]: CSÚCSOK;
        ÉL[]: ÉLEK;
        HÁROMSZÖG[]: HÁROMSZÖGEK;
STRUKTÚRA_VÉGE;
```

Poliéderek megjelenítése

Hátsó lapok eltávolítása

Zárt poliéderek megjelenítését általában a hátsó, nem látható lapok eltávolításával kezdjük. Ehhez elő kell állítani az adott lap egy normálvektorát egy adott pontban (vektoriális szorzat), majd annak az adott pontból a kamerába mutató vektor szögének függvényében (90 foknál nagyobb szöget zárnak be) jelenítsük meg a lapot (skaláris szorzat).

Mielőtt megjelenítenénk a lapokat az átfedések kezelésére rendezzük azokat. A háromszögeket súlypontjuk z koordinátája szerint érdemes rendezni, az így kapott háromszögeket hátulról kezdjük megjeleníteni. Így az elől lévő lapok ki fogják takarni a hátsókat. A rendezéshez érdemes a gyorsrendezést módosítani úgy, hogy a háromszögek fent említett adata alapján rendezzen.

Z-Buffer algoritmus

Poligonokkal határolt alakzatok megjelenítésre való. Az egyes pixeleken vizsgálja, hogy melyik alakzat látszik rajta. Implementálásakor szükség van a színek tárolása mellett az egyes mélységek tárolására is, innen a neve. A mélység puffert maximális mélységre, a pixelekhez pedig a háttérszínt rendeljük. Bejárjuk az összes objektumot, azon belül az objektum vetületének összes pixelét, majd ha az adott pixelhez tartozó eredeti pont z koordinátája nagyobb, mint az ebben a pontban tárolt mélység érték, akkor kiszámítjuk és tároljuk a pixel színét az adott pontban. A mélység értékét az objektum z koordinátájának értékére állítjuk.

Árnyalási technikák

A megjelenítés során természetes igény, hogy az adott testet ne csak háttérszínnel töltsük ki, hanem színezzük is azt ki.

Flat árnyalás

Konstans árnyalás. Minden lap esetén egy pontban kiszámítjuk a színértéket, és ezt alkalmazzuk a teljes lap minden pixelén. Nem kielégítő az eredmény. Akkor lenne az, ha:

az alakzatot megvilágító fénysugarak párhuzamosak

a nézőpont végtelen távol van (párhuzamos vetítés)

az ábrázolandó objektum poliéder, nem pedig egy görbült felület poliéderrel való közelítése

A megjelenítés során általában a felületi normálisok alapján határozzuk meg a színeket. Ekkor a szomszédos lapok egymáshoz való viszony miatt ugrásszerű változások lehetnek. Ezért a csatlakozó lapok közös csúcspontjában kiszámítjuk a normálisokat, majd azoknak vesszük az átlagát és azt rendeljük a megjelenítendő laphoz.

Ekkor az így kapott vektor egyik lapra sem lesz merőleges. Görbült felületek esetén konstans árnyalás helyett folytonos árnyalást érdemes használni.

Gouraud árnyalás

A háromszöglap minden csúcsában kiszámítjuk a színinformációt a normálisok segítségével. A lap élein a csúcspontok színének lineáris interpolációjával határozzuk meg a színt, belső pontok esetén kettős lineáris interpolációval.

Hátrányai:

- a test körvonala nem lesz sima (látszik a poligonnal való közelítés)
- nagy tükröződő felületeknél ugyanúgy ugrásszerű változás mutatkozik

Ezek megoldhatóak a poligonháló finomításával, de akkor megnő a tárigény és a feldolgozási idő.

Phong árnyalás

A színek helyett a normálisokat interpolálja. A lap élei mentén a csúcspontokban lévő normálisok lineáris interpolációjával határozzuk meg a normálist, míg belső pont esetén kettős interpolációval.

Minden pontban kiszámítjuk (interpoláljuk) a felületi normálist, majd annak értéke alapján kiszámítjuk a pontbeli színértékeket.

Több számítást igényel, de szebb eredményt ad. A képhatár szögletessége szintén a poligonháló finomításával lehetséges.

13. tétel

a) Webprogramozás II: A dinamikus weboldalak jellemzői, összehasonlításuk a statikus tartalmakkal. A PHP nyelv jellemzői, használatának feltételei a webfejlesztés során. Kliens és szerver oldali infrastruktúra a PHP futtatásához. A PHP típusrendszere: típusok, konstansok, változók a PHP-ban. A HTML és a PHP kapcsolata. Modulszerkezet kialakítása, a forráskód újrahasznosításának lehetőségei. Adatcsere a kliens és a szerver oldal között. Biztonsági kérdések. Munkafolyamatok és sütik kezelése.

A dinamikus weboldalak jellemzői, összehasonlításuk a statikus tartalmakkal.

Dinamikus: tartalma nem fix, változik, mert egy adatbázis adatait jeleníti meg, vagy interakcióra változik meg a tartalom.

Előnyei:

- megoldható és könnyen kezelhető az adminisztrációs felület
- a felhasználó szerkesztheti a tartalmat
- lehetőség van a látogatók és az üzemeltető közötti interakcióra

Hátrányai:

- lassabb megjelenítés
- feltörhető, védelemről gondoskodni kell
- nagyobb és többfunkciós tárhelyre van szükség (webszerver, adatbázis, futtató környezet)
- Statikus weboldal

Előnyei:

- gyors
- biztonságos, nincs futó program, amit feltörjenek
- gyorsabban fejleszthető
- könnyen és gyorsan készíthető biztonsági mentés
- kevesebb tárhely kell
- nem kell hozzá adatbázis és kódkörnyezet
- olcsóbb üzemeltetni
- nincsenek verzióproblémák és frissítési kényszer

Hátrányai:

- nincs adminisztrációs felület
- a frissítéshez mindenképpen kódolási ismeret szükséges
- a felhasználó nem tudja szerkeszteni
- a tartalom fix

PHP jellemzői

Dinamikus weboldalak előállításához fejlesztették ki. Sok beépített függvény és opcionálisan használható modulja van. A HTML-t csak formázásra használják, mert a funkcionalitás a PHP-re épül. Amikor elérünk egy ilyen oldalt, akkor a kiszolgáló feldolgozza a PHP utasításokat és

csak a kész HTML kimenetet küldi el a böngészőnek. Ehhez egy interpretert használ, amely általában egy külső modulja a webszervernek. PHP-val könnyedén megoldható a bejelentkezés, az adatbáziskezelés, fájlkezelés, kódolás, adategyeztetés, kapcsolatok létrehozása, e-mail küldése, adatfeldolgozás, dinamikus listakészítés stb.

Használatának feltételei: a szerveroldali webprogramozás eszközigénye bonyolultabb, mint a kliensoldalié.

Webszerver: az oldalak kiszolgálását végzi el, amely képes futtatni PHP állományokat. Apache.

PHP értelmező: tipikusan .php állományokhoz szokták kapcsolni a lefutását. Telepítésekor meg kell adnunk a betöltendő modulok listáját és a PHP konfigurálását is el kell végezni a php.ini fájlban.

Böngésző: a kérés elküldéséért és az oldalak megjelenítéséért felel.

Szerkesztő: szöveges szerkesztő program. Sublime Text, Notepad++, PhpStorm, VisualCode.

SFTP, SCP kliens: a kliensoldalon szerkesztett állomány webszerverre feltöltéséért felelős programok. Érdemes olyat választani, ami a kliens oldalon megnyit egy szerveroldalon szerkesztésre jelölt fájlt, és minden mentéskor automatikusan feltölti a szerverre.

Dokumentáció: a PHP függvények rendszerezett gyűjteménye és leírása.

PHP debugger: a debuggolás PHP-ban további szoftverkomponensek telepítését teszi szükségessé.

Kliens és szerver oldali infrastruktúra a PHP futtatásához

A PHP szerveroldali programozási nyelv. Nem a böngésző értelmezi, hanem egy webszerver, és ez közvetíti a böngészőnek az információkat. A szerveroldali a fent felsoroltak.

Típusrendszer

- A PHP gyengén típusos, értelmezett, általános célú szkriptnyelv.
- Kis-és nagybetű érzékeny.
- Utasításokat pontosvessző zárja le.
- Objektumorientált nyelv
- Nincs főprogram, fentről lefelé haladunk.

Típusok

Elemi típusok: logikai, egész, lebegőpontos, szöveg

Összetett típusok: tömb, objektum (osztályokon keresztül)

Speciális típusok: erőforrás, null, callback

Konstansok

Az adott név alatt tárolt érték nem változik a program futása során. define() függvény kell hozzá, vagy const kulcsszó. Nem kell $-al kezdődniük. Az állandók neve NAGYBETŰS.

Beépített állandó: __FILE__: az értelmező által beolvasott fájl nevét tartalmazza, __LINE__: az aktuális sor számát tartalmazza.

Változók

- gyengén típusosak
- nem kell deklarálni őket, a használat helyén keletkeznek
- nem kell kezdőértéket adni (de célszerű)
- élettartamuk a deklaráló blokk végéig vagy a rájuk vonatkozó unset() utasításig tart
- változó nevét $ dollárjellel kell kezdeni
- értékadás és paraméterátadás alapesetben értékszerinti, de lehetőség van referenciaszerintire is
- értékszerinti: $x = $y;
- referenciaszerinti: $x = &$y;

HTML és PHP

A PHP egy HTML-be ágyazott szerveroldali szkriptnyelv. Komplex weboldalnál keresztezzük őket. A HTML oldalon a PHP kód a <?php ?> tagek közé van zárva. A tagek között lévő kódot feldolgozza a szerver és visszaküldi a kimenetnek az eredményt. Ami meg nem, azt a böngésző dolgozza fel egyenesen.

Modulszerkezet, forráskód újrahasznosítása

Egységbezárás: az adatok és a hozzájuk tartozó metódusok egységet alkotnak, amely védett a külvilágtól. Az objektum belső világába csak ellenőrzött módon kerülhet be bármilyen adat.

Öröklődés: objektumok egymáshoz való viszonya. Egy meglévő, mezőkkel és metódusokkal rendelkező osztályból létrehozhatunk egy másikat, ami az ős minden tulajdonságát tartalmazza, vagyis örökli.

Többalakúság: az öröklés során létrejövő metódusokat meg tudjuk változtatni úgy, hogy a típusukat és paraméterlistájukat változatlanul hagyjuk, a törzsüket viszont átalakítjuk. A leszármazott osztály át tudja alakítani saját maga számára a metódust.

Adatcsere kliens és szerver között

A kliens elküldi a kérését a szerver felé, pl. űrlapon begépelt adatokat. Ezek megérkeznek a szerverhez, amin ha fut a feldolgozó program (webszever), akkor a tárolt kód alapján a kérést feldolgozza és visszaküldi a választ a kliensnek. Ehhez használhatja az adatbázist.

Biztonsági kérdések

A PHP fejlesztése során az egyszerűségre törekedvén sok olyan döntést hoztak, ami miatt hibalehetőségek kerültek be a nyelvbe. Ezért kerültek be olyan lehetőségek, amik ezeket korlátozzák. Ezek a tárhelyszolgáltatóknál jellemzően be vannak kapcsolva.

Safe Mode

Minden PHP script csak olyan fájlokat érhet el, melyeknek tulajdonosa megegyezik. Ha ez ki van kapcsolva, akkor a mappaszerkezet machinálásával elérhető a szerveren más felhasználó mappája is. Az az alap, hogy egy szerver több emberke weboldalát is kiszolgálja. PHP 5.3-as verziójától kezdve deprecated, a fejlesztők szerint OS szintjén kell szabályozni a hozzáféréseket.

SQL Injection

A kód injektálás leggyakoribb alkalmazása. A kód injektálás azt jelenti, hogy egy adatot tartalmazó memóriatartományra, vagy ami adatbevitelre van fenntartva valamilyen futtatható kódot helyezünk el. Ha nem ellenőrzi a tartalmat az input idején, akkor a kód lefut.

Cross-Site Scripting

Cél: tetszőleges JS kód futtatása az áldozat számítógépén. Kattintások is szimulálhatóak. Például kereső mezőbe script.js beszúrása. Kivédhető azzal, ha szűrjük az ilyen fajtájú keresési paramétereket. Átirányításos: rá kell venni a felhasználót, hogy rákattintson a hamisított linkre. Tárolt: az oldal biztosít lehetőséget arra, hogy a kódunkat fixen a forráskódba tudjuk helyezni és az sima megnyitásra lefut.

File Inclusion

Local file inclusion: helyi gépen férhetünk hozzá az adatokhoz

Remote file inclusion: távoli helyről is be tudunk hívni kódot az oldalba

Fájl, kód bejuttatása a rendszerbe, akár paraméterként.

Munkafolyamatok és sütik

A sütik arra valók, hogy a felhasználót azonosíthassuk velük. A felhasználó gépén tárolódnak meghatározott ideig, amit mi állíthatunk be, ha nincs letiltva. Egy cookie a setcookie(név, érték, időbélyeg, elérési út, tartomány, egész) függvénnyel hozható létre és mindig a html tag előtt kell. Egyik paraméter sem kötelező.

érték: a cookie értéke, pl. „Lajos”

időbélyeg: mennyi ideig tárolódjon, meddig legyen érvényes. time() lekérdezhetjük a pillanatnyi időt és ehhez írjuk hozzá, hogy time()+60, akkor 1 perc.

útvonal: elérhetőség helyét jeleníti meg. „/”-t megadva a webhely minden oldaláról elérhető a süti.

tartomány: megmutatja, hogy mely tartomány lesz jogosult a süti fogadására, pl. azenoldalam.hu. Ha azt szeretnénk, hogy csak a létrehozó férjen hozzá, akkor $_SERVER[’SERVER_NAME’].

egész: szám, azt mutatja, hogy a süti milyen kapcsolaton keresztül közlekedhet. 0-val nem biztonságos is engedélyezett.

A cookie-t a $_COOKIE[’cookie neve’] tömbön keresztül érhetjük el.

Süti törlése: új értéket adunk lejárt dátummal.

Session-ök: olyan süti, ami addig él, amíg a böngésző meg van nyitva. Élettartamot 0-ra kell állítani.

b) Programozási technológiák: Tervezési minták szerepe és osztályozása a szoftverfejlesztésben. A stratégia és a sablon metódus tervezési minta összehasonlítása, a megfigyelő tervezési minta szerepe és fajtái, egyéb tervezési minták. Az objektumorientált tervezési alapelvek szerepe a szoftverfejlesztésben, a nyitva-zárt alapelv bemutatása, a GOF1 és a GOF2 alapelvek rövid bemutatása, egyéb alapelvek. A jól bevált módszerek szerepe a szoftverfejlesztésben, a TDD bemutatása, egyéb jól bevált módszerek.

Tervezési minták szerepe és osztályozása

Gyakorlott programozók, miután már sokszor megoldottak egy-egy problémát, kialakították a bevett megoldást rá. Így születtek a tervezési minták, amelyek legjobb megoldások bizonyos problémákra. A minták alkalmazásával könnyen bővíthető, módosítható, rugalmas programkódot kapunk. Az OOP-ben egy osztály a valóság absztrakciója, de a mintákban olyan osztályok vannak, amiknek ehhez nem sok közük van. Emiatt bonyolultabbá válhat a kód. Ezek technikai osztályok a rugalmasság érdekében. Ezeket nem kell újra kitalálni, mert a minták bevett gyakorlatok, rendelkezésre állnak.

Osztályozás

- **Architekturális minták:** az architektúra nem változik az idők során, vagy ha igen, akkor nagyon nehezen. Pl.: MVC, ASP:NET MVC Framework, MVVM
- **Létrehozási tervezési minták:** objektumok gyártásával foglalkozik. Pl.: Singleton, Prototype, Factory Method, Abstract Factory
- **Szerkezeti tervezési minták:** hogyan használjuk az objektum-összetételt új objektum szerkezetek létrehozására. Ezt gyakran wrappingnak (becsomagolás) hívjuk. Hogyan használjuk a gyakorlatban az objektum-összetételt, hogy az igényeinknek megfelelő objektumszerkezetek létrejöhessenek futási időben. Pl.: adapter, proxy, decorator
- **Objektum-összetétel:** HAS-A kapcsolat, mely
    - **a birtoklás módja szerint:**
        - aggregáció: az összetételben szereplő objektum nem kizárólagos tulajdona az őt tartalmazó objektumnak. Gitáros – gitár. Ha meghal a gitáros, a gitárt nem temetik el vele. Stratégia.
        - kompozíció: az összetételben szereplő objektum kizárólagos tulajdona az őt tartalmazó objektumnak. A kutya-farok. Ha meghal a kutya, a farkát vele temetjük. Decorator.
    - **becsomagolás módja szerint:**
        - átlátszó: Decorator. A becsomagolt példány ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai elérhetők a becsomagolón keresztül. Megvalósításhoz IS-A és HAS-A is kell. Karácsonyfára díszek rátétele után is karácsonyfa marad.
        - átlátszatlan: Adapter. A becsomagolt példány nem ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai rejtve maradnak, kívülről nem érhetőek el. HAS-A kapcsolat kell hozzá. A becsomagolt karácsonyfa nem marad karácsonyfa, mert nincs IS-A kapcsolat és ezért nem lehet újabb díszeket rátenni.
- **Viselkedési tervezési minták:** hogyan írjunk olyan programot, amit nagyon könnyű újfajta viselkedéssel bővíteni. Pl.: State, Observer, Template Method, Strategy, Visitor.
    - **Separation of Concerns:** szétválasztás elve: ha valamit szét lehet választani, akkor azt érdemes szétválasztani. A változékony metódusokat mindig érdemes

kiemelni a tartalmazó osztályokból. Ezen kiemelt metódusok meghívására sok módszer van:

- **felelősségátadás, delegálás:** egy objektum valamely metódusa meghívja a birtokolt objektum egy metódusát, hogy az helyette oldja meg a feladatot részben vagy egészben.
- **Inversion of Control:** nem a különleges kód hívja az általános kódot, hanem az általános kód hívja a különlegest. Nem a gyermekosztály hívja az ősosztályt, hanem fordítva.
- **broadcast:** egy metódus sok más, előre nem ismert metódust hív meg egy lista alapján.

Strategy és Template Method összehasonlítása

- Stratégia: ugyanazt csináljuk, de másképp
- Template Method: ugyanúgy csináljuk, de mást

Stratégia

Kiemel egy változékony metódust egy külön, az objektumhoz objektum-összetétellel kapcsolódó osztályhierarchiába, majd felelősségátadással hívja meg a kiemelt metódust. A változékony metódust az eredeti osztályban rengetegszer módosítani kellene a stratégáitól függően, ez nem jó. Ezért az osztályhierarchia tetejére egy absztrakt osztályt teszünk és ennek a gyermekei lesznek a stratégiák. Akármennyi. Bővíthető. Az eredeti osztály objektum-összetétel segítségével kap referenciát a stratégiára. Ezen keresztül hívja a kiszervezett metódust, amelynek a megfelelő fajtája fog lefutni. Például az a színű tűzijáték fog felrobbanni, amilyen stratégiát injektáltunk be az objektum-összetételt megvalósító mezőbe. A kiemelt metódus tevékenységét az eredeti osztály delegálja a stratégiának az objektum-összetételt megvalósító mezőn keresztül. A késői kötés miatt a beinjektált konkrét stratégia fut le. Fontos, hogy a stratégiában egyetlen metódust van, mert mindig egy változékony metódust szervezek ki. Két változékony metódust akkor kell ugyanabba a stratégia gyerekosztályba tenni, ha ugyanúgy változnak, pl.: ugyanakkor változik a repülés és a hápogás. De ha ezt csináljuk, az már State állapottervezési minta. A gyereknek referenciája van a stratégiára, amit a konstruktorában injektálunk be (this.hs = hs), így csak meghatározott viselkedéssel lehet létrehozni a gyerekosztályt. Az objektum-összetétel alapján kapott referencián át hajtjuk végre a viselkedést: hs.Hapog();

Template Method

Van egy algoritmus vázam és egy algoritmus családom. Például a bableves receptjeinek vannak közös részei, de vannak különbözőek. Absztrakt ős lesz. Egy publikus metódusa van (hogy kívülről ne lehessen hívni megváltoztatott sorrenddel, ugyanezért a lépések nem publikusak), amely meghívja a többi függvényt adott sorrendben. Pl.: a Recept() függvény a minta megvalósítása, ő hívja a többi függvényt sorrendben.

- Kötelező és közös: az abstract ősben konkrét metódusok, private, kidolgozott törzs.
- Kötelező, nem közös: abstract ősben abstractot, protected abstract, nincs törzse
- Nem kötelező: hook metódusok, protected virtual, van törzse, de üres

Inversion of Control: ős hívja a gyerek metódusát, ha az ős abstract-ja ki van dolgozva a gyerekben. Amikor a publikus metódusunkban meghívunk egy az ősben abstract metódust, vagy hook-ot, akkor hajtunk végre IoC-t. Az ős hívja azt, ami a gyerekben van kifejtve.

Observer szerepe és fajtái

Lehetővé teszi az objektumok közötti kommunikációt anélkül, hogy bármit is tudna róluk. Ehhez a kommunikációhoz interfészt biztosít. Egy esemény által kiváltott változékony metódust emel ki egy 1-sok kapcsolat sok oldalára. Ez broadcast-tal hívja meg a kiemelt metódusokat (lista alapján előre nem ismert metódusokat hív meg).

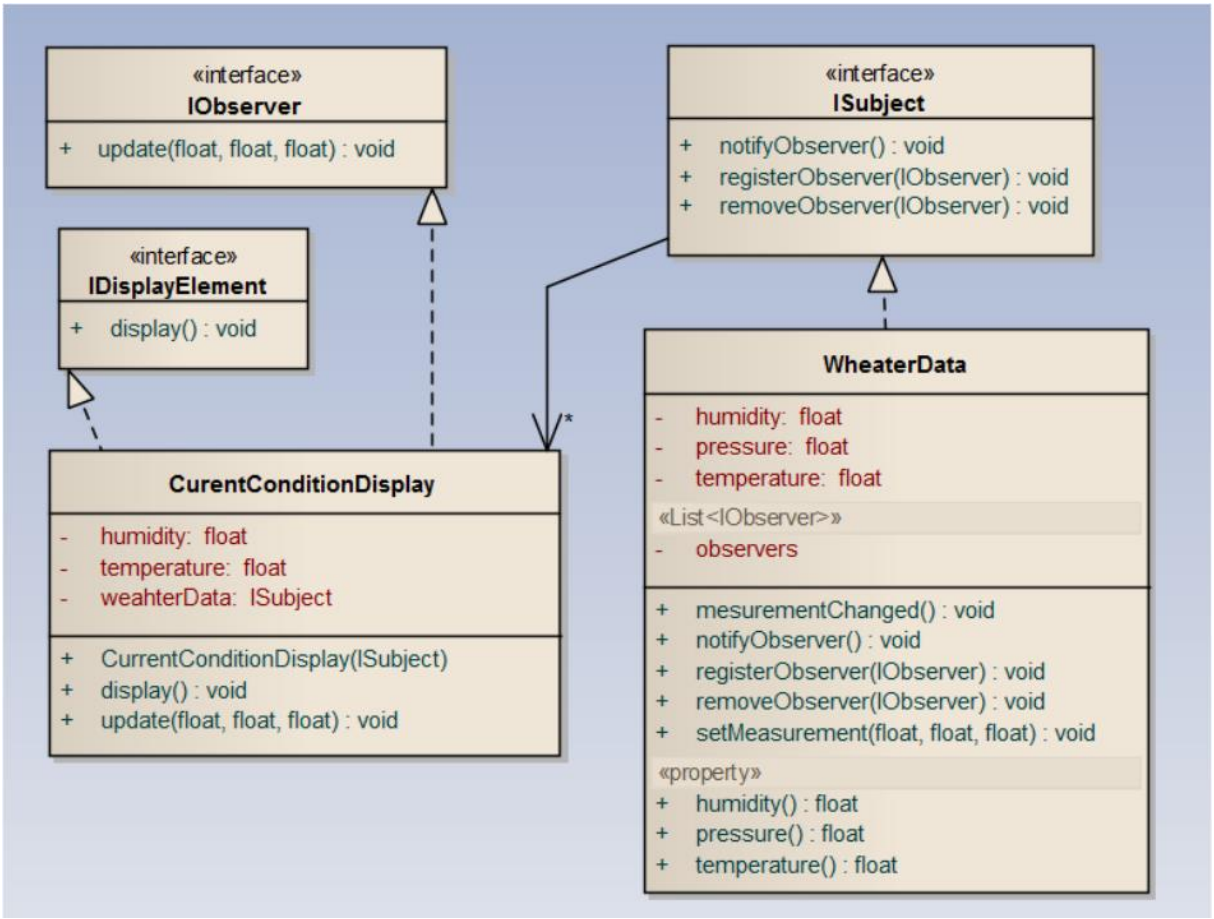
Hollywood Principle: ne hívj, majd mi hívunk.

- Megfigyelt: tárolja a felregisztrált megfigyelőket, lehetőséget ad a felregisztrált megfigyelők értesítésére
- Megfigyelő: azok az objektumok, amelyek értesülni szeretnének az alanyban bekövetkezett változásokról. Erre az update() metódus szolgál.

Observer fajtái:

- Pull: a megfigyelő lehúzza a változásokat az alanyról
- Push: az alany odanyomja a változásokat a megfigyelőnek

A különbség az update() metódus paramétere. Ha az alany önmagát adja át paraméterként a megfigyelőnek, akkor ezen referencián keresztül a megfigyelő lekérdezi a változásokat. Ha az alany a megváltozott mezőket adja át paraméterként, akkor a másik változat van.



Egyéb tervezési minták

- Létrehozási tervezési minták: singleton, prototype, factory method
- Szerkezeti tervezési minták: adapter, decorator, proxy

- Viselkedési tervezési minták: state, observer, template method, strategy

OOP alapelvek szerepe a szoftverfejlesztésben

Egységbezárás: az adatstruktúrát és az adatokat kezelő metódusokat egységként kezeljük és elzárjuk őket a külvilág elől. Az objektum belső állapotát védjük. Az így kapott egység az objektumosztály.

Öröklődés: a meglévő objektumokból levezetett újabb objektumok öröklik a definiálásukhoz használt alap objektumok egyes adatstruktúráit és függvényeit, de újabb tulajdonságokat is definiálhatnak vagy régieket újraértelmezhetnek. Öröklődés során a private mezők és metódusok kivételével átvesszük a dolgokat. Az öröklődés miatt csökken a kód redundanciája és olvashatóbb, elszeparálhatóbb lesz a kód. Az öröklődés implementációs függőséget okoz az ős és a gyermek között. Helyette, ahol lehet objektum-összetételt használjunk.

Sokalakúság: egy metódus azonosítója közös lehet egy adott objektumosztály-hierarchián belül, de a hierarchia minden egyes objektumosztályában a metódus implementációja az adott objektumosztályra nézve specifikus lehet. A sokalakúság miatt egy objektum több típusként, azaz több alakban is használható.

Nyitva-zárt alapelv OCP

A forráskód legyen nyitva a bővítésre, de zárt a módosításra. Bármikor rakhatok be új elemet (osztály, metódus), de meglévőt lehetőleg ne írjak felül. Abstract elemekre és hook metódusokra nem vonatkozik a megkötés. Az abstractnak nincs törzse az ősben, a hook-é pedig üres.

GOF1

Programozz felületre megvalósítás helyett.

Az osztálynak két jellemzője van: megvalósítás (forráskód) és felület (publikus rész).

Az objektumnak három jellemzője van: felület (típus), viselkedés (forráskód dinamikus vetülete, a futó kód), belső állapot (mezők pillanatnyi értéke).

- abstract osztály: felülete és részleges megvalósítása van
- interface: csak felülete van, nincs megvalósítása

Megvalósításra programozunk, ha: tudjuk, hogyan van implementálva az osztály és a többi megírásához ezt felhasználom. Ha az osztály nem fedi le teljesen a felelősségi körét, akkor azokat külső osztályok látják el, amiknek tudniuk kell, hogyan van az eredeti implementálva. Ha az eredeti osztály megváltozik, a vele implementációs függőségben lévők is megfognak.

Felületre programozunk, ha: nem vesszük figyelembe, hogy hogyan van implementálva az osztály, csak a felületét ismerem. Minden metódusnak van elő és utófeltétele. Ez az osztály szerződése. Például: bináris keresés csak rendezett sorozatra működik jól. Előfeltétele a rendezett sorozat és a belső állapot. Utófeltétele a visszatérési érték és az új belső állapot.

GOF1 ajánlása, hogy típusként a még megfelelő legősibbet kell használni, hogy minél absztraktabb legyen a kód.

- Error: megállítja a programot.

- Kivétel: try-catch-csel elkaphatjuk.

GOF2

Öröklődés helyett használjunk objektum-összetételt, ahol lehet. De nem szabad túlzásba vinni, mert az ilyen kód nehezebben átlátható, illetve öröklődés nélkül nincs többalakúság, amire viszont sokszor szükség van.Az öröklődés erős csatoltságot okoz, ha az ős kódja változik, a gyereké is fog valószínűleg. Objektum-összetétel lényege, hogy referenciát hoz létre egy másik objektumra. Futási időben is létrejöhet és változhat. Az öröklődés fordítási időben dől el és nem változhat.

- **a birtoklás módja szerint:**
  - aggregáció: az összetételben szereplő objektum nem kizárólagos tulajdona az őt tartalmazó objektumnak. Gitáros – gitár. Ha meghal a gitáros, a gitárt nem temetik el vele. Stratégia.
  - kompozíció: az összetételben szereplő objektum kizárólagos tulajdona az őt tartalmazó objektumnak. A kutya-farok. Ha meghal a kutya, a farkát vele temetjük. Decorator.
- **becsomagolás módja szerint:**
  - átlátszó: Decorator. A becsomagolt példány ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai elérhetők a becsomagolón keresztül. Megvalósításhoz IS-A és HAS-A is kell. Karácsonyfára díszek rátétele után is karácsonyfa marad.
  - átlátszatlan: Adapter. A becsomagolt példány nem ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai rejtve maradnak, kívülről nem érhetőek el. HAS-A kapcsolat kell hozzá. A becsomagolt karácsonyfa nem marad karácsonyfa, mert nincs IS-A kapcsolat és ezért nem lehet újabb díszeket rátenni.

Csatoltság erőssége: öröklődés, kompozíció, aggregáció.

Egyéb alapelvek

- **SRP: Egy felelősség egy osztály elve:** egy osztály fedje le teljesen a saját felelősségi körét. Ha ez nem történik meg, akkor implementációra kell programozni, hogy egy másik osztály megvalósítsa azokat a szolgáltatásokat, amik kimaradtak az osztályból. Legyen külön Macska és Kutya osztályunk MacsKutya helyett, mert így ha a Macska viselkedése változik, csak azt kell módosítani. Ha több osztálynak kell ugyanazt a felelősségi kört ellátnia (naplózás), akkor jöhet az aspektus-orientált programozás, amit kiemelhetünk az aspektusba.
- **LSP: Liskov-féle behelyettesítési elv:** a program viselkedése nem változhat meg attól, hogy az ős osztály egy példánya helyett a jövőben egy gyerekosztály példányát használom fel. Négyzet – téglalap.
- **Függőség megfordításának elve (IoC):** absztrakcióktól függj, ne konkrét osztályoktól. A magas szintű komponensek ne függjenek alacsony szintű implementációs részeket kidolgozó osztályoktól, hanem fordítva. Absztrakttól függjenek a konkrétak.
  - **Felelősség injektálás típusai**
    - konstruktor: konstruktoron keresztül kapja meg az osztály a szükséges referenciákat. Leggyakoribb megvalósítás.

- setter: setteren keresztül kapja meg az osztály a szükséges referenciákat. Akkor használjuk, ha opcionális működéshez kell objektum-összetétel.
- interfész: ha a példányt magasszintű komponens is elkészítheti, akkor elegendő megadni a példány interfészét, amit általában maga a magasszintű komponens valósít meg
- elnevezési konvenció, konfigurációs állomány, annotáció: keretrendszerekre jellemző.

Best practice szerepe a szoftverfejlesztésben

TDD

Agilis módszer. Először a unit-tesztet írjuk meg, majd a hozzá tartozó metódusnak azt a részét, ami teljesíti a tesztet. Ezt folytatjuk, amíg kész nincs a metódus. Piros-Zöld-Piros

Hatásai:

- lesznek unit-testek
- tesztelőként gondolkodunk először, nem programozóként. Programozóként szép és hatékony kódot akarsz írni, de az nehezen tesztelhető. Így könnyű lesz.
- nyugodtan hozzányúlhatunk a kódhoz, mert látjuk, ha egy változtatás után nem teljesül a teszt.
- nagy lesz a kódlefedettség
- kevesebb idő a hibakeresés
- a unit-tesztek felfoghatók programozóknak szóló dokumentációként is

Extrém programozás

- páros programozás: az egyik programozó írja, a másik figyeli a kódot. Ha a figyelő hibát lát, vagy nem érti, szól. Folyamatosan megbeszélik, hogy hogyan érdemes megoldani a problémát.
- teszt vezérelt fejlesztés: a metódus elkészítése előtt megírjuk a hozzá tartozó unit-tesztet. Folyamatosan mindig előre írjuk meg az adott bővítéshez szükséges tesztet, és aztán úgy bővítjük a metódust, hogy a tesztet átmenjen.
- forráskód átnézés (review): elkészült nagyobb modulokat egy vezető fejlesztő átnézi. A fejlesztők elmagyarázzák mit miért csináltak. A vezető fejlesztő elmondja hogyan lehet jobban.
- folyamatos integráció: a nap vagy hét végén a verziókezelő rendszerbe bekerült kódokat integrációs teszteljük, hogy kiderüljön, képesek-e együttműködni. Így korán kiszűrhető a programozók közötti félreértés.
- kódszépítés (refactoring): a már letesztelt, működő kódban lehet szépíteni a lassú, rugalmatlan vagy csúnya részeket. Előfeltétele, hogy legyen sok unit-teszt. A funkcionalitást nem szabad megváltoztatni. Minden unit-tesztet le kell futtatni a végén, nem csak a módosított részhez tartozókat.

14. tétel

a) Programozási technológiák: Az osztály jellemzői: felület és megvalósítás, az objektum jellemzői: felület, belső állapot és viselkedés. Az OOP alapelvek megfogalmazása ezekkel a fogalmakkal. A GOF1 és a GOF2 tervezési alapelvek bemutatása. A stratégia részletes, illetve a viselkedési tervezési minták általános bemutatása. Az egyke részletes, illetve a létrehozási tervezési minták általános bemutatása. A díszítő részletes, illetve a szerkezeti tervezési minták általános bemutatása.

Az interfésznek csak felülete van, az absztrakt osztálynak felülete és részleges viselkedése.

Az osztály jellemzői: felület és implementáció

Az osztály

- a valóság egy megfogható vagy megfoghatatlan darabkájának absztrakciója. A darabka mérete adja meg az osztály durvaságát.
- lehet teljesen technikai is, a valósághoz nem kapcsolható, például tervezési minták esetében. összetett, inhomogén adattípus. Vannak mezői, amelyek bármilyen típusúak lehetnek, vannak metódusai és különleges metódusai, property-jei.
- az adatok és a rajtuk elvégzett műveleteket foglalja egységbe.
- lehetnek példányai.
- felülete: az osztály publikus részei adják, jellemzően metódusok és property-k. Leírja, hogy milyen szolgáltatásokat nyújt az osztály, pl.: getNév().
- implementációja: az összes metódus implementációja határozza meg. Pl.: getNév() visszaadja a nevet, de mást is megadhatnánk.

Az objektum jellemzői: felület, belső állapot és viselkedés

Az objektum:

- felülete: az osztály egy példánya, felületük megegyezik, vagyis az objektum adott osztály típusú.
- viselkedése: az összes metódus implementációja határozza meg. Megegyezik annak az osztálynak a viselkedésével, aminek a példány az objektuma. A viselkedés futási időben változhat.
- belső állapota: mezők pillanatnyi értéke határozza meg. Az osztály metódusai megváltoztathatják a mezők értékeit, ezért a metódusokat tekinthetjük állapot átmeneti operátoroknak is. A kezdő belső állapotot a konstruktor hívása határozza meg.

OOP alapelvek ezekkel a fogalmakkal

Egységbezárás

Alapfogalom: az adattagokat és a rajtuk műveleteket végrehajtó metódusokat egységbe zárjuk.

Ami ide kell: az objektum belső állapotát meg kell védeni, azt csak a saját metódusai változtathatják meg.

Öröklődés

Alapfogalom: a gyermekosztály az ősosztály minden nem privát mezőjét és metódusát örököli.

Ami ide kell: a gyermekosztály örököli az ősosztály felületét és viselkedését. Implementációs függőség jön létre.

Többalakúság

Az öröklődés következménye. Mivel a gyermekosztály örököli az ős felületét, ezért a gyermekosztály példányai megkapják az ős típusát is. Így egy objektum több típusként, azaz több alakban is használható. Egy osztály példányai az öröklési hierarchián felfelé haladva rendelkeznek az összes típussal. Ezért minden objektum végül is object típusú.

GOF1

Programozz felületre megvalósítás helyett.

Az osztálynak két jellemzője van: megvalósítás (forráskód) és felület (publikus rész).

Az objektumnak három jellemzője van: felület (típus), viselkedés (forráskód dinamikus vetülete, a futó kód), belső állapot (mezők pillanatnyi értéke).

- abstract osztály: felülete és részleges megvalósítása van
- interface: csak felülete van, nincs megvalósítása

Megvalósításra programozunk, ha: tudjuk, hogyan van implementálva az osztály és a többi megírásához ezt felhasználom. Ha az osztály nem fedi le teljesen a felelősségi körét, akkor azokat külső osztályok látják el, amiknek tudniuk kell, hogyan van az eredeti implementálva. Ha az eredeti osztály megváltozik, a vele implementációs függőségben lévők is megfognak.

Felületre programozunk, ha: nem vesszük figyelembe, hogy hogyan van implementálva az osztály, csak a felületét ismerem. Minden metódusnak van elő és utófeltétele. Ez az osztály szerződése. Például: bináris keresés csak rendezett sorozatra működik jól. Előfeltétele a rendezett sorozat és a belső állapot. Utófeltétele a visszatérési érték és az új belső állapot.

GOF1 ajánlása, hogy típusként a még megfelelő legősibbet kell használni, hogy minél absztraktabb legyen a kód.

- Error: megállítja a programot.
- Kivétel: try-catch-csel elkaphatjuk.

GOF2

Öröklődés helyett használjunk objektum-összetételt, ahol lehet. Az öröklődés erős csatoltságot okoz, ha az ős kódja változik, a gyereké is fog valószínűleg. Objektum-összetétel lényege, hogy referenciát hoz létre egy másik objektumra. Futási időben is létrejöhet és változhat. Az öröklődés fordítási időben dől el és nem változhat.

- **a birtoklás módja szerint:**
  - aggregáció: az összetételben szereplő objektum nem kizárólagos tulajdona az őt tartalmazó objektumnak. Gitáros – gitár. Ha meghal a gitáros, a gitárt nem temetik el vele. Stratégia.

        - kompozíció: az összetételben szereplő objektum kizárólagos tulajdona az őt tartalmazó objektumnak. A kutya-farok. Ha meghal a kutya, a farkát vele temetjük. Decorator.
    - **becsomagolás módja szerint:**
        - átlátszó: Decorator. A becsomagolt példány ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai elérhetők a becsomagolón keresztül. Megvalósításhoz IS-A és HAS-A is kell. Karácsonyfára díszek rátétele után is karácsonyfa marad.
        - átlátszatlan: Adapter. A becsomagolt példány nem ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai rejtve maradnak, kívülről nem érhetőek el. HAS-A kapcsolat kell hozzá. A becsomagolt karácsonyfa nem marad karácsonyfa, mert nincs IS-A kapcsolat és ezért nem lehet újabb díszeket rátenni.

Csatoltság erőssége: öröklődés, kompozíció, aggregáció.

Viselkedési tervezési minták általános bemutatása

Középpontjukban az algoritmusok és az objektumokhoz rendelt felelősségi körök állnak. Az osztályok és objektumok rendszerén túl a közöttük folyó kommunikációt és a felelősségi köröket is leírják. Öröklődést helyett objektum-összetételt alkalmaznak. Hogyan írjunk olyan programot, amit nagyon könnyű újfajta viselkedéssel bővíteni. Pl.: State, Observer, Template Method, Strategy, Visitor.

- **Separation of Concerns:** szétválasztás elve: ha valamit szét lehet választani, akkor azt érdemes szétválasztani. A változékony metódusokat mindig érdemes kiemelni a tartalmazó osztályokból. Ezen kiemelt metódusok meghívására sok módszer van:
    - **felelősségátadás, delegálás:** egy objektum valamely metódusa meghívja a birtokolt objektum egy metódusát, hogy az helyette oldja meg a feladatot részben vagy egészben.
    - **Inversion of Control:** nem a különleges kód hívja az általános kódot, hanem az általános kód hívja a különlegest. Nem a gyermekosztály hívja az ősosztályt, hanem fordítva.
    - **broadcast:** egy metódus sok más, előre nem ismert metódust hív meg egy lista alapján.

Stratégia részletes bemutatása

Kiemel egy változékony metódust egy külön, az objektumhoz objektum-összetétellel kapcsolódó osztályhierarchiába, majd felelősségátadással hívja meg a kiemelt metódust. A változékony metódust az eredeti osztályban rengetegszer módosítani kellene a stratégáitól függően, ez nem jó. Ezért az osztályhierarchia tetejére egy absztrakt osztályt teszünk és ennek a gyermekei lesznek a stratégiák. Akármennyi. Bővíthető. Az eredeti osztály objektum-összetétel segítségével kap referenciát a stratégiára. Ezen keresztül hívja a kiszervezett metódust, amelynek a megfelelő fajtája fog lefutni. Például az a színű tűzijáték fog felrobbanni, amilyen stratégiát injektáltunk be az objektum-összetételt megvalósító mezőbe. A kiemelt metódus tevékenységét az eredeti osztály delegálja a stratégiának az objektum-összetételt megvalósító mezőn keresztül. A késői kötés miatt a beinjektált konkrét stratégia fut le. Fontos,

hogy a stratégiában egyetlen metódust van, mert mindig egy változékony metódust szervezek ki. Két változékony metódust akkor kell ugyanabba a stratégia gyerekosztályba tenni, ha ugyanúgy változnak, pl.: ugyanakkor változik a repülés és a hápogás. De ha ezt csináljuk, az már State állapottervezési minta. A gyereknek referenciája van a stratégiára, amit a konstruktorában injektálunk be (this.hs = hs), így csak meghatározott viselkedéssel lehet létrehozni a gyerekosztályt. Az objektum-összetétel alapján kapott referencián át hajtjuk végre a viselkedést: hs.Hapog();

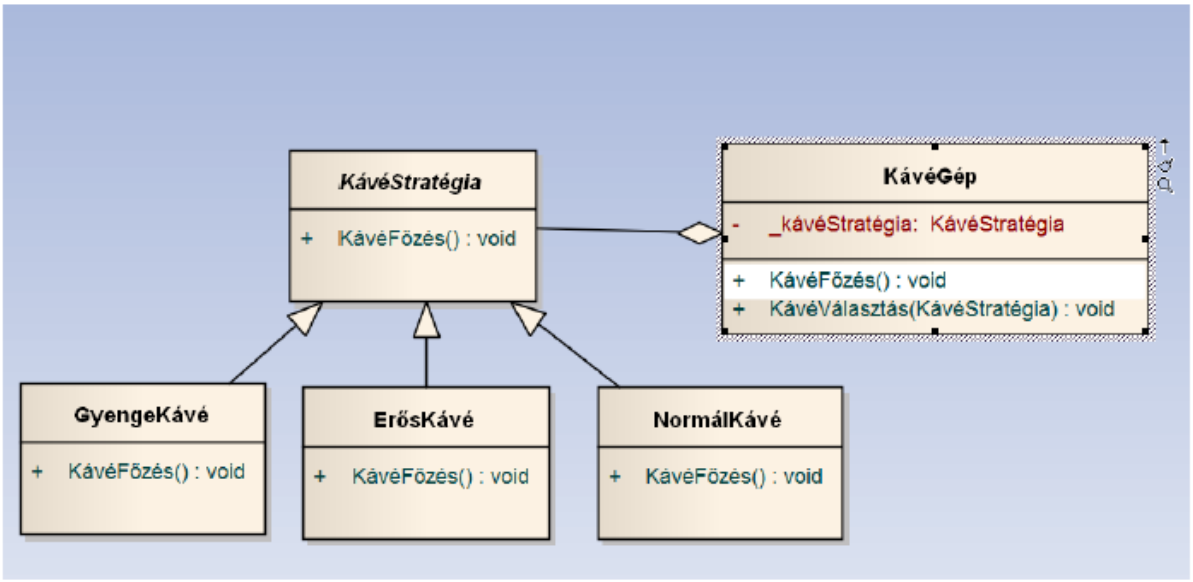
Alkalmazási terület:

1. Olyan osztályaink vannak, amelyek csak viselkedésükben különböznek. A stratégia minta segítségével az osztályunkat a sok viselkedés egyikével ruházhatjuk fel.

2. Egy algoritmus különböző változataira van szükségünk. A stratégia mintát akkor használhatjuk, ha ezeket az algoritmusokat osztályhierarchiában implementáltuk

3. Az algoritmus olyan adatot használ, amiről a kliensnek nem kell tudnia. A stratégia minta segít elkerülni a komplex, algoritmus specifikus adatok feltárását.

4. Egy osztály sokféle viselkedést definiál, amelyek a műveleteiben többször megjelennek feltételként.



24. ábra A Stratégia tervezési minta

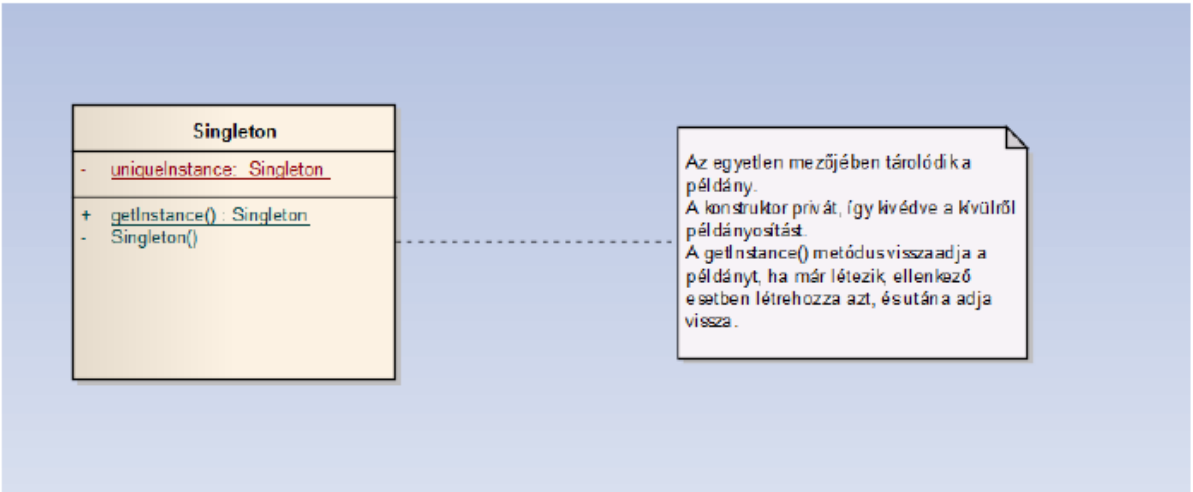
Létrehozási tervezési minták általános bemutatása

Középpontjukban a különféle osztályok példányainak hatékonyabb létrehozása áll, pl.: a new kulcsszó kikerülésével. Ha mindenhová new-t írunk, később nehéz lesz lecserélni más hívásra, stb. Ehelyett a példányok gyártását ezekkel a mintákkal oldjuk meg. Ha változnak a követelmények, csak egy helyen kell módosítani a létrehozás módját.

Egyke részletes bemutatása

Célunk, hogy egy osztályból csak egyetlen példányt lehessen létrehozni. Osztályból konstruktorral készítünk példányt. Ha van ebből publikus, akkor akárhány példány készíthető. Ellenben, ha nincs konstruktor, akkor nincs példány, amin keresztül hívhatnánk a

metódusokat. A megoldás az osztályszintű metódus, amit akkor is lehet hívni, ha nincs példány. Az egykének van egy osztályszintű metódusa, ami mindenkinek ugyanazt a példányt adja vissza. Ezt is létre kell hozni, amit privát konstruktorral megteszünk, amit ez az osztályszintű metódus hívhat például, hiszen rálát, mert az osztály része.



17. ábra A Egyke tervezési minta

Szerkezeti tervezési minták általános bemutatása

Hogyan használjuk az objektum-összetételt új objektum szerkezetek létrehozására. Ezt gyakran wrappingnak (becsomagolás) hívjuk. Hogyan használjuk a gyakorlatban az objektum-összetételt, hogy az igényeinknek megfelelő objektumszerkezetek létrejöhessenek futási időben. Pl.: adapter, proxy, decorator.

- **Objektum-összetétel:** HAS-A kapcsolat, mely
  - **a birtoklás módja szerint:**
    - aggregáció: az összetételben szereplő objektum nem kizárólagos tulajdona az őt tartalmazó objektumnak. Gitáros – gitár. Ha meghal a gitáros, a gitárt nem temetik el vele. Stratégia.
    - kompozíció: az összetételben szereplő objektum kizárólagos tulajdona az őt tartalmazó objektumnak. A kutya-farok. Ha meghal a kutya, a farkát vele temetjük. Decorator.
  - **becsomagolás módja szerint:**
    - átlátszó: Decorator. A becsomagolt példány ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai elérhetők a becsomagolón keresztül. Megvalósításhoz IS-A és HAS-A is kell. Karácsonyfára díszek rátétele után is karácsonyfa marad.
    - átlátszatlan: Adapter. A becsomagolt példány nem ugyanolyan felületű, mint a becsomagoló, ezért a becsomagolt objektum szolgáltatásai rejtve maradnak, kívülről nem érhetőek el. HAS-A kapcsolat kell hozzá. A becsomagolt karácsonyfa nem marad karácsonyfa, mert nincs IS-A kapcsolat és ezért nem lehet újabb díszeket rátenni.

Díszítő részletes bemutatása

Az átlátszó csomagolás klasszikus példája, pl.: karácsonyfa. Attól, hogy egy gömböt felteszek rá, még karácsonyfa marad. Az objektum-összetételből szereplő mindkét osztály azonos ősből

származik. Vagyis a gömb és a karácsonyfa őse ugyanaz, ami hétköznapi logikával hülyeség, de ez a tervezési minták sajátossága.

Egy absztrakt ősből indulunk ki, aminek vannak alap és díszítő gyerekosztályai. A díszítőket is általában egy absztrakt díszítő alá szervezzük.

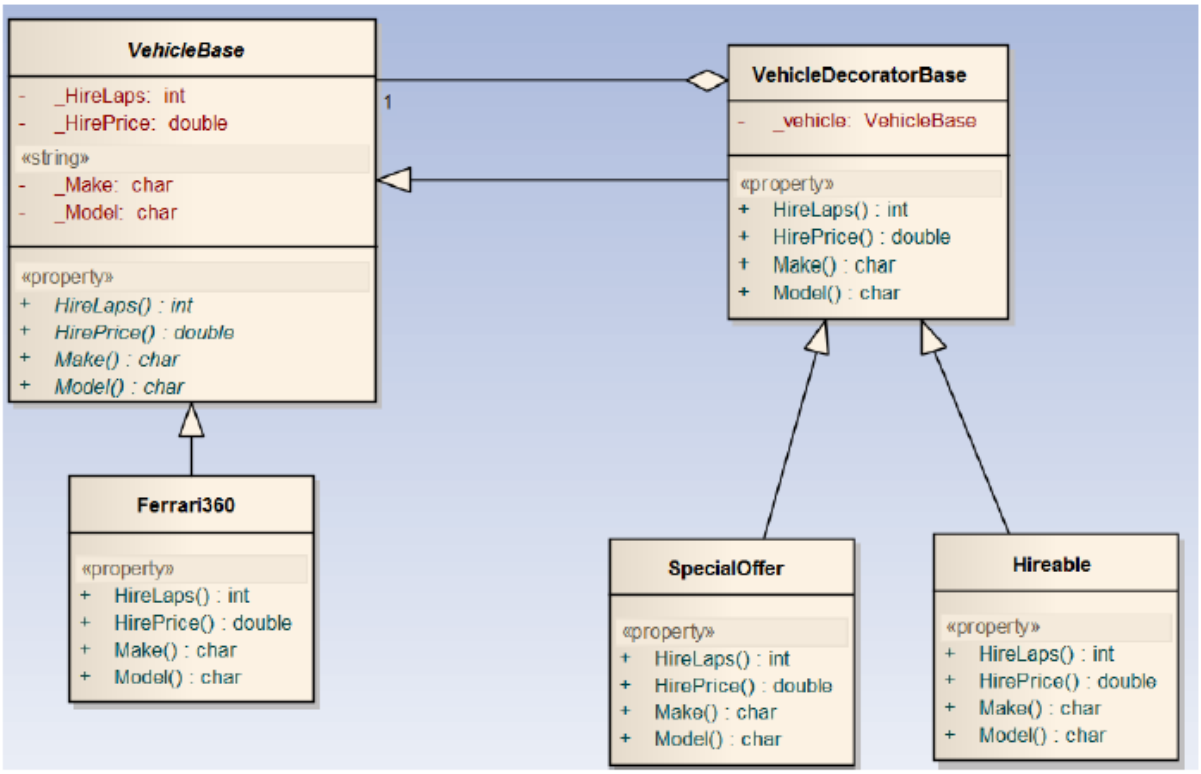
A díszítés során az ős minden metódusát implementálni kell úgy, hogy a becsomagolt példány metódusát meghívjuk, illetve ahol szükséges, ott hozzáadjuk a plusz funkcionalitást.

Díszítés fajtái:

- amikor a meglévő metódusok felelősségkörét bővítjük. Karácsonyfás példa.
- amikor új metódusokat is hozzáadunk a meglévőkhöz. Java stream kezelése, kölcsönözhető jármű az ábrán.

Mindkét esetben a példányosítás: ŐsOsztály példány = new DíszítőN(…new Díszítő1(new AlapOsztály())…); A csomagolás átlátszó, ezért akárhányszor elvégezhető, ettől dinamikus lesz.

Az UML ábrán a díszítő osztályból az ősosztályra mutat vissza egy aggregáció. Ez olyan, mint az Alkalmazott-Főnök reláció, ahol az Alkalmazott tábla önmagával áll egy-több kapcsolatban, ahol az idegen kulcs a főnök alkalmazott_ID értékét tartalmazza.



20. ábra A Díszítő tervezési minta

b) Adatbázisrendszerek II: Kurzorok: implicit és explicit kurzorok kezelése, kurzorattribútumok használatának lehetősége, kurzorváltozók. Tranzakciókezelés, triggerek a PL/SQL nyelvben. Összetett adatszerkezetek (kollekciók) típusai, jellemzői, használatának lehetőségei és előnyei, hátrányai.

Kurzorok

Egy SQL utasítás feldolgozásához az Oracle a memóriában egy környezeti területnek nevezett részt használ, amely információkat tartalmaz az utasítás által feldolgozott sorokról, lekérdezés esetén tartalmazza a visszaadott sorokat (aktív halmaz) és tartalmaz egy mutatót az utasítás belső reprezentációjára.

A kurzorral megnevezhetjük a környezeti területet, hozzáférhetünk az információkhoz és ha az aktív halmaz több sort tartalmaz, azokat egyenként elérhetjük. Pl telefonkönyvben keresés az ujjunkkal. Az ujjunk a kurzor, ami adott rekordra mutat.

Implicit kurzor: a PL/SQL automatikusan felépít egy ilyet minden DML-utasításhoz, beleértve az olyan lekérdezéseket is, amelyek pontosan egy sort adnak vissza.

Explicit kurzor: a több sort (akárhány sort) visszaadó lekérdezések eredményének kezeléséhez.

Implicit kurzorok

Attribútumai az INSERT, DELETE, UPDATE és SELECT INTO utasítások végrehajtásáról adnak információt, mely mindig a legutoljára végrehajtott utasításra vonatkozik.

- %FOUND: Az utasítás végrehajtása alatt értéke NULL. Utána értéke TRUE, ha az INSERT, DELETE, UPDATE utasítás egy vagy több sort érintett, illetve a SELECT INTO legalább egy sort visszaadott. Különben értéke FALSE.
- %ISOPEN: Az Oracle automatikusan lezárja az implicit kurzort az utasítás végrehajtása után, ezért értéke mindig FALSE.
- %NOTFOUND: Értéke TRUE, ha az INSERT, DELETE, UPDATE egyetlen sorra sem volt hatással, illetve ha SELECT INTO nem adott vissza sort, egyébként FALSE.
- %ROWCOUNT: Értéke az INSERT, DELETE, UPDATE utasítások által kezelt sorok darabszáma. A SELECT INTO utasítás esetén értéke 0, ha nincs visszaadott sor és 1, ha van. Ha a SELECT INTO utasítás egynél több sort ad vissza, akkor a TOO_MANY_ROWS kivétel váltódik ki. Ekkor tehát a %ROWCOUNT nem a visszaadott sorok tényleges számát tartalmazza.

Explicit kurzorok

A kezelésének 4 lépése van: deklarálás, megnyitás, sorok betöltése, lezárás

Megnyitáskor lefut a lekérdezés, meghatározódik az aktív halmaz és az ahhoz rendelt kurzormutató az első rekordra áll rá. `OPEN kurzor_név [(aktuális_paraméter_lista)];`

Megnyitott kurzor újbóli megnyitása a CURSOR_ALREADY_OPEN kivételt váltja ki. A kurzor működése az elején olyan, mint egy egyszerű lekérdezésé, de aztán az eredményhalmaz bekerül a memóriába, ahol hozzáférünk az elemekhez, akár módosítani is lehet őket a kurzor típusától függően.

Az aktív halmaz sorainak feldolgozását a FETCH utasítás teszi lehetővé: adott kurzorhoz vagy kurzorváltozóhoz tartozó kurzormutató által címzett sort betölti, a kurzormutatót a következő sorra állítja. Nem létező sor betöltése nem vált ki kivételt.

```
FETCH{kurzor_név|kurzorváltozó_név}
{INTO{rekord_név|változó_név[,változó_név]…} LIMIT sorok};
```

A kurzor lezárása érvényteleníti a kurzor és az aktív halmaz közötti kapcsolatot és megszünteti a kurzormutatót: `CLOSE {kurzornév|kurzorváltozó_név};`

```
DECLARE
 CURSOR wf_holiday_cursor IS
 SELECT country_name, national_holiday_date
 FROM wf_countries WHERE region_id IN(30,34,35);
 v_country_name   wf_countries.country_name%TYPE;
 v_holiday        wf_countries.national_holiday_date%TYPE;
BEGIN
 OPEN wf_holiday_cursor;
  LOOP
    FETCH wf_holiday_cursor INTO v_country_name, v_holiday;
    EXIT WHEN wf_holiday_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_country_name|| ' '||v_holiday);
END LOOP;
CLOSE wf_holiday_cursor;
END;
```

Kurzorattribútumok használata

A DML-utasítás végrehajtásáról szolgáltatnak információkat. Csak procedurális utasításokban használhatóak, SQL-ben nem.

- %FOUND: az első sor betöltése előtt értéke NULL. Sikeres sorbetöltés esetén TRUE, egyébként FALSE.
- %NOTFOUND: a %FOUND ellentétje
- %ISOPEN: értéke TRUE, ha a kurzor meg van nyitva, egyébként FALSE
- %ROWCOUNT: értéke az első sor betöltése előtt 0. Ezután mindig 1-gyel nő, megadja a darabszámot.

Ha az explicit kurzor vagy a kurzorváltozó nincs megnyitva, akkor az %ISOPEN kivételével a másik három az INVALID_CURSOR kivételt váltja ki.

Kurzorváltozók

Explicit kurzorhoz hasonlóan ez is aktív halmaz valamely sorának feldolgozására alkalmas. Dinamikus, futási időben bármely típuskompatibilis kérdés hozzákapcsolható. Lényegében egy referencia típusú változó, amely mindig a hivatkozott sor címét tartalmazza. Ugyanúgy használjuk, mint az explicit kurzort.

Deklarálás: REF CURSOR saját típus, majd ezzel deklarálunk egy változót.

```
TYPE név IS REF CURSOR [RETURN
{{ab_tábla_név|kurzor_név|kurzorváltozó_név}%ROWTYPE|
rekord_név%TYPE|
rekordtípus név|
kurzorreferenciatípus_név}];
```

Ha van RETURN akkor erős, ha nincs akkor gyenge kurzorreferenciáról beszélünk. Erősnél a fordító ellenőrzi a kompatibilitást, gyengéhez bármely kérdés kapcsolható.

Tranzakciókezelés

A tranzakció az adatbázis-műveletek végrehajtási egysége, amely DML-béli utasításokból áll és ACID tulajdonságokkal rendelkezik:

- Atomosság (atomicity): a tranzakció mindent vagy semmit jellegű végrehajtása
- Konzisztencia (consistency): az a feltétel, hogy a tranzakció megőrizze az adatbázis konzisztenciáját, azaz a tranzakció végrehajtása után is teljesüljenek az adatbázisban előírt konzisztenciamegszorítások (integritási megszorítások), az az az adatelemekre és a közöttük lévő kapcsolatokra vonatkozó elvárások ????????? nem lehetett volna magyarul????????
- Elkülönítés (isolation): minden tranzakciónak látszólag úgy kell lefutnia, mintha ezalatt az idő alatt semmilyen más tranzakciót nem hajtanánk végre
- Tartósság (durability): ha egyszer egy tranzakció befejeződött, akkor már soha többé nem veszhet el a tranzakciónak az adatbázison kifejtett hatása

Implicit COMMIT hajtódik végre, ha:

- egy DDL vagy DCL utasítást adunk ki.
- ha a felhasználó kilép az adatbázisból

Implicit ROLLBACK hajtádóik végre, ha a módosításokat végző alkalmazás rendellenesen fejeződik be.

- COMMIT: véglegesíti a tranzakció során elvégzett adatmódosításokat. Ezek a többi felhasználó számára is látható válnak.
- ROLLBACK: visszagörgeti (meg nem történtté teszi) a tranzakció során elvégzett adatmódosításokat. Ha volt COMMIT, nincs rá lehetőség.

Mindkettő lezárja a tranzakciót.

Triggerek PL/SQL nyelvben

Olyan tevékenység, amely automatikusan végbemegy, ha egy tábla vagy nézet módosul, vagy egyéb felhasználói vagy rendszeresemények következnek be. A trigger adatbázis-objektum.

- Megírható PL/SQL, vagy akár objektum-orientált nyelven (Java, C).
- A felhasználó számára látható módon működik.
- A tárolt eljárások egy speciális fajtája, amelynek lefutását nem lehet elkerülni.
- Táblához vagy adatbázis-eseményekhez kötődnek.
- Mindig abban az adattáblában érvényes és hajtódik végre, amelyben definiáltuk, és az INSERT, UPDATE, DELETE után futnak le.
- Egy táblának lehet több triggere, és összetettebb eseményekre is lehet őket alkalmazni.

- Megírásukhoz és futtatásukhoz engedélyre van szükség, ezt a rendszer egyes felhasználóknak automatikusan biztosítja.
- Ha a felhasználónak nincs jogosultsága a trigger lefutásához, az nem fut le, a teljes művelet érvénytelenítődik a trigger meghívása után.
- Amikor egy esemény triggert indít el egy táblára, akkor létrejön egy átmeneti trigger tábla, ahol követi a változásokat, és akár a törléseket is vissza tudja állítani.
- Sok memóriát, időt igényel.
- Felhasználónak szánt üzeneteket le lehet cserélni ezekkel, ami miatt rugalmasabb lesz az adatbázis. Az adatokat módosítás előtt és után össze tudjuk hasonlítani.
- Lehet használni naplózásra (user mikor mit csinált).
- Felhasználó nem tudja módosítani, ezért jó módszer bizonyos szabályok betartatására, fenntartja az adatintegritást.

Típusai:

- sorszintű: lefut, ha a tábla adatai módosulnak. DELETE esetén minden törölt sor aktiválja a triggert.
- utasításszintű: egyszer fut le, a kezelt sorok számától függetlenül.
- BEFORE-AFTER: BEFORE a hozzákapcsolt utasítás előtt kerül fut le, az AFTER utána. Lehet sor vagy utasításszintű. Csak táblához kapcsolhatóak, nézethez nem, de egy alaptáblához kapcsolt trigger lefut a nézeten végrehajtott DML-utasítás esetén is. DDL-hez kapcsolt is létrehozható, de csak adatbázison és sémán, táblán nem.
- INSTEAD OF: a hozzákapcsolt utasítás helyett fut le. Sorszintű lehet és csak nézeteken definiálható.
- rendszer: adatbáziseseményekről információkat ad (rendszeresemények, adatbázis elindítása, leállítása, szerverhiba, felhasználói események, bejelentkezés, kijelentkezés, DDL-utasítások (CREATE, ALTER, DROP), DML-utasítások (INSERT, DELETE, UPDATE)).
    - rendszeresemény, ki-be jelentkezés és DDL triggerek séma vagy adatbázisszinten hozhatóak létre.
    - DML-hez kapcsoltak táblákon és nézeteken hozhatóak létre.
    - saját sémában CREATE TRIGGER, másik felhasználó sémájában CREATE ANY TRIGGER, adatbázison ADMINISTER DATABASE TRIGGER jogosultság kell.

```
CREATE [OR REPLACE] TRIGGER [séma.]triggernév

{BEFORE|AFTER|INSTEAD OF}

{dml_trigger|{ddl_esemény [OR ddl_esemény]…|

ab_esemény [OR ab_esemény]…}

ON {DATEBASE|[séma.]SCHEMA}

[WHEN (feltétel)] {plsql_blokk|eljáráshívás}
```

Kollekciók

Azonos típusú adatelemek rendezett együttese. Minden elemnek egyedi indexe van, mely meghatározza a kollekción belüli helyét. Átadhatók paraméterként, így a táblák oszlopai mozgathatók az alkalmazások és az adatbázis között.

Típusai: asszociatív tömb, beágyazott tábla, dinamikus tömb.

Asszociatív tömb

Hash tábla, indexei (kulcs) tetszőleges egészek vagy stringek. Indexnek nincsenek határai. Deklarálható PL/SQL blokk, alprogram és csomag deklarációs részében. Csak PL/SQL programokban használható.

Beágyazott tábla

Elemei lehetnek objektumtípus példányai és ők lehetnek objektumtípus attribútumai. Index alsó határa 1. Deklarálható PL/SQL blokk, alprogram és csomag deklarációs részében. Létrehozható adatbázis objektumként a CREATE TYPE utasítással. PL/SQL programokban használható, és lehet adatbázistábla oszlopának típusa is, pl: ugyfel tábla konyvek oszlopa. Az elemek szétszórtan helyezkednek el és lehetnek közöttük lyukak. Új elemeket tetszőleges indexű helyre bevihetünk és bármely elem törölhető.

Dinamikus tömb

Elemei lehetnek objektumtípus példányai és ők lehetnek objektumtípus attribútumai. Index alsó határa 1. Deklarálható PL/SQL blokk, alprogram és csomag deklarációs részében. Létrehozható adatbázis objektumként a CREATE TYPE utasítással. PL/SQL programokban használható, és lehet adatbázistábla oszlopának típusa is, pl: konyv tábla szerzo oszlopa. Deklarációjához meg kell adni maximális méretet, ez lesz az index felső határa. Bővíthető új elemekkel, de az egyszer bevitt elemek csak cserélhetők, nem törölhetők. Az elemek a kisebb indextől indulva folytonosan helyezkednek el.