Csomagszállítás

Egy repülőtéri csomagszállító ***Kiskocsi*** működését modellezzük. A repülőtér utasainak ***N*** darab csomagját a kiskocsi segítségével szeretnénk egy repülőgéphez szállítani. A különböző tömegű csomagok a terminál futószalagján egymás után érkeznek, azokat érkezésük sorrendjében helyezzük a kiskocsira. A kiskocsi ***M*** tömeghatárig terhelhető.

A modell folyamatosan formálódó, egymást követően kialakuló változatai a valóság különböző dimenzióiban jönnek létre:

Osztály absztrakció – Kezdetben a *Kiskocsi* osztály modellje gondolati terünkben létezik a külvilág belső agy-nyelvi mintázataként – *érzet*.

Osztály objektum – Az osztály absztrakció agy-nyelvi mintázatát sikeresen vagy kevésbé sikeresen emberi nyelvre próbáljuk fordítani. Ez a reprezentáció szintén gondolati terünkben létezik, de már tudatosan – *tudatos gondolat*.

Osztály leírás – Az osztály objektum emberi nyelven megfogalmazott - a hangos kimondás vagy a néma nyelvi intonálás révén - tudatosult gondolatának leírt változata, olvasható formában, strukturált betűhalmazként. Belső gondolati terünkből kilépve, a külsődleges fizikai térben, papíron, megtekinthető formában létezik. – *tudatos gondolat leírt változata*.

Osztály source kód – A külsődleges fizikai tér gépi memóriájában, a fejlesztőkörnyezet belsejében létezik. Az emberi nyelvű osztály leírás emberközeli programozási nyelvre történt átfordítása, még mindig olvasható formában, strukturált betűhalmazként. A forráskód betűnkénti kódolást alkalmazva elektronikus formában is létezik – *az osztály leírás tovább kódolt, elektronikus változata*.

Osztály byte kód – A külsődleges fizikai tér gépi memóriájában létezik, az osztály source kód elektronikus változatának alapján gépi fordítóprogrammal előfordított, gépközeli programozási nyelvű kód – *gépközeli nyelvű „tervrajz”*.

Osztály natív kód – A fizikai tér gépi memóriájában létezik, az osztály egy példánya, a konstruktor által példányosított működőképes objektum, a byte kód alapján a JIT által „legyártott”, a processzor számára már ténylegesen érthető, ezáltal végrehajtható gépi kód, elkülönülve az osztály byte kódjától – *gépi nyelvű „gyártmány”*.

Készülő modellünk szempontjából az *osztály forráskód*jának elkészítése kulcsfontosságú lépés, hiszen ennek alapján adhatjuk át fordításra a gépi dimenziónak az „algoritmikus nyersanyagot”. Az osztály forráskód nyelvi elemeinek körültekintő kiválasztását az osztályleírás beszélt nyelvi struktúráinak elemzése teszi lehetővé. Mivel emberi nyelvünk segítségével saját környezetünket igyekszünk modellezni, megfelelő strukturális elemeinek kiválasztása a programozási nyelvet is alkalmassá teheti a környezet modellezésére. Valami hasonló történik ilyenkor, mint a matematika „mesterséges” nyelvének kialakításakor: a „kevesebb néha több” elvét alkalmazva igyekszünk megtisztítani nyelvünket terjengősségétől, pontatlanságaitól, korlátozzuk strukturális elemeit, mindezt egyetlen cél, a nyelv egzaktsága érdekében.

Világ-Modell

Világmodellt építünk. Világmodellünk tartalmi elemei meghatározzák az osztály forráskódban használni kívánt nyelvei elemeket.

Világ-modellünkben az energiaóceánban anyagszigetek jelennek meg. Az anyagszigetek kémiai kötések segítségével stabil vegyületeket alkotnak, strukturális állandóságuk van. Az állandó struktúrák az őket ért hatásokra konzekvensen hasonló módon reagálnak. A struktúrák és önazonos reakcióik a struktúrák szintjénél magasabb szinten létrejövő entitásokra utalnak, melyek az időben stabilak és jól felismerhetők – ezért van értelme alanyiságról, azaz *objektum*okról beszélni. Az objektumok strukturális állandóságához jól felismerhető *állapot*ok rendelhetők, melyek alkalmas *állapotjelző*kkel leírhatók. Egy objektum állapota a tapasztalat szerint az objektumot ért hatás következtében változik meg, az egyes változások okai maguk az objektumok. Ez az okság törvénye. Az objektumokat tehát nemcsak a pillanatnyi állapotuk jellemzi, hanem állapotváltoztató képességük is. Az objektum állapotváltoztató képességét, illetve az ennek hatására bekövetkező változási folyamatot *metódus*nak(*eljárás*) nevezzük. A metódus tehát az objektum által kifejteni képes hatásmechanizmus, annak következményeivel. A metódusoknak időbelisége van, az idő dimenziójában leírható koreográfiával rendelkeznek. A koreográfia minden esetben jellemezhető *ismétlés*ek és *elágazás*ok megfelelő sorozatával. A nem koreografált eljárást *elemi utasítás*nak nevezzük. Az objektumokat, metódusaikat, továbbá az objektumok állapotait nevekkel azonosítjuk, mely nevek egy jól meghatározott *névtér* elemei. A nevek használata azért szükséges, hogy ezáltal elvben lehatárolhassuk a valóság minket érdeklő részét. A tapasztalat szerint egy adott pillanatban, a makroszkópia szintjén, első közelítésben egymásra relevánsan ható objektumok száma korlátozott, az elvi lehatárolás tehát értelmes utánzása a valóságnak. Az elmúlt évszázadokban megalkotott fizikai képleteket tanulmányozva például azt látjuk, hogy egyetlen képletben sincs négynél több paraméter. (További érdekesség, hogy a képletekben egyetlen paraméter sem szerepel negyediknél magasabb hatványon.)

Az osztály forráskód mondatszerű nyelvi elemei

Eljárás

Eljárás vége

Ismétlés i = ( ) – ( )

Ismétlés amíg ( )

Ismétlés vége

Ismétlés

amíg ( )

Objektum

Utasítás( )

állapotjelző

állapotjelzők kapcsolata

Ha ( ) akkor

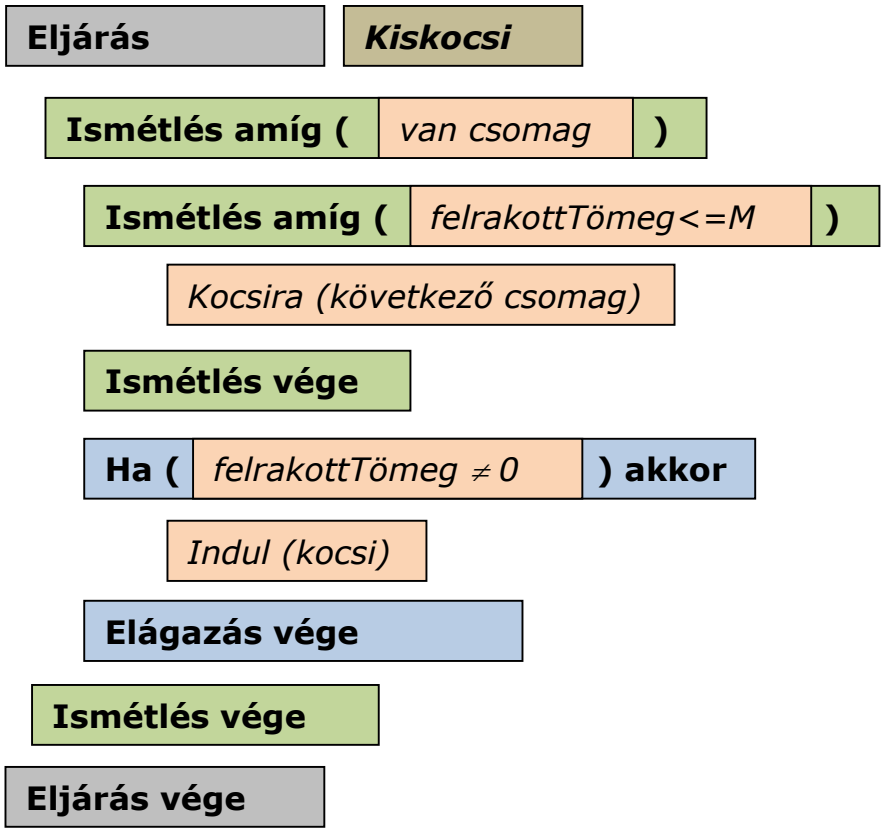
egyébként

Elágazás vége

A Kiskocsi működésének természetes nyelvű modellezése

Utasítás a kiskocsi vezetőjének: Kérem, pakolja fel a gép utasainak futószalagon érkező csomagjait a kiskocsira, és vigye azokat a kifutón álló repülőgéphez! Ha lehet, ne terhelje túl a kocsit, inkább forduljon többször! Köszönöm!

A Kiskocsi működésének mondatszerű nyelvi leírása



A mondatszerű nyelvi leírás logikai struktúráinak kialakítása után elkészítjük a forráskód programnyelvi változatát:

```
public class Kiskocsi {
  public static void main(String[] args) {
    int N=50;            //összes csomag száma
    int M=200;           //kiskocsi terhelhetősége
    int fdb=0;           //felrakott csomag
    long osszt;          //összes felrakott tömeg
    while (fdb<N){
      while (osszt<=M){
        osszt=osszt+kovetkezoCsomagTomeg;
        fdb=fdb+1;
      }
      if (osszt!=0){    //összes felrakott tömeg nem nulla
        Indul(kocsi);
      }
    }
  }
}
```

Látható, hogy a mondatszerű nyelvi leírás struktúrái és az ennek alapján elkészített programnyelvű forráskód struktúrái kölcsönösen egyértelműen megfelelnek egymásnak. Ez a fajta megfeleltethetőség nem magától értetődő, a programnyelv megalkotóinak sok munkája fekszik abban, hogy ez megvalósulhat. Ehhez több lépéses fejlesztőmunkára volt szükség. Egyrészt a természetes nyelvi leírást a már említett módon egzakttá kellett tenni, részben a megengedett nyelvi struktúrák maximális korlátozása árán, másrészt az erre alapozott programnyelvi kódokat gépi kódú fordításaik segítségével hosszú tesztelésnek kellett alávetni, majd ennek alapján optimalizálni a programnyelv elemeit és struktúráit. A leíró nyelv két közegben is zajló párhuzamos fejlesztése, és az eredmények folyamatos egymásra hatásának következtében kialakulhatott a bemutatott standard nyelvi eszközkészletet.

A fejlesztőkörnyezet fordítóprogramjának segítségével most már elkészíthetjük az osztály forráskódjából a byte kódot, ennek alapján pedig a futtatható natív kódot. Megkezdődhet a forráskód tesztelése az algoritmus natív kódú változatán keresztül. A forráskód teszteléseink nyomán egy jól meghatározható „fejlődési utat” jár be:

Szintaktikai állapot

A kód kezdetben tartalmazhat formális nyelvi hibákat, például hiányzik a sor végén szükséges pontosvessző vagy egy zárójelpár záró tagja. Ezeket a hibákat a gépi fordító fordítási időben a kód futtatása előtt észreveszi, ezért ezeket a hibákat *fordítási idejű hibáknak* nevezzük.

Szemantikai állapot

A kód szintaktikailag ugyan hibátlan, futási időben mégis „lefagy”, vagy kivételt dob. A megadott forráskódunkkal is megtörténhet, hogy végtelen ciklusba kerül, például túlsúlyos csomag esetén. A belső ciklusba nem lép be a vezérlés a túlsúly miatt, ezért a csomagszám adminisztrálása - `fdb=fdb+1;` - elmarad. Így viszont a külső ciklus újra és újra végrehajtódik. Ezek a hibák a gépi fordító számára csak a kód futtatása során derülhetnek ki, ezért az ilyen jellegű hibákat *futás idejű hibáknak* nevezzük.

Logikai állapot

Elképzelhető, hogy egy program hibamentesen lefut ugyan saját programkörnyezetében, de mégsem a kitűzött feladatot oldja meg, azaz *logikailag hibás*. A logikai hibákat kizárólag a programozó javíthatja, hiszen a gépi fordító nem „látja át” a program valósághoz fűződő viszonyát, nem „érti” a végrehajtott utasítások tartalmát.

A következőkben a forráskód tesztelésének tapasztalatait elemezzük.

A Kiskocsi osztály forráskódjának tesztelése

Az osztály forráskódjában a nyelvi leírásban nem szereplő változók jelennek meg. Ennek az az oka, hogy a nyelvi leírást mi készítettük és éreztünk magunkban annyi intelligenciát, hogy az egyes objektumok állapotait a szükséges pillanatokban képesek leszünk majd elemezni. Ezért a nyelvi leírásban nem is utaltunk a csomagok számára, de a kiskocsi terhelhetőségére sem. Annyi csomag lesz, amennyi lesz, a kiskocsi terhelhetőségét pedig le

fogjuk olvasni a rászegecselt fémbilétáról. A mázsamérleget is folyamatosan figyelni fogjuk a rakodás során, és nem fogjuk túlterhelni a kiskocsit, ahogy azt a műszakvezető kérte. Vannak tehát olyan képességeink, amelyek, tapasztalatunk szerint, velünk együtt „mozognak", nem szükséges formalizálnunk őket. Ha például elfogynak a csomagok, azt majd úgyis látni fogjuk. Ezek a képességek saját „rejtett" képességeink, mintha, saját magunkat objektumnak tekintve, p*rivate method()*szerűen lennének bennünk megfogalmazva. A tevékenység teljes modellezése ezért valójában kettévált: egyik részét explicit módon lemodelleztük, ezek a fizikai dimenzióban zajló események, a másik részét viszont nem(!), mert nem érzékeltük ennek szükségességét, ezek a gondolati dimenzió eseményei.

A modellezés során újonnan megalkotott leírónyelvünket használva kihívást jelent létrehozni modellünk eseménybázisát, a „koreográfiát", azonban még nagyobb kihívás a modellhez természetes módon csatlakozó logikai tér elemeinek tudatosítása. Gondolkodásunknak ekkor kissé hasonlatossá kell válnia a geometriai bizonyítások során megszokott diszkusszív szemléletű megközelítéshez: tudatosítanunk kell minden rejtett feltételezést. Lehetséges ugyanis, hogy rejtett feltételezéseink nem minden esetben valósulnak meg! Ha tehát gépesíteni szeretnénk a csomagok szállítását, automatánknak minden olyan „belső tudást" oda kell „adnunk", ami számunkra természetes, de tapasztalatok és intelligencia híján az automatának nem az! A programozás egyik legizgalmasabb része, hogy a modellezni kívánt eseménnyel kapcsolatos minden rejtett tudásunkat „előbányásszuk" gondolataink mélyéről, és az algoritmus leíráson keresztül odaadjuk az automatának, amely tudásra a legkülönfélébb esetekben szüksége lehet, például egy teljesen váratlan esetben. Feltételezésünk szerint a váratlan esemény bekövetkezésekor ugyanis mi nem kívánunk jelen lenni. A paraméterek megnevezése és kezdőértékük inicializálása éppen ennek a diszkussziónak kezdő lépései.

A forráskód formalizált, letisztult logikai állapota áttekinthetőbbé teszi modellünket, annak elemzése pedig több hibára is felhívja figyelmünket. A belső elöl tesztelős ciklus belépési feltétele azt vizsgálja, kisebb-e a kiskocsin lévő össztömeg a megengedettnél, és ha igen, gondolkodás nélkül felrakjuk a következő csomagot. Ekkor azonban túlléphetjük a terhelhetőséget. A feltételt tehát javítani kell: (ossztomeg + kovetkezoCsomagTomeg <= M). Egy szekvencia elemeinek ilyen jellegű vizsgálatát *előreolvasás*nak nevezzük.

Ennél a lépésnél egy újabb érdekes kérdés merül fel: hogyan lehet a *kovetkezoCsomagTomeg* névvel jelzett változóval matematikai műveleteket végezni? Ráadásul eddig ilyen változónk nem is volt. A csomagok és tömegeik mindenképpen léteznek. Szempontunkból érdektelen, hogy egyszerre kapjuk-e meg a tömeg értékeket, például azért, mert a csomagfelvételnél már lemérték az összes csomagot, vagy a csomagok érkezésekor mi mérünk. A logikai térben mindenképpen létezik a csomagok sorozata és ezáltal tömegeik sorozata is. A csomagokat ezért egy *Tömeg* nevű tömbbel reprezentáljuk, ahol *T(i)* az i-edik csomag tömegét jelenti, $1 \leq i \leq N$, $i \in \mathbb{N}^+$. A feltétel új alakja i=fdb esetén: (ossztomeg + T(fdb) <= M).

Korábban már észrevettük, hogy a fizikai dimenzióban zajló eseményeket explicit módon modellezzük, azonban a gondolati tér eseményeit nem. A

forráskód további tesztelése rámutat arra, hogy a fizikai dimenzió modellezése is kívánni valót hagy maga után. Amikor elszállítjuk a kiskocsira helyezett csomagokat a repülőgéphez és lepakoljuk azokat, a kiskocsi kiürül. Így a kiskocsin lévő csomagok össztömege a „valóság erejénél fogva” lenullázódik. Mi ennek adminisztrálásáról eddig elfelejtkeztünk! Az okság törvénye ezt automatikusan biztosította számunkra. Mégis szükséges az össztömeg „manuális” nullázása minden forduló után (össztomeg=0;). Ha ez az információ az automata „fejében” maradna, a kiskocsi nem indulna el többször.

A sorrendben utoljára érkező csomagok esetében elfordulhat, hogy terhelhető lenne még a kiskocsi, de elfogynak a csomagok. Az automata számára a nem létező következő csomag tömege értelmezhetetlen lenne, illetve végtelenségig növelné az *fdb* csomagszám értékét. Mindez azért fordulhat elő, mert egy szükséges feltételt a belső „ciklusfejtől távol” fogalmaztunk meg, és nem a belső ciklusfejben: fdb<N. A helyes belépési feltétel tehát a következő: (össztomeg+T(fdb)<=M és fdb<N).

Észrevehetjük azt is, hogy a külső ciklus végén alkalmazott logikai elágazás valójában felesleges, hiszen, ha beléptünk a külső ciklusba, akkor mindenképpen lesz csomag a kiskocsin. Mivel egy idő után ki fogunk lépni a belső ciklusból, vagy nincs több szállítandó csomag, vagy nem terhelhető tovább a kiskocsi, így mindenképpen indulni kell a felrakott csomagokkal a repülőgéphez. Az elágazásban eddig szerepeltetett *Indul(kocsi)* tulajdonképpen ezt az indulási szándékot jelenítette meg. Az absztrakció szintjétől függően itt részletes utasításokat adhatunk az automatának egy *Indul()* metódus segítségével, de elegendő lehet az indulások, azaz fordulók számának adminisztrálása is: induldb=induldb+1.

A belső ciklusban megfogalmazott *Kocsira (következő csomag)* metódust korábban már „lefordítottuk”:

```
osszt=osszt+kovetkezoCsomagTomeg;
fdb=fdb+1;
```

A bevezetett pontosított jelöléssel:

```
osszt=osszt+T(fdb);
fdb=fdb+1;
```

Az eddigi javítások alapján kialakult forráskód:

```
public class Kiskocsi {
  public static void main(String[] args) {
    int N=10;   int M=50;
    int fdb=1;  long osszt;   int induldb=0;
    while (fdb<N){
      osszt=0;
      while (osszt+T(fdb)<=M és fdb<N){
        osszt=osszt+T(fdb);
        fdb=fdb+1;
      }
      induldb=induldb+1;
    }
  }
}
```

Kialakított algoritmusunk már majdnem tökéletes, azonban tesztelése során *ArrayIndexOutOfBoundsException* kivételt kapunk. Ennek az az oka, hogy az utolsó, N-edik csomag kiskocsira helyezése után, belépve a ciklus belsejébe automatikusan növelni fogjuk az *fdb* csomagszámot, ezért a következő belépési feltétel ellenőrzésekor a T(N+1) N+1-dik tömbelemre fogunk hivatkozni az össztömeg vizsgálata során. Viszont nincs N+1-dik tömbelem! Két megoldás kínálkozik. Vannak olyan programozási nyelvek, amelyek akkor is kiértékelik egy „és" logikai operátorral összekapcsolt kifejezés mindkét részét, ha az egyik rész (fdb<N) hamis értéket szolgáltat. Ekkor nem tehetünk mást, be kell vezetni egy nullaértékű pszeudo tömbelemet a tömb utolsó elemeként. Vannak olyan programozási nyelvek, amelyek az *A és B* összetett logikai kifejezés kiértékelésekor befejezik a kifejezés kiértékelését akkor, ha *A* hamis. Ekkor ugyanis *A és B* mindenképpen hamis lesz. A JAVA nyelv ezen utóbbi nyelvekhez tartozik. A feltételek sorrendjének megcserélése esetén tehát nem kell bevezetnünk a 0 értékű pszeudo elemet.

Van még egy feladatunk. Tudatnunk kell az automatával a csomagok tömegértékeit! Például: T ={23, 11, 5, 34, 19, 41, 3, 7, 19, 25};

Új kódunkban *fdb* értékét 0-val indítjuk, ugyanis a JAVA nyelvben a tömbelemek kezdőindexe nem 1, hanem 0. Hibamentes kódunk, melyet kiegészítettünk a kialakuló eredmények kiírásával a következő lesz:

```
public class Kiskocsi {
  static int [] T ={23, 11, 5, 34, 19, 41, 3, 7, 19, 25};
  public static void main(String[] args) {
    int N=10;   int M=50;
    int fdb=0;  long osszt;   int induldb=0;
    System.out.print("Szállított tömegek: ");
    while (fdb<N){
      osszt=0;
      while ((fdb<N) && (osszt+T[fdb]<=M)){
        osszt=osszt+T[fdb];
        fdb=fdb+1;
      }
      induldb=induldb+1;
      System.out.print(osszt+ " kg, ");
    }
    System.out.println("\nA kiskocsi "+induldb+" alkalommal fordult.");
  }
}
```

```
run:
Szállított tömegek: 39 kg, 34 kg, 19 kg, 44 kg, 26 kg, 25 kg,
A kiskocsi 6 alkalommal fordult.
```

További változtatásokat szeretnénk eszközölni a kódban, ugyanis a két egymásba ágyazott ciklus belépési feltétele majdnem azonos. Bevezetjük a *rakodas [ ]* segédtömböt. A segédtömb tárolni fogja az egyes fordulók alkalmával elszállított csomagok össztömegét. Az egyszerre szállított tömegeket eddig csak kiírtuk a kimenetre, azonban ez nem tekinthető visszaadott értéknek, ugyanis a monitor pixeleiről nem lehet visszahívni a kiírt értékeket.

```
public class Kiskocsi {
  static int [] T ={23, 11, 5, 34, 19, 41, 3, 7, 19, 25};
  static int [] rakodas = new int[T.length];
  public static void main(String[] args) {
    int N=10; int M=50;
    int fdb=0; int db=0; //db ≡ induldb
    while (fdb<N) {
      if (rakodas[db]+T[fdb]>M){
        db=db+1;
      }
      rakodas[db]= rakodas[db]+T[fdb];
      fdb=fdb+1;
    }
  }
}
```

A csomagok tömegének előzetes lemérése esetén feltételezhetjük, hogy nincs túlsúlyos csomag. Ha rakodás előtt közvetlenül mérünk, erre az esetre is fel kell készülni. A megoldás szempontjából releváns körülményeket (paramétereket) minden esetben a feladat kitűzésekor a feladat *specifikáció*jában kell meghatározni.

```
public class Kiskocsi {
  static int [] T ={23, 11, 5, 34, 19, 41, 3, 7, 19, 25};
  static int [] rakodas = new int[T.length];
  public static void main(String[] args) {
    int N=10; int M=25;
    int fdb=0; int db=0; //db ≡ induldb
    int nemszallithato=0;
    System.out.println("A kiskocsi terhelhetősége: "+M+" kg");
    System.out.print("Szállított tömegek: ");
    while (fdb<N) {
      if (rakodas[db]+T[fdb]>M){
        db=db+1;
      }
      if (T[fdb]>M){
        fdb=fdb+1;
        nemszallithato=nemszallithato+1;
      }
      rakodas[db]= rakodas[db]+T[fdb];
      fdb=fdb+1;
    }
    for(int i=0;i<=db;i++){
      System.out.print(rakodas[i]+ " kg, ");
    }
    System.out.println("\nA kiskocsi "+(db+1)+" alkalommal fordult.");
    if (nemszallithato!=0){
        System.out.print(nemszallithato+ " darab csomag túlsúlyos.\n");
    }

  }
}
```

```
run:
A kiskocsi terhelhetősége: 25 kg
Szállított tömegek: 23 kg, 16 kg, 19 kg, 10 kg, 19 kg, 25 kg,
A kiskocsi 6 alkalommal fordult.
2 darab csomag túlsúlyos.
```