NEUMANN JÁNOS
INFORMATIKAI KAR

Bedők Dávid

Programozási feladatok megoldási módszertana

ÓE-NIK 5018

Budapest, 2015.

Készült az Óbudai Egyetem Neumann János Informatikai Karán az ÓE-NIK 5018. sz. jegyzetszerződés keretein belül 2015-ben.

Szerző: Bedők Dávid
ügyvivő szakértő
bedok.david@nik.uni-obuda.hu

Lektor: Dr. Szénási Sándor
egyetemi docens
szenasi.sandor@nik.uni-obuda.hu

1.1. verzió
2016. január 27.

A jegyzet legfrissebb változata letölthető az alábbi címről:
http://users.nik.uni-obuda.hu/david.bedok/progi-felok-megoldasi-moda-latest.pdf

Ez a jegyzet LaTeX segítségével készült.

ISBN 978-615-5460-66-1

Tartalomjegyzék

Bevezetés

Ez a jegyzet az Óbudai Egyetem Neumann János Informatikai Karán Mérnök informatikus alapszakon tanuló hallgatók számára készült. A jegyzet a Programozás I. és a Programozás II. tárgyak labor foglalkozásaihoz íródott. A labor foglalkozások során a hallgatók elsajátítják az algoritmikus gondolkodás alapjait, illetve az algoritmusok implementálásának képességét C# programnyelven. A kiválasztott programozási nyelv évek óta a domináns nyelv az intézményben, azonban mindez nem jelent korlátozást az elsajátítandó gondolkodásmód és algoritmizálási képesség terén. Egyrészt a C# egy általános, hibrid típusú programozási nyelv, melyben egyszerűen megvalósíthatóak elemi algoritmusok illetve az objektum-orientált paradigmák szerint felépített alkalmazások, másrészt egyáltalán nem a C# nyelv specifikus elsajátítása az oktatási célja a kapcsolódó tárgyaknak, így a megszerzett tudás könnyen átültethető - hasonlóan magas szinten értelmezett - más programozási nyelvekre (pl. Java, Python).

A jegyzet anyaga részben fedi, részben pedig kiegészíti a labor foglalkozásokon bemutatásra kerülő algoritmusokat, programtervezési módszereket. Lépésről lépésre bevezeti az olvasót a legelső futó alkalmazásának elkészítésétől egészen az összetettebb objektumorintált tervezési mintákat is tartalmazó alkalmazás megtervezéséig. A jegyzet legtöbb esetben mindig egy előre meghatározott valós (vagy valós helyzet által szült) problémára keres algoritmikus megoldást. A jegyzetnek nem célja az algoritmusok helyességének, illetve elemzésének leírása. E célból az említett tárgyak előadásaihoz már készült szakirodalom (Sergyán Szabolcs: Algoritmusok, adatszerkezetek I. illetve Szénási Sándor: Algoritmusok, adatszerkezetek II.).

A jegyzet nem tér ki a C# programozási nyelv szintaktikai szabályainak ismertetésére, és a vezérlési szerkezetek sem lesznek nyelv specifikusan bemutatva. E célból számos könyv íródott már, illetve az Internet is gazdag forrása jobbnál jobb szakmai anyagoknak.

A jegyzetnek nem célja referencia könyvnek lenni az érintett algoritmusok C# implementációjának. A bemutatott forráskód részletek nem végtermékként, hanem elkészülésük egymásra épülő lépéseivel együtt érik csak el azt az oktatási célt, hogy az algoritmikus gondolkodás és implementációs tervezés alapjaival az olvasó megismerkedhessen. Alapvető fontosságú egy algoritmus implementációjakor az, hogy a forráskód szintaktikailag helyes legyen, azonban mindez szinte semmit sem növel a munka minőségén (a szintaktikailag helytelen programot nehéz bármilyen szinten értékelni). A kezdő fejlesztő/mérnök számára a szintaktika elsajátítása után elsődleges cél lesz a kívánt probléma/problématér megoldásának előállítása, vagyis hogy a már futó alkalmazás a helyes kimenetet állítsa elő. Ez már sok ponton értékelhető eredményt adhat, azonban a szerző ezt még mindig nem tekinti egy (leendő) mérnök által megírt alkalmazásnak. Ahhoz, hogy egy kódtömeget alkalmazásnak hívhassunk, minimális követelmény a helyes output előállítása a problématér minden lehetséges elemére. A minőségi ugrások az implementáció miként jében jelennek meg, illetve annak tiszta és olvasható voltában. A jegyzet egyik célja eme tiszta kód[9] előállításában való útmutatás.

Adódik a kérdés, ha a jegyzet nem foglalkozik szintaktikával és nem kíván a tanult algoritmusok referencia könyvének sem lenni, akkor miként tekintsen rá az olvasó. A szerző elsődleges célja az algoritmikus gondolkodás lépésről lépésre való bevezetése, illetve e közben a szakirodalom által is ismert programozási módszertanok bemutatása. Akkor éri el a jegyzet a célját, ha utóbbiak használatára az igényt is fel tudja kelteni, és nem valamilyen "kötelezően" használandó módszertan leírásának hamis képét vetíti az olvasó elé.

A bemutatott forráskódok készítése során a szerző kiemelten figyelt arra, hogy egy tiszta, jól strukturált, logikusan felépített képet nyújtsanak, mely mások számára is egyértelmű és megjegyzések nélkül is gyorsan értelmezhető. Ahol szükséges, a jegyzet kitér ennek a kódolási technikának apró részleteire.

Az 1. fejezet segíti az olvasót abban, hogy a téma előadásán elsajátított programozási tételeket megfelelő módon tudja a gyakorlatba ültetni. A 2. fejezetben véletlen teszt adatok generálásával, és ezáltal az elkészült algoritmusok egyszerű szimulációjával bővítjük ismereteinket. A kettő együttesen már megfelelő alap lehet ahhoz, hogy komplexebb alkalmazásokat készítsünk. A téma feldolgozása során az olvasó ekkora megismeri az objektum-orientált programozás elméleti alapjait, melyet a 3. fejezetben bemutatásra kerülő grafikai lehetőségekkel kiegészítve már képes lesz arra, hogy önállóan egy egyszerűbb üzleti probléma köré egy kész alkalmazást építsen.

A 4. fejezet már elsősorban a *Programozás II* tantárgy anyagához köthető, és a benne lévő nyelvi ismeretek a könyvben ezt követően elő-elő fordulhatnak, ám nem kizárólagosan. A fejezet átfutása után az 5. fejezet eleddig csak részben érintett programozási technikákat vesz sorba. Elsősorban 5.3. fejezet az, mely épít a *Programozás II* tantárgy elméleti anyagára, a fejezet többi része korábban is bátran feldolgozható.

A 6. fejezet a jegyzet által nyújtott fejlesztési technikák összefoglalója, melyben az olvasó egyre komplexebb játékprogramok tervezésével és készítésének lépéseivel ismerkedik meg. A fejezet teljes feldolgozásához szükség lehet a *Programozás II* tantárgy során elsajátított ismeretekre is, azonban az előzmény tárgy tematikája elegendő a sikeres adaptáláshoz, lévén külön alfejezetek foglalkoznak a "technikásabb" részekkel.

A jegyzet utolsó, 7. fejezetében az objektum-orientált tervezés során gyakran alkalmazott tervezési minták közül tekintünk át néhányat, ezzel nem titkoltan a szerző keretet kívánt adni a műnek: a gyakori algoritmizálási problémákat bemutató programozási tételektől jut el az olvasó a gyakori tervezési problémák megoldásaként is értelmezhető tervezési mintákig. Ezen utolsó fejezet csupán egy ízelítő a témából, elsősorban a *Programozás II* tantárgy hallgatói számára javasolt a feldolgozása.

A jegyzetben előfordulhatnak hibák. Nem merjük állítani ennek ellenkezőjét, azonban célunk e hibák mielőbbi kijavítása, pontosítása. Kérjük, hogyha egy-egy fejezet nehezen érthető, vagy egy téma alaposabb kifejtést vagy más megközelítést érdemelne, jelezze ezt a szerző felé. Közös célunk, hogy olyan jegyzet kerüljön az olvasó kezébe, ami a tananyag megértését és elsajátítását szolgálja.

Nevezéktan és program szervezési szabályok

Ma már minden fejlesztési munka szinte kivétel nélkül kisebb-nagyobb csoportban zajlik. Ritka az, hogy egyszemélyes, úgynevezett "hős programozók" lennének. Egy-egy ilyen fejlesztési feladatra verbuválódott csapat a fejlesztés első szakaszában meg kell hogy egyezzen valamilyen szabályrendszerben, melyben a csapat minden egyes tagja teljes egészében egyetért, és a jövőre nézve betartja. E szabályok halmaza az egyetértés szintjével arányos, vagyis minél több dologban értenek egyet a résztvevők, annál több és pontosabb

szabály határozható meg. A szabályok kitérnek többek között a forráskód szervezésének mikéntjére, az egyes elnevezések használatára, a fordítás, telepítés lépéseire, a verziókezelés szabályaira, a review és a refaktorálás alkalmazásának idejére és szükségességére, illetve gyakorlatilag bármire, amiben egyetértés van. Ezen szabályok jelentős részét ma már automatizált módszerekkel is lehet ellenőrizni (és ezzel kiváltani a szabályok alkalmazását), mely hatékony módszer lehet új csapattagok integrálásakor (is), azonban sosem szabad megfeledkezni arról, hogy ezen szabályokat mindenkinek belső késztetésből kell betartania, vagyis azért kell így eljárni a fejlesztés során, mert egyetértünk e szabályokban.

A jegyzet írása során is lesz egy ilyen szabályrendszer, melyet a szerző amennyire lehetséges, belső késztetésből be fog tartani. A különbség itt természetesen az, hogy az olvasó e szabályrendszer egyeztetésekor nem volt jelen, ezért e szabályok egy részével bizonyosan nem fog egyetérteni. Ezzel nincs különösebb probléma, azonban az olvashatóságot megkönnyíti, ha ismeri ezen szabályokat.

Elsősorban algoritmusokat tárgyal a jegyzet C# nyelvi szintaktikával, így számos nevezéktani szabály a kiválasztott nyelv belső szabályrendszeréből adódik. A teljesség igénye nélkül az alábbi szabályok a legfontosabbak annak érdekében, hogy már az első fejezetekben se legyen kérdés egy-egy létrehozott nyelvi elem helye, megnevezése (Természetesen minden szabály csak a hozzá adott magyarázattal együtt értelmezhető. Ha egy szabályra nincs magyarázat, azt senkinek sem kell betartania. A magyarázatok egy részére a jegyzet fejezeteiben is lesz utalás.).

- A programozási nyelvek szintaktikája angol nyelvű és a belső programkönyvárak szintén angol nyelven készültek. Bármilyen más nyelv használata egy forráskódban (a literálokat és igény szerint alkalmazott dokumentációs leírásokat nem figyelembe véve) nem észszerű, ennélfogva kerülendő. Ez nem nyelvi preferencia kérdése.
- A forráskódokban inline commentnek nincs helye. Néhány különleges kivételtől eltekintve (pl. regexp kifejezés használata során), ha egy kódsor csak és kizárólag a beszúrt megjegyzés segítségével értelmezhető, akkor a kódrészlet hibás szervezésű, vagy rossz névvel rendelkezik.
- Minden típus neve kezdődjön nagybetűvel. A név lehetőség szerint valamilyen főnév közeli kifejezés legyen.
- Minden mező neve kezdődjön kisbetűvel.
- Minden metódus neve kezdődjön nagybetűvel [1], és lehetőség szerint valamilyen ige közeli kifejezés legyen.
- A konstansok és osztályszintű végleges adattagok nevei csupa nagybetűsek legyenek. Szóelválasztó elemnek az aláhúzás karaktere legyen használva.
- A nem csupa nagybetűs nyelvi elnevezések a CamelCase írásmódot kövessék. Mellőzzük az aláhúzás használatát az érintett kontextusokban.
- A Hungarian Notation és változatainak használata kerülendő [2].

[1] Ezzel a szabállyal a szerző nem ért egyet, azonban a C# a metódusok neveit így használja a belső programkönyvtárakban, zavaró lenne ha ehhez nem illeszkednénk.

[2] E szabály alól később fogunk egy-két kivételt megfogalmazni, illetőleg a C# is alkalmazza (sajnos) néhol az írásmódot belső programkönyvtáraiban.

- Minden namespace neve kezdődjön nagybetűvel.
- Minden namespace-nek legyen a forrásban egy vele azonos nevű könyvtára, és ezen könyvtár pontosan ott helyezkedjen el, ahol a namespace is elhelyezkedik a többi namespace hierarchiájában.
- Minden ún. Top Level Type egy saját nevével ellátott forrás állományban szerepeljen, pontosan abban a könyvtárban, melyet a típust közvetlenül befoglaló névtér számára meghatároz. Pl. a Car osztály a Car.cs állományban legyen megtalálható.
- Minden esetben jelezzük az egyes nyelvi elemek láthatóságát, ha az rajtuk értelmezett, függetlenül az alapértelmezett láthatóságtól [3].
- A forráskódok a Visual Studio alapértelmezett kódformázási szabályai szerint legyenek formázva (az egyszerűség és egyértelműség kedvéért).

Még számos szabály elő fog kerülni a jegyzetben, a felsoroltak azonban általánosságban igazak minden fejezetben leírt forráskódra nézve. A *Top Level Type*-ok, illetve a *namespace* fizikai állományokhoz és könyvtárakhoz kötése a Java nyelvből származik, ahol mindez nem lehetőség, hanem egy kötelező és kikerülhetetlen érvényű szabály. Bár a C# ezt nem írja elő, a Visual Studio már évek óta támogatja a fenti szabályrendszer kényelmes használatát (pl. ha átnevezünk egy állományt, melynek neve megegyezik a benne szereplő *Top Level Type* nevével, felajánlja a típus refaktorálását (átnevezését), illetve ha egy adott könyvtármélységben létrehozunk egy új típust, annak névtér hierarchiája automatikusan a könyvtárak hierarchiájából fog származni).

Köszönetnyilvánítás

E jegyzet nem születhetett volna meg a szerző tollából, ha sok évvel ezelőtt Dr. Vámossy Zoltántól nem kapta volna meg azon alapokat, melyek a mai napig végigkísérik szakmai életét. A számos nagyszerű könyv és szakmai anyag szerzői közül Robert C. Martin külön kiemelést érdemel, az ő iránymutatásai és tanácsai nagyban hozzájárulnak ahhoz, hogy a kor követelményeinek megfelelő munkák kerüljenek ki a szerző kezei közül.

Köszönet illeti Dr. Sergyán Szabolcsot, és Dr. Szénási Sándort, akik a Programozás I. és Programozás II. tárgyak előadásaihoz elkészítették Algoritmusok, adatszerkezetek I. illetve Algoritmusok, adatszerkezetek II. című jegyzeteket, melyek fejezetei iránymutatást adtak jelen munkához is. Dr. Szénási Sándort mint a jegyzet lektorát külön is köszönet illeti meg. Sok hasznos észrevételt köszönhet a szerző Czigány Andrásnak is, aki az Ericsson Magyarország mérnökeként és egyben a szerző munkatársaként átolvasta a jegyzet munkaverzióját.

Végül, de nem utolsó sorban szeretnék köszönetet mondani az Alkalmazott Informatikai Intézet munkatársai közül Cseri Orsolya Eszternek és Szabó Miklós Zsoltnak, akikkel az elmúlt évek során számos alkalommal egyeztettük a labor foglalkozások során bemutatott alkalmazásokat és módszereket.

[3] Ez a szabály az olvashatóságot javítja, bár kétség kívül többlet munkával jár. Kezdő fejlesztők számára nagyon ajánlott a használata, ezért került bele a jegyzet szabályai közé.

1. fejezet

Programozási tételek

Ismeretszerzés

fekete doboz elve, kliens kód, névadás, metódusok mérete, sorrendje és felelőssége, hatékonyság, problématér felbontása

A programozási tételek olyan matematikailag is bizonyítható helyességű és futási idejű algoritmusok csoportja, melyek alapvető problémákra adnak kész megoldást. Kész megoldás alatt azt értjük, hogy egy-egy tétel önmagában is implementálható, bár valós problématér esetén általában a tételek kisebb-nagyobb módosítására van szükség. Egy (szoftver) fejlesztéssel foglalkozó szakember munkája elképzelhetetlen volna, hogyha nem tudná valamennyi programozási tételt készség szinten bármely felmerülő problématér esetén azonnal implementálni. Nem szabad legyinteni ezen algoritmusok első ránézésre tűnő egyszerűségüknek. A bennük lévő módszerek olyan alapvető építőkövei az algoritmikus gondolkodásnak, melyet ha nem sajátítunk el ismeretszerzésünk elején, sosem válhat belőlünk igazi programozó.

Ma már számos magas szintű programozási nyelv támogatja a programozási tételek deklaratív alapokon nyugvó implementációját, ami mondhatni egybe esik a problématér megoldásának megfogalmazásával (C# nyelvi kontextusban a LINQ[1] eszköztára nyújt többek között ehhez rendkívül jó támogatást). Ez ma már így természetes, azonban mindez semmiképpen sem jelenti azt, hogy a programozási tételek algoritmusának ismeretére ne volna ugyanolyan nagy szükségünk. Pontosan azért lehet "kérni" a fordítótól egy adott problématérre nézve egy programozási tétel "gépi" implementációját, mert egyrészt pontos matematikai alapokon nyugszanak, másrészt minden fejlesztő pontosan ugyanazt írná le imperatív módszerekkel, amit a "gép" megtesz jelenleg a fejlesztő helyett.

A fejezetben a programozási tételeket imperatív módszerek szerint fogjuk implementálni, azonban ugyanúgy célja kell legyen egy alkalmazásnak az, hogy egy adott szintjén ugyanolyan olvasható legyen a probléma megfogalmazása, és az arra adott eredmény kinyerése, mintha ezt deklaratív eszközökkel készítettük volna.

[1] Language-Integrated Query

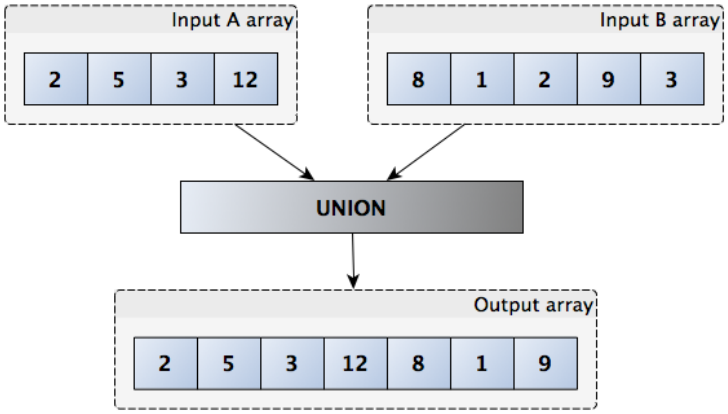
1.1. Unió

Előfeltétel

változók, hatókörök, literálok, alprogramok, fix elemszámú tömbök

Határozzuk meg két egész számokat tartalmazó véges elemszámú halmaz unióját. Feltételezhetjük, hogy a bemeneti halmazok mindegyike ismétlődésmentes. A halmazokat fix elemszámú tömb adatszerkezetekkel reprezentáljuk. A megoldás során az objektum-orientált programozás szabályai figyelmen kívül hagyhatóak.

A feladat megoldására tisztán az *unió programozási tétel*[1] alkalmazható. Az implementáció szintjén a lényegi algoritmust egy metódus formájában implementáljuk. Tekintsük ezt a metódust egyfajta fekete doboznak, melynek bemeneti oldalára két tömböt vezetve a kimeneten előáll egy olyan új tömb adatszerkezet, melyben a bemeneti tömbökben előforduló összes elem szerepel, pontosan egyszer.



1.1. ábra. Két halmaz uniója

Bármilyen konkrét implementációt választunk, a fenti fekete dobozos modellnek célszerű megjelennie nyelvi szinten is. Ez más szavakkal úgy is megfogalmazható, hogy a megoldásunkat képesnek kell lennie használni egy olyan felhasználónak, aki mit sem tud az unió algoritmusáról. Utóbbi esetén természetesen nem valamilyen csinos felhasználói felületről van szó, felhasználó alatt egy fejlesztőt értünk, aki az általunk megírt algoritmus/alkalmazás meghívására képes. Jelen feladat során mindez természetesen egy metódus formájában tökéletesen tud reprezentálódni, azonban a fent leírt módszert a jegyzet összes többi feladata során is igyekszünk majd alkalmazni.

1.1.1. Kliens kód készítése

Nagyon sokat segít abban, hogy az implementáció során ne veszítsünk célt, ha a feladat megoldását nem magával az unió algoritmusának "gépelésével" kezdjük el, hanem e fekete doboz keretének elkészítésével (kliens kód), és meghívásával. Ez az eljárás az alapja a későbbiekben még szóba kerülő TDD[2] módszerének.

[2] Test Driven Development

```
1  using System;
2
3  namespace ProgrammingTheorem
4  {
5    public class UnionApplication
6    {
7
8      private static void Main(string[] args)
9      {
10       TestUnion();
11     }
12
13     private static void TestUnion()
14     {
15       int[] inputA = { 2, 5, 3, 12 };
16       int[] inputB = { 8, 1, 2, 9, 3 };
17       Console.WriteLine("(A): " + ArrayToString(inputA));
18       Console.WriteLine("(B): " + ArrayToString(inputB));
19       int[] output = Union(inputA, inputB);
20       Console.WriteLine("(A union B): " + ArrayToString(output));
21     }
22
23     private static String ArrayToString(int[] elements)
24     {
25       return "[" + String.Join(", ", elements) + "]";
26     }
27
28     private static int[] Union(int[] dataA, int[] dataB)
29     {
30       return new int[]{};
31     }
32
33   }
34 }
```

1.1. kód. Alkalmazás kerete

Létrehozzuk a bemeneti tömböket (vizuális hatás kedvéért megjelenítjük az adatokat a képernyőn), majd meghívjuk az *Union* metódust. A visszakapott tömb kimenetet eltároljuk, majd hasonló módon megjelenítjük. Annak érdekében, hogy forduló alkalmazást kapjunk, az *Union* metódusnak vissza kell térnie egy egész számokat tartalmazó tömbbel. Válasszunk egy egyszerű és gyors megoldást erre (a példában egy elemeket nem tartalmazó üres tömbbel térünk vissza).

Ha futtatjuk a programot, nyilvánvalóan helytelen eredményt fog adni. Nem is az a fontos, hogy ezen most nem lepődünk meg, sokkal inkább a következő gondolkodásmód a lényeges: "mivel a két bemeneti halmazom tartalmaz egész számokat, az üres tömb, mint kimenete az unió tételének, nem lehet helyes". Vagyis még mielőtt futtatjuk az alkalmazást, fejben végiggondoljuk, hogy mit is várunk a kimeneten. Fordítva a történet mit sem ér.

Az elkészült keret alkalmazásnak a lényegi része az *Union* metódus aláírása, vagyis tömb paraméterei, és tömb kimenete (1.1. kód 28. sor). Ez a szintaktika reprezentálja számunkra azt a fekete dobozt, amit a feladat felvetésekor felrajzoltunk.

A jegyzet további fejezeteiben a forráskód részletek nem fogják tartalmazni a feladat megoldása szempontjából lényegtelen elemeket, mint pl. a felhasznált névterek listája (*using*), a namespace deklaráció és a *Main()* metódus. Az aktuális feladat mellőzi az objektum-orientált programozás elveit, csak annyira tartalmaz szintaktikai megkötéseket e kapcsán, amennyire minimálisan szükséges (pl. láthatóságok).

A feladat már most is tartalmaz négy metódust, melyeknek a sorrendje az adott kontextusban nem véletlen. Minden metódus pontosan azon metódust követően van definiálva, ahol annak első használata megtalálható. Ez nem egy olyan szabály, amelytől nem

lehet eltérni, azonban később majd látni fogjuk, elsősorban *private* láthatóságú metódusok során rendkívül hasznos az olvashatóság szempontjából az, mikor keressük egy metódus implementációját, az legtöbbször a hívás helye alatt megtalálható lesz.

1.1.2. Algoritmus implementálása

Nincs más dolgunk, mint az ismert *unió programozási tételt* implementálni pl. pszeudokód alapján. Az algoritmus helyességére és miként tjére a jegyzetben nem térünk ki, arra viszont igen, hogy a fix elemszámú tömbök kezelésével kicsit foglalkoznunk szükséges. Vannak nyelvek, ahol a dinamikus tömbök kezelése összemosódik a fix elemszámú tömbökkel, a C# viszont a kettőt külön kezeli, és a felhasználóra bízza, mikor melyiket használja (mivel mindegyiknek megvan a maga előnye és hátránya).

A fix elemszámú kimeneti tömb létrehozásával nem tudjuk kezdeni a megoldásunkat, mivel nem tudjuk előre a különböző elemek számát. Azonban egy felső korlátot meg tudunk határozni! Biztos, hogy nem lesz több elem a kimeneten a két bemeneti tömb elemszámainak összegénél. Kihasználva ezt már létre tudunk hozni egy ideiglenes tömböt, melyben az egyedi elemeket fogjuk lépésről lépésre eltárolni. Miközben az algoritmus sorban feltölti elemekkel e tömböt, egyben számolja is ezen elemek számát. Az algoritmus legvégén nincs más dolgunk, mint az eredetileg felső korláttal létrehozott tömböt csonkolni a szükséges méretig. Ezt a műveletet végzi el a *Cropping()* metódus.

```
1  private static int[] Union(int[] dataA, int[] dataB)
2  {
3    int[] result = new int[dataA.Length + dataB.Length];
4    int index = 0;
5    for (int i = 0; i < dataA.Length; i++)
6    {
7      result[index++] = dataA[i];
8    }
9    for (int i = 0; i < dataB.Length; i++)
10   {
11     int k = 0;
12     while ((k < dataA.Length) && (dataB[i] != dataA[k]))
13     {
14       k++;
15     }
16     if (k == dataA.Length)
17     {
18       result[index++] = dataB[i];
19     }
20   }
21   return Cropping(result, index);
22 }
23
24 private static int[] Cropping(int[] source, int length)
25 {
26   int[] data = new int[length];
27   if (length <= source.Length)
28   {
29     for (int i = 0; i < length; i++)
30     {
31       data[i] = source[i];
32     }
33   }
34   return data;
35 }
```

1.2. kód. Unió tétele

1.1.3. Ciklusváltozók, kód tömörítés

Néhány egyszerű ám sokszor annál értékesebb apróságra is érdemes odafigyelni a bemutatott algoritmusban (1.2. kód). A ciklusváltozók nevei mindig rövidek, gyakran egy betűsek, és legtöbbször az *i* betűt használjuk e célra (az *iteration* angol szóból). Ha több egymásba ágyazott ciklusunk van, akkor gyakran az abc következő betűje, a *j* szokott a belső ciklus változója lenni, azonban ez nem javasolt. A két ciklusváltozó ez esetben nagyon hasonlítana egymásra, célszerűbb egy másik betűt választani (a szerző erre a *k* betűt tartja fenn). A másik apróság a kód tömörítésének példája a 1.2. kód 7. és 18. sorában. Általánosságban az mondható el, hogy magas szintű programozási nyelven ne helyezzünk el lehetőség szerint fejtörőket a forráskódban, vagyis szép dolog spórolni három leütött karaktert, de nem minden esetben kifizetődő, ha később mi magunk is (nem beszélve munkatársainkról) perceket gondolkodunk az adott kódsor értelmezésén. Az említett sorokban azonban mégis két utasítás egységbe van zárva: értéket adunk a tömb soron következő elemének, majd növeljük az *index* "mutató" értékét egyel. Az alkalmazott összevonás használata ez esetben szándékos, és üzenete a két művelet szétválaszthatatlan, ún. *atomic* volta.

1.1.4. Deklaratív megoldás

Bár az algoritmizálási képességünket nem fejleszti, említsük meg hogy a C# LINQ lehetőségei miatt egyszerűen meghatározhatjuk ugyanezen eredményt. A 1.3. kód pontosan ugyanolyan egyszerű és deklaratív leírásmód, melyet egy ilyen szituációban elvárunk (a forráskód részletben megtalálható [..] blokk a már bemutatott, avagy oda illő, de nem részletezett kódsorokat jelöli, szintaktikailag helytelen a begépelése).

```
1  using System;
2  using System.Linq;
3
4  namespace ProgrammingTheorem
5  {
6    public class UnionApplication
7    {
8      [..]
9
10     private static void TestUnionWithLINQ()
11     {
12       int[] inputA = { 2, 5, 3, 12 };
13       int[] inputB = { 8, 1, 2, 9, 3 };
14       Console.WriteLine("(A): " + ArrayToString(inputA));
15       Console.WriteLine("(B): " + ArrayToString(inputB));
16       int[] output = inputA.Union(inputB).ToArray();
17       Console.WriteLine("(A union B): " + ArrayToString(output));
18     }
19
20   }
21 }
```

1.3. kód. Feladat megoldása LINQ segítségével

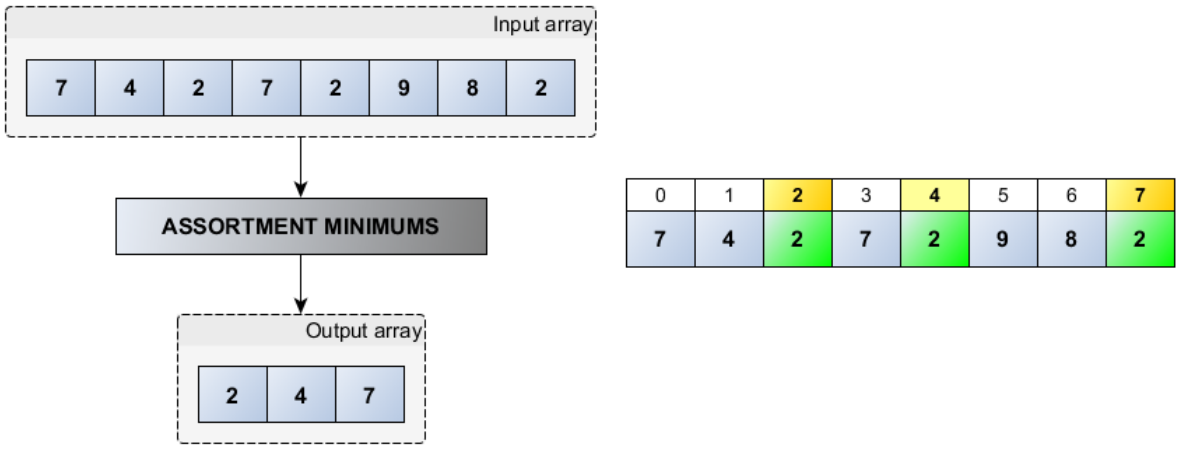
1.2. Minimumhelyek kiválogatása

Előfeltétel

változók, hatókörök, literálok, alprogramok, fix elemszámú tömbök

Készítsünk alkalmazást, mely egy pozitív egész számokat tartalmazó sorozatból meghatározza a legkisebb elem előfordulási helyeit. A sorozatot fix elemszámú tömb adatszerkezettel reprezentáljuk. A megoldás során az objektum-orientált programozás szabályai figyelmen kívül hagyhatóak.

Mindenek előtt érdemes felrajzolni a feladat fekete dobozát, meghatározni a bemeneteket és a kimeneteket. Egy olyan alkalmazásra van igény, mely egy sorozat bemenetre egy sorozat kimenettel válaszol. Természetesen hogyha az eredeti sorozatban a legkisebb elem csak egyszer fordul elő, a kimeneten csak ennek a helynek az indexe fog megjelenni, azonban ettől még egy egy elemű sorozatban fog mindez reprezentálódni. A 1.2. ábra egy általános esetet mutat be. Minden esetben érdemes egy egyszerű, fejben kiszámolható - a lehetőséghez mérten általános esetet - definiálni, de később majd látni fogjuk, a bemeneti lehetőségek variációinak az egész halmazát ellenőrizni szükséges.



1.2. ábra. Minimum helyek kiválogatása

1.2.1. Problématér felbontása

A feladatot nem tudjuk egyetlen programozási tétellel megoldani, azonban könnyen felbontható a probléma három ismert programozási tétel kombinációjára. Fontos most is figyelembe venni a C# nyelvi lehetőségeit. Az eredményt fix elemszámú tömbbel kell reprezentálni, azért mielőtt a kimeneti tömböt létrehozzuk, már ismertnek kell lennie ezen tömb méretének.

1. **Minimumkiválasztás tétele**[1] (sorozathoz-értéket rendelő) - Meghatározzuk a bemeneti sorozat legkisebb elemét (a példa sorozatára nézve ez kettőt fog eredményül adni).

2. **Megszámlálás tétele**[1] (sorozathoz-értéket rendelő) - Megszámoljuk, hogy összesen mennyiszer fordul elő a legkisebb elem a sorozatban (a példát folytatva, a kettes elem háromszor fordul elő).

3. **Kiválogatás tétele**[1] (sorozathoz-sorozatot rendelő) - Létrehozzuk a három elemű kimeneti tömböt, majd ide kiválogatjuk a legkisebb elemek indexeit (így elő fog állni a példában a [2, 4, 7] adatszerkezet).

1.2.2. Kliens kód

Mielőtt az implementáció részleteit kidolgoznánk, érdemes ezúton is legalább az általános eset viselkedését letesztelni, egy e célra létrehozott metódussal. Bár e kódsorok nagyon egyszerűek, mégis sokat adhatnak hozzá ahhoz, hogy programunk áttekinthető legyen, és azt implementáljuk le, ami valójában is a feladat. Már a munka elején meghatározza az *AssortmentMinimums()* metódus aláírását, kikényszerítve ennek alkalmazását.

```
1  public static void Test()
2  {
3    int[] data = { 7, 4, 2, 7, 2, 9, 8, 2 };
4    System.Console.WriteLine("input: " + ArrayToString(data));
5    int[] minimums = AssortmentMinimums(data);
6    System.Console.WriteLine("output: " + ArrayToString(minimums));
7  }
8
9  public static int[] AssortmentMinimums(int[] data)
10 {
11   return null;
12 }
```

1.4. kód. Feladat "fekete doboza"

Az implementáció kódolásának sorrendje ezt követően a programozó saját preferenciáján múlik. Választhatjuk azt az irányt, hogy az adott probléma részfeladatait különkülön implementáljuk, majd a végén ezen részeredményeket összefogjuk az *AssortmentMinimums()* metódusban, de kezdhetjük az implementációt az *AssortmentMinimums()* metódussal is, megvalósítva a részeredményeket előállító metódusok aláírásait, és majd ezt követően előállítva az egyes részfeladatok implementációját.

Bármelyik megoldást is választjuk, figyelembe kell vennünk, hogy jelenleg a problémát előzőleg már elemeztük, fejünkben megtalálhatóak az implementáció részletei, csupán a kódolás sorrendjén elmélkedünk. Lesznek olyan szituációk, amikor a részfeladatokra bontás nem lehetséges a munka legelején (ennek számos oka lehet, melyek közül pl. az emberi elme határa is ide sorolható), ilyen esetben célszerű nem ennyire szabadon megválasztani a kódolás irányát.

1.2.3. Tételek implementálása

```
1  public static int[] AssortmentMinimums(int[] data)
2  {
3    int[] result = null;
4    int min = MinimumSelection(data);
5    if (min != -1)
6    {
7      int count = CountingItem(data, min);
8      result = new int[count];
9      int index = 0;
```

```
10      for (int i = 0; i < data.Length; i++)
11      {
12        if (min == data[i])
13        {
14          result[index++] = i;
15        }
16      }
17    }
18    return result;
19  }
20
21  private static int MinimumSelection(int[] data)
22  {
23    return 2;
24  }
25
26  private static int CountingItem(int[] data, int item)
27  {
28    return 3;
29  }
```

1.5. kód. Kiválogatás tétele

A *minimum kiválasztás tételének* implementációjakor kihasználhatjuk azt, hogy a bemeneti sorozatban csak pozitív elemek lehetnek (feladat meghatározása szerint), ezért visszaadhatunk pl. -1-et (negatív számot) abban az esetben, ha a sorozatra nem értelmezett a legkisebb elem meghatározása (üres sorozatban nincsen szélsőséges elem). Mindennek ellenőrzését láthatjuk az 1.5. kód 5. sorában.

1.2.4. Tiszta metódusok

Eleddig nem lett külön megemlítve, de érdemes ejteni pár szót a metódusok méretéről és felelősségéről. A feladat megoldása szempontjából bőven elegendő lenne egy metódust elkészíteni, mégis ahogy azt az 1.5. kód mutatja, legalább három készült el. Jelen esetben a részfeladatok szerinti bontás adja ezt a fajta szeparációt, azonban még ennél is tovább lehet menni. Ideális esetben 4-5 soros utasításoknak már lehet egy nevet adni, mely az ő közös felelősségüket jelöli, ezáltal elhelyezhetőek egy e névvel ellátott metódusban. Mindez hosszú távon nagyon jól olvasható kódot eredményez, hiszen az egy metódusba zárt 4-5 programsorra egy jól meghatározott névvel hivatkozunk. Nagyon sok múlik természetesen a névadáson, egy rossz, vagy félrevezető név többet árt mint használ. A technika alkalmazásának elején gondolkodhatunk a következő módon is: ha van pár utasítás, mely mellé egy inline megjegyzést szeretnénk elhelyezni a forráskódban, annak érdekében hogy kicsit megmagyarázzuk miért is van itt ez a pár sor, inkább hozzunk létre egy metódust ezen megjegyzés szövegéből szülve egy értelmes metódus nevet.

Lényeges talán az is, hogy miként gondolunk egy-egy metódusra. A metódus, mint alprogram végrehajt valamilyen meghatározott feladatot. Ez továbbra is igaz, azonban ha nagyon sok 4-5 soros metódusunk lesz, sok olyan feladat fog "látszódni", ami minket nem annyira érdekel. Ez a gyakorlatban dezinformálja a kódot, elfedi a lényegi részt. Azonban ne féljünk ettől, hiszen egy objektum-orientált világban a metódusokat megfelelő felelősség szerint szét tudjuk válogatni, és ezen belül láthatósággal el tudjuk választani az alkalmazás szempontjából lényeges metódusokat (tipikusan *public* láthatóság), illetve a részfeladatok megoldásait (tipikusan *private* láthatóság). A nyilvános metódusoknak adjunk rövid(ebb) és könnyen megjegyezhető, egyértelmű nevet, a *private* metódusok nevei azonban lehetnek sokkal hosszabbak és akár "túlmagyarázóak" is.

```
1  private static int MinimumSelection(int[] data)
```

```
2  {
3    int min = -1;
4    if (data.Length > 0)
5    {
6      min = data[0];
7      for (int i = 1; i < data.Length; i++)
8      {
9        if (min > data[i])
10       {
11         min = data[i];
12       }
13     }
14   }
15   return min;
16 }
17
18 private static int CountingItem(int[] data, int item)
19 {
20   int count = 0;
21   for (int i = 0; i < data.Length; i++)
22   {
23     if (data[i] == item)
24     {
25       ++count;
26     }
27   }
28   return count;
29 }
```

1.6. kód. Minimum kiválasztás és megszámlálás tétele

1.2.5. Hatékonyság vizsgálata

Vizsgáljunk meg az elkészült implementáció gyakorlati hatékonyságát. Látható, hogy egyszerű utasításaink közül különösen nagy memóriát igénylő, illetve nagy processzor időt használó tételek nincsenek, ezért leegyszerűsíthetjük a problémát a ciklusmagok futásának számosságára. Egy N elemű sorozatot nézve a bemeneten, a megoldásunk *3N-1* ciklusmagot hajt végre (a minimum kiválasztás tétele az első elem elővizsgálata miatt *N-1* ciklusmagot futtat, de nagy N esetén ennek szerepe elenyésző lesz). Szerencsére számos ponton lehet javítani ezen a teljesítményen.

A legkisebb elem meghatározásához egyszer mindenképpen végig kell mennünk az egész sorozaton, hiszen akár az utolsó elem a sorozatban is lehet kisebb bármelyik másiknál. Eközben viszont egy apró módosítással egy lépésben azt is meg tudjuk határozni, hogy e legkisebb elem mennyiszer található meg a sorozatban[3].

```
1  private static int[] MinimumSelectionAndCounting(int[] data)
2  {
3    int[] result = new int[2];
4    int min = -1;
5    int count = 0;
6    if (data.Length > 0)
7    {
8      min = data[0];
9      count = 1;
10     for (int i = 1; i < data.Length; i++)
11     {
12       if (min > data[i])
```

[3] Az is lehetséges, hogy ugyanezen lépésben el is tároljuk ezen elemek indexeit, azonban - ha megfelelünk a feladat kiírásának, és csak fix elemszámú tömböket dolgozunk, ehhez egy legalább akkora puffer tömb létrehozása volna szükséges, amekkora a bemeneti sorozatunk, és számos új értékadás művelet is bekerülne a forráskódba. Mindez bár ciklusmagok számában előnyös volna, a költséges utasítások számát növelné.

```
13        {
14          min = data[i];
15          count = 1;
16        }
17        else if (min == data[i])
18        {
19          count++;
20        }
21      }
22    }
23    result[0] = min;
24    result[1] = count;
25    return result;
26  }
```

1.7. kód. Összevont kiválasztás és megszámlálás

A jegyzet e fejezetében az objektum-orientált programozás eszközkészletét, illetve a C# nyelvi lehetőségeit (Tuples) szándékosan kerüljük. A megoldás során csak elemi utasításokat és adatszerkezeteket használunk. Ennek egy következménye lett a *MinimumSelectionAndCounting()* metódus visszatérési értéke, mely egy tipikus rossz példának tekinthető a későbbiekben. A metódus két értékkel tér vissza, és e két értéket egy két elemű tömbbe zárja. A hívás helyén (lásd. 1.8. kód 4. sor) nem lesz meg az az információ, hogy mely érték van az első, és mely a második helyen. Programozás technikai eszköztárunk bővítése során egy ilyen feladatra más megoldást fogunk alkalmazni a későbbiekben.

A kiválogatás tétele "definíció" szerint egy számlálós ciklussal implementálható, hiszen a kiválogatandó elemek bárhol előfordulhatnak a sorozatban, annak minden elemére szükséges elvégezni a kiválasztás feltételét. A feladatban azonban már rendelkezünk egy nem elhanyagolható kiegészítő információval a kiválogatandó elemekről: mielőtt a kiválogatást elvégezzük, már rendelkezünk a kiválogatandó elemek számával! Vagyis bőven elegendő, ha addig válogatjuk ki az elemeket, amíg az utolsó hely is feltöltésre kerül az eredmény sorozatban (ezt követően lehet, hogy még számos elem van a tömbben, mi azonban tudjuk, hogy azok egyike sem fog megfelelni a kiválogatás feltételének, mivel előzőleg a megszámlálás tétele során már megvizsgáltuk az összes elemet).

```
1  public static int[] EffectiveAssortmentMinimums(int[] data)
2  {
3    int[] result = null;
4    int[] minAndCount = MinimumSelectionAndCounting(data);
5    if (minAndCount[0] != -1)
6    {
7      result = new int[minAndCount[1]];
8      int index = 0;
9      int i = 0;
10     while (index < result.Length)
11     {
12       if (minAndCount[0] == data[i])
13       {
14         result[index++] = i;
15       }
16       i++;
17     }
18   }
19   return result;
20 }
```

1.8. kód. Hatékonyabb kiválogatás

A hatékonyabb megoldása a feladatnak immáron egy N elemű sorozatot, mint bemenetet vizsgálva, $N+k$ ciklusmag végrehajtása után előáll ($k<N$), vagyis a változtatás átlagosan 50%-os javulást eredményezett.

1.3. Gyakorló feladatok

Számoljuk meg egy egész számokat tartalmazó sorozatban azon hárommal vagy öttel osztható számok számjegyeinek az összegét, melyek nem oszthatóak 15-el (pl.: bemenet: [2, 5, 12, 25, 7] kimenet: 15 (5 + 1 + 2 + 2 + 5))!

Gyerekek célba dobnak egy céltáblára, ahol 1-től 10-ig lehet pontokat szerezni. Mindenki négyszer dob a táblára, de nem minden esetben találja azt el. Egy fűrészfogas tömb segítségével modellezze le a gyerekek dobásait, és a dobások összege alapján határozza meg a dobogós gyerekeket (tömb indexeket).

Egész számokat tartalmazó sorozatból válogassuk ki azon értékeket, melyeknek prímtényezős felbontása legalább 3, de legfeljebb 5 tagú!

Sok évre visszamenőleg a hivatalos dáma versenyzők összeírják a nyertes verseny eredményeiket az alábbi módon: hány lépésben és mennyi perc alatt győztek. Minden sportolóhoz az évek során tartozik így egy két dimenziós sorozat (az ügyesebb versenyzőknek hosszabb sorozata van). A versenyzők adatait tartalmazó sorozatból határozza meg a leggyorsabb versenyzőt (aki átlagosan a legrövidebb idő alatt győzelemre vitte a partiait) illetve a legtaktikusabbat (aki átlagosan a legkevesebb lépésből győzelemre vitte a játékait). A két szélsőséges versenyző esetén határozza meg azon eredményeket, melyek esetén azonos a lépések száma!

Diákok matematika osztályzatait egy-egy sorozatban tároljuk el. Rendezze az eredményeket tartalmazó fűrészfogas tömböt átlag szerint növekvő sorrendbe. A rendezett sorozatból keresse ki azt a diákot, akinek az átlaga a legközelebb van az osztályátlaghoz!

2. fejezet

Véletlen szimulációk

Ismeretszerzés

bemeneti adat előállítása, valós körülmények szimulálása, felsorolt értékek listázása

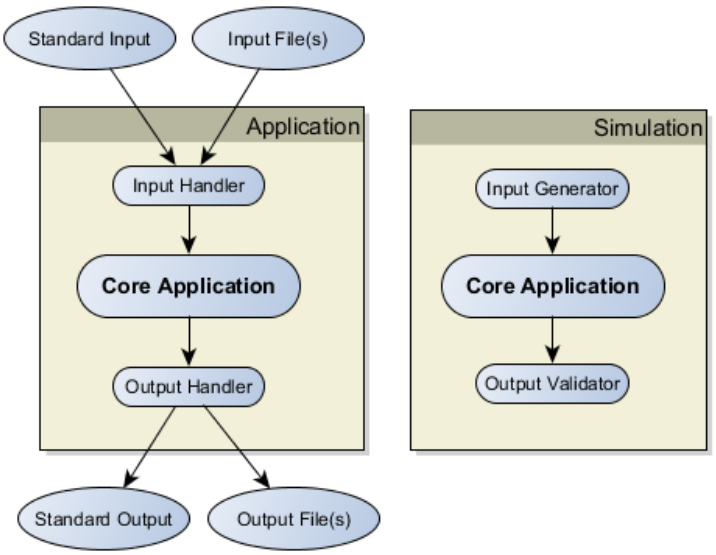
Vannak olyan kötelezőnek nevezhető kódsorok, melyeket szinte kivétel nélkül minden programozással foglalkozó szakember élete során legalább egyszer begépel. Ilyen a Kernighan és Ritchie szerzőpáros által híressé tett "hello, world" program[3], és ide tartozik valószínűleg a "Hogy hívnak?" és a "Mikor születtél? kérdést feltevő, majd pár perces gondolkodási idő után képernyőn "Szia Dávid, 32 éves vagy!" felirattal köszöntő programok csoportja is. Ezek talán a legegyszerűbb példák arra, hogy egy alkalmazás bemeneteivel és kimeneteivel dolgozzunk. Hamar rájön azonban a legtöbb ember, hogy később, ha pl. egy katalógus program algoritmusait szeretnénk elkészíteni, letesztelni, nem túl szerencsés a program indulása után néhány száz kérdéssel megtámadni a felhasználót (aki a fejlesztés során mi magunk leszünk), annak érdekében hogy pl. egy bináris keresést a bevitt személyek között elvégezzünk.

Adatokra azonban szüksége van egy alkalmazásnak, és ha ezt nem kapja meg a felhasználótól direktben, akkor más lehetőségek után szükséges néznünk. Ilyen megoldás lehet pl. az adatok állományból történő beolvasása, azonban sokszor még ez is sok időt elvesz a fejlesztésből (mindig foglalkozni kell a beolvasással és a hibakezeléssel is). Fontos szem előtt látnunk, hogy elsődleges célunk algoritmusok fejlesztése és tesztelése, és nem egy komplett dobozos termék gyártása. Egy-egy program bemenetét szimulálni a leggyorsabb és a legegyszerűbb, ami gyakorlatilag nem jelent mást, mint az eredeti alkalmazásunkat egy keretbe helyezni, és ezt a keretet egyszerűen literálokkal inicializálni.

Ha áttanulmányozzuk a 2.1. ábrát, még egy lényeges feltételt észrevehetünk, mely szükséges a szimuláció alkalmazásához: a program által kezelt bemeneteket és kimeneteket kezelő programrészeket külön kell szervezni. Egy objektum-orientált szemléletben készült alkalmazás esetén szerencsére mindez mondhatni természetes.

Ha szimulációt készítünk, akkor nem kell foglalkoznunk az adatbevitellel, és jobb esetben a végeredmény validációjával sem (bár ez már sokkal inkább egység tesztelési terület). A jegyzet az algoritmusok készítésének és tervezésének módszereivel és technikáival foglalkozik, és e szimulációs elvet veszi figyelembe a megoldások bemutatása során. Más szavakkal és kontextussal élve, minden feladatra adott megoldáshoz egy olyan API-t[1] készítünk, melyet egy másik fejlesztő önleíró módon használni tud, és általa eléri a programunk összes nyilvános funkcióját.

[1] Application Programming Interface



2.1. ábra. Szimuláció alkalmazása

Mi sem egyszerűbb tehát, ha egy programot el szeretnénk indítani, és annak a programnak szüksége van adatokra. Begépeljük őket egy keretalkalmazásba mint literál, és meghívjuk az eredetileg készített alkalmazásunk megfelelő műveleteit. Itt jön el a következő probléma: mi van, ha sok száz adatra van szükségünk ahhoz, hogy az alkalmazásunk viselkedését ellenőrizzük? Ilyen esetben adatot kell előállítanunk, pl. valamilyen pseudorandom generátor segítségével. A fejezetben bemutatott kódrészletek ilyen véletlen adatok generálásához nyújtanak segítséget.

2.1. Véletlen számok

Előfeltétel

System.Random osztály, felsorolás típusok, programozási tételek

Készítsünk alkalmazást, mely képes

- két egész szám között egy véletlen valós számot generálni,
- két egész szám között véletlen páros illetve páratlan számot generálni,
- egy véletlen ám egyedi egész számokat tartalmazó tömböt létrehozni.

Egy számítógép számára a nehezebb problémák közé tartozik visszaadni egy "véletlen" számot. Vannak erre megbízható algoritmusok, melyekkel jelen jegyzet nem foglalkozik, helyette megelégszik olyan pseudo-random generátorok alkalmazásával, mint pl. a .NET framework részeként elérhető *System.Random* osztály. Ha egy ilyen osztályt létrehozunk, akkor a létrehozás pillanatában inicializálódik (a megadott, avagy meg nem adott *seed* értékkel), és az ugyanabban a "pillanatban"[2] elkért véletlen számok mind azonosak lesznek. Több módszer létezik e hatás kikerülésére, mi azt fogjuk választani, hogy *Random* osztály példányából egy alkalmazáson belül minden esetben csak egyetlen egy fog létezni, és ezen példány referenciáját adjuk át minden olyan osztálynak vagy metódusnak, akinek szüksége van rá.

2.1.1. Véletlen valós szám

A feladat három önálló igényt fogalmaz meg, ezeket sorban egymás után készítsük el, miután a *Random* osztály lehetőségeivel megismerkedtünk. Valós számot generálni 0 és 1 között lehetséges, ahhoz, hogy két egész szám közé szorítsuk ezt az értéket, néhány matematikai művelet közé szükséges a generálást zárni, ahogy ezt a 2.1. kódrészlet mutatja.

```
1 public double GetRealNumber(Random random, int minValue, int maxValue)
2 {
3   return minValue + random.NextDouble() * (maxValue - minValue);
4 }
```

2.1. kód. Véletlen valós szám létrehozása

2.1.2. Véletlen paritás

A következő részfeladat véletlen páros és páratlan számok létrehozását irányozza elő. Készíthetünk két külön metódust is e célból, avagy felvehetünk egy felsorolást típust, mely

[2] Azonos "pillanatnak" tekinthetünk két kódsort, ha azok között néhány ezer alapvető, nem blokkoló C# utasítás fut le, ugyanis ezen utasításokat a számítógép azonos időegységben fogja használni.

segítségével vezéreljük a megírt metódust. Az utóbbira ad példát a 2.2. kódrészlet, mely kihasználja hogy minden páros szám 2K alakban, illetve minden páratlan 2K+1 alakban felírható.

```
1  public enum Parity
2  {
3    EVEN, ODD
4  }
5
6  public int GetNumber(Random random, Parity parity, int minValue, int maxValue)
7  {
8    int baseValue = random.Next(minValue / 2, maxValue / 2);
9    switch (parity)
10   {
11     case Parity.EVEN:
12       return baseValue * 2;
13     case Parity.ODD:
14     default:
15       return baseValue * 2 + 1;
16   }
17 }
```

2.2. kód. Véletlen páros/páratlan szám létrehozása

2.1.3. Véletlen egyedi elemek

A feladatkiírás utolsó részében egy véletlen egész számokat tartalmazó tömböt kell létrehoznunk, melyben minden elemnek egyedinek kell lennie. Minden új elem létrehozásakor a már generált elemeket tartalmazó résztömbön egy *eldöntés tételét*[1] szükséges implementálnunk. Erre ad példát a 2.3. kódrészlet.

```
1  public int[] GetUniqueNumbers(Random random, int size, int maxValue)
2  {
3    int[] data = new int[size];
4    for (int i = 0; i < size; i++)
5    {
6      int k;
7      int value;
8      do
9      {
10       value = random.Next(maxValue);
11       k = 0;
12       while (k < i && data[k] != value)
13       {
14         k++;
15       }
16     } while (k != i);
17     data[i] = value;
18   }
19   return data;
20 }
```

2.3. kód. Egyedi értékeket tartalmazó sorozat

A bemutatott metódusok csupán példák mindazon problémák szimulációjának megoldására, ahol valamilyen szabály szerint létrehozott véletlen számokra van szükségünk. A véletlen számokkal való játék nem jelenti azt, hogy nem lehet hatással lenni az eredményre. Gyakori példa lehet véletlen hegyvonulatok magassági metszetének előállítása, melyben meghatározott számú lejtőnek, lokális maximumok közötti távolságnak, stb. kell lennie annak érdekében, hogy egy-egy érdekesebb algoritmusnak az így létrehozott véletlen domborzati jelenség megfelelő bemenete lehessen.

2.2. Véletlen értékek és megnevezések

Előfeltétel

System.Random osztály, felsorolás típusok, String műveletek, StringBuilder

Készítsünk alkalmazást, mely képes

- egy véletlen autómárkát visszaadni az alábbiak közül: (ROVER, TOYOTA, SUZUKI, HONDA, FORD, SEAT, OPEL) illetve,
- egy véletlen fantázia személy nevet létrehozni (valós kereszt- és családnevek alkalmazása nem feltétel).

2.2.1. Véletlen felsorolt érték

A C# felsorolás típusa (*enum*) ún. *valuetype*, vagyis alapesetben a felsorolás egyes elemei azonosnak tekinthetőek egy-egy egész számmal. Bár mindez teljesítmény szempontból nagyon előnyös, megvannak a maga hátrányai is (pl. számhoz nem rendelt érték létezése). Véletlen értékek létrehozása szempontjából azonban könnyedén ki lehet használni azt, hogy bármikor oda-vissza alakíthatjuk az *enum* értékeket és egész számokat. Ha az *enum* értékek szám reprezentációja 0-tól kezdődő és nem tartalmaz lyukakat (vagyis egyesével emelkedik), akkor is szükségünk van arra, hogy programozottan kinyerjük a felsorolás típusban megtalálható értékek darabszámát (legalábbis amit a forráskódban felvettünk). Ennél általánosabb megoldást mutat be a 2.4. kódrészlet, melyben a visszaadott értékek tömbjéből választunk ki egy tetszőleges indexű elemet (ez esetben nem probléma, ha az *ordinal* értékek nem sorban követik egymást).

```
1  public enum CarBrand
2  {
3    ROVER, TOYOTA, SUZUKI, HONDA, FORD, SEAT, OPEL
4  }
5
6  [..]
7
8  public CarBrand GetCarBrand(Random random)
9  {
10   CarBrand[] values = (CarBrand[])Enum.GetValues(typeof(CarBrand));
11   return values[random.Next(values.Length)];
12 }
```

2.4. kód. Véletlen autó márka kiválasztása

2.2.2. Véletlen fantázia név

Egy fantázia név létrehozásához már kellenek apró ötletek. Véletlen karaktereket véletlen karakterkódok segítségével tudunk előállítani, vagyis ha megelégszünk az angol nagybetűs ábécé karaktereivel, akkor összerakhatunk véletlen karaktersorozatokat 65 és

90 között generált számok segítségével. Annak érdekében, hogy ne legyenek *mágikus számok* a kódban (mi a 65? mi a 90?), utóbbi számokat a nevezett karakterek számmá konvertálásával érdemes elhelyezni a forráskódban (lásd. 2.5. kódrészlet 16. sora).

Ha létrehozunk egy metódust, mely adott hosszúságú véletlen nagybetűs karakterláncot hoz létre, akkor egy vezeték- vagy keresztnév létrehozásához csupán egy nagy kezdőbetűre, és néhány kisbetűre lesz szükségünk[3].

```
1  public String GetName(Random random)
2  {
3    return GetNamePart(random, 8) + " " + GetNamePart(random, 8);
4  }
5
6  private String GetNamePart(Random random, int maximumLength)
7  {
8    return GetCharacters(random, 1) + GetCharacters(random, random.Next(3, maximumLength -
         1)).ToLower();
9  }
10
11 private String GetCharacters(Random random, int length)
12 {
13   StringBuilder builder = new StringBuilder();
14   for (int i = 0; i < length; i++)
15   {
16     builder.Append(Convert.ToChar(random.Next((int)'A', ((int)'Z' + 1))));
17   }
18   return builder.ToString();
19 }
```

2.5. kód. Fantázia nevek létrehozása

2.2.3. Véletlen paraméterezése

Mivel napjainkban nagyon népszerűek a három, vagy akár négy tagból álló nevek, egy apró kiegészítéssel megoldható, hogy pl. 5%-ban négytagú, 20%-ban pedig háromtagú neveket hozzunk létre (lásd. 2.6. kódrészlet).

```
1  public String GetName(Random random)
2  {
3    int chance = random.Next(0, 100);
4    return GetNameParts(random, (chance < 5 ? 4 : chance < 25 ? 3 : 2), 8);
5  }
6
7  private String GetNameParts(Random random, int size, int maximumLength)
8  {
9    StringBuilder name = new StringBuilder(size * maximumLength);
10   for (int i = 0; i < size - 1; i++)
11   {
12     name.Append(GetNamePart(random, maximumLength)).Append(" ");
13   }
14   name.Append(GetNamePart(random, maximumLength));
15   return name.ToString();
16 }
```

2.6. kód. Több tagból álló fantázia nevek létrehozása

[3] A jegyzet általában kerüli a nagyon C# specifikus megoldást, de pl. a CultureInfo.CurrentCulture.TextInfo.ToTitleCase(GetCharacters(random, random.Next(4, 8)).ToLower()) utasítás eredménye is hasonló eredményt produkálna.

2.3. Gyakorló feladatok

Egy sporttáborban a résztvevő gyerekek számára különféle sportágakban versenyeket szerveznek. Egy két dimenziós tömbbe írják fel az eredményeket, ahol az oszlopokban a diákok, a sorokban pedig az egyes versenyszámok "találhatóak". Minden versenyszámban az első 6 helyezés kap pontot, rendre 10, 8, 5, 3, 2 és 1 pontot. Hozzon létre egy ilyen versenytáblázatot véletlen adatokkal, majd határozza meg a legeredményesebb és a legtöbb érmet szerzett játékost!

Egy számítógépes játékban különféle harci gépezetek küzdenek egymással. Minden gépnek van egy támadási- és egy védekezési értéke, melyet egy 3 és 10 közötti valós szám reprezentál. Ha egyik gépezet megtámadja a másikat, akkor mindketten "dobnak" egy hat oldalú szabályos dobókockával, és a támadó a támadási-, míg a megtámadott a védekezési értékét szorozza meg a dobott értékkel. A nagyobb értéket dobó játékos győz. Hozzon létre véletlenszerű adatokkal harci gépezeteket, majd addig válasszon ki két gépet egymás elleni küzdelemre, míg végül csak egy marad.

3. fejezet

Grafikai lehetőségek Console ablakban

Ismeretszerzés

objektum-orientált illetve állapot-felelősség felbontás, entitás aggregációs kapcsolatok, karaktergrafika, kurzor és vezérlő billentyűk kezelése, karakter puffer, TDD alapok, adattagok függése és száma

Érdekes dolog a vizuális megjelenés és a programozás kapcsolata. Vitathatatlan tény, hogy egy-egy szebb vizuális élmény érdekében fantasztikus algoritmusok születtek az elmúlt évtizedekben. Sokszor nehéz bemutatni egy egyszerű algoritmus eredményét egy társaságnak, ha annak csak pár soros kimenete van (ettől még természetesen maga az alkalmazás megoldhatja a világ aktuális problémáinak a felét). Talán ennél is fontosabb az a felismerhető jelenség, miszerint a programozással ismerkedő személyeknél motivációs problémák léphetnek fel. Nem mindenki szemében gyújt lángot a terminál képernyőn megjelenő felirat: *"All tests passed"*[1]. Az algoritmusokkal való ismerkedés elején is igény van vizualizációra (itt most nem térünk ki arra az ágra, aki ezen a ponton áttér valamilyen játékfejlesztő keretrendszer használatára). Ahhoz, hogy eme igényt valamilyen formában kielégítsük, miközben nem veszítjük szem elől azt, hogy a hangsúly ismeretszerzésünk elején elsősorban az algoritmusokon van, nézzünk pár egyszerű példát arra, hogy miként tudjuk személyre szabni A .NET framework *Console* ablakát.

[1] Minden teszt sikeresen lefutott, átment az ellenőrzésen. Tipikusan egység tesztek futása után megjelenő összefoglaló üzenet.

3.1. Labirintus

Előfeltétel

osztályok definiálása, osztályrelációk, egységbezárás, szöveges állományok megnyitása, véletlen számok, felsorolás típusok

Készítsünk alkalmazást, mely szöveges állományból töltse be egy labirintus felülnézeti térképét, majd karaktergrafika segítségével jelenítse azt meg. Egy véletlenszerű, ám szabad (nem fal) pozíción helyezzünk el egy játékost, akit a kurzor billentyűk segítségével négy irányban lehessen mozgatni a folyosókon. A labirintus állományának első sorában pontosvesszővel elválasztva kiolvasható a felülnézeti térkép szélessége és magassága. A megoldás során a hibakezeléssel nem kell foglalkozni (állomány beolvasásakor). Az alkalmazás az objektum-orintált módszertanoknak feleljen meg. Az alkalmazásból kilépni az Esc billentyű leütésével lehessen.

```
30;12
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X X     X  X      X   X      X XX
X X X   X XX XXX X  XX XXX X  X
X XXXX X     X   XX    X   XX X
X X   X XX X XXX X X X XXX X  X
X           X  X      X  X    XX
X X     X  X      X   X      X XX
X X X   X XX XXX X  XX XXX X  X
X XXXX X     X   XX    X   XX X
X X   X XX X XXX X X X XXX X  X
X           X  X      X  X    XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

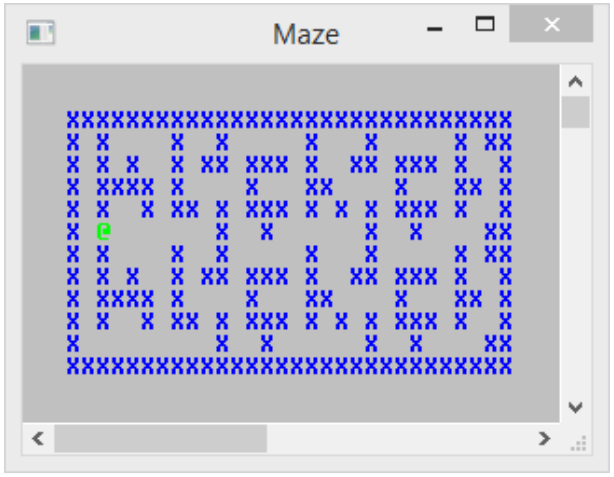
3.1. kód. Labirintus modelljének állománya

Fontos, hogy mielőtt az implementáció bármely részletén elkezdenénk gondolkodni, legyen egy pontos kép a szemünk előtt arról, hogyan is fog kinézni az elkészült alkalmazás. Ezt lerajzolhatjuk vagy egyszerűbb esetben elgondolhatjuk. A jegyzetben a 3.1. ábra egy valós képernyőképet mutat be a majdan futó alkalmazásról. A feladat megfogalmazása a problémateret defininálja, apró részletekre azonban nem tér ki. Le lehetett volna írni, hogy a labirintus a képernyő közepén jelenjen meg, a falakat *'X'* karakter jelölje és kékek legyenek, míg a játékost egy zöld *'@'* szimbólum vizualizálja. Mindez azonban a feladat megoldása során teljesen lényegtelen, és ha mindezt már az elején felismerjük, ezen "paraméterek" algoritmusokba "égetésével" nem fogunk foglalkozni.

Algoritmikus szemmel nézve a feladatot, nem igen található benne kihívást jelentő kódrészlet. Sokkal inkább fog szólni a megoldás arról, hogy egy egyszerű feladatot miként implementálunk egy kiválasztott felelősség szerinti objektum-orientált felbontásban, illetve a fejezet témája végett a karaktergrafikus megjelenítés lehetőségeiről.

3.1.1. Entitások számbavétele

Vizsgáljuk meg, milyen entitások fordulnak elő a leírásban, és ezek közül melyek azok, melyekhez egyértelmű felelősség rendelhető!



3.1. ábra. Labirintus képernyőkép

- **Labirintus** Szükségünk lesz egy modellre, mely reprezentál számunkra egy *felülnézeti térképet*, gondoskodik ennek beolvasásáról, megjelenítéséről, és határainak vizsgálatáról (hol van fal, hol nincs fal).
- **Játékos** Megjelenik egy *játékos* is a pályán, akinek aktuális pozíciója mellett mozgási szabadsága is van, természetesen figyelembe véve a labirintus által meghatározott folyosókat.
- **Játék** A *játék* egy adott pálya betöltésével indul, folytatódik egy *játékos* elhelyezésével majd mozgatásával. ESC hatására a játékból, és egyben az alkalmazásból is kilépünk.

Azok a névvel rendelkező elemek egy objektum-orientált környezetben, melyekhez adat és felelősség rendelhető, biztosan különálló típusként fognak megjelenni az implementáció során. E szabály egy minimálisan teljesülő feltétel. Nem korlátoz abban, hogy valamely entitásból több típus jöjjön létre, nem mondja azt sem, hogy nem lehetnek olyan típusok később az alkalmazásban, melyek a felsorolt elemek közül egyikhez sem tartoznak, viszont kiköti azt, hogy ezen elemek önállóan, és jól meghatározott módon el kell hogy különüljenek az alkalmazásban. Mindez olvasva bizonyosan egyértelmű, azonban vajon mennyien készítenék el ugyanezt a programot egy osztályban? Valószínűleg olyan megoldás is születhetne, mely egyetlen *Main()* metódusból állna[2]. Olyan alkalmazást készítsünk, melynek ha a forráskódját odaadjuk egy hasonló szakmai kompetenciákkal rendelkező ismerősünknek, az képes legyen a feladat leírását percek alatt előállítani[3].

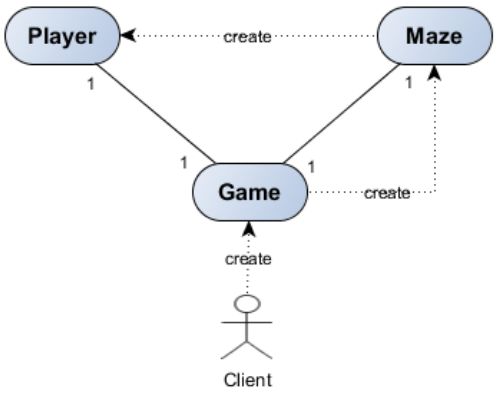
3.1.2. Osztálykapcsolatok felírása

Hasonló felbontással fogunk még találkozni a jegyzetben, ennél talán kevésbé egyértelmű szituációkban is. Amit ilyenkor még érdemes átgondolni, hogy milyen kapcsolatban vannak egymással a felsorolt entitások. A 3.2. ábra egy tervrajz (nem szabványos modell), melyet bátran felrajzolhatunk magunknak, és agilis kifejezéssel élve, bármilyen olyan jelölésmódot bevezethetünk rajta, mely egyértelmű a célközönség számára (érdekesség, hogy

[2] A feladatot meg lehet oldani körülbelül 50 programsorban C# nyelven, ami nagyszerű, ám kis túlzással semmi köze nincsen a programozáshoz.

[3] Remélhetőleg említeni sem szükséges, de a forráskódban semmilyen szöveges megjegyzés nem szerepelhet.

mindez akkor a legnehezebb, ha egyedül dolgozunk!). Leolvashatjuk (értsd. fejünkben lévő terveket végiggondolhatjuk) hogy egy *játék* példányt fogunk létrehozni, mely egységbe zár egy *labirintus* és egy *játékos* példányt (aggregáció). A *kliens*[4] számára teljesen elfedhetjük, hogy a háttérben milyen entitásokat definiáltunk (e miatt a *játékban* megtalálható aggregációk valójában már kompozíciónak is tekinthetőek). Létrehozunk egy *játékot* pl. egy X, Y koordináta pár (hol jelenjen meg a labirintus) és egy állomány név segítségével. Ezt követően a *játék* betölti a *labirintust*, majd ennek ismeretében már meg tud határozni egy olyan véletlen, ám szabad folyosó koordinátát, melyen a *játékkos* létre tud jönni (ezt szebben úgy mondhatjuk, hogy a *labirintus* lesz a *játékos* példányának *gyára*, vagyis *factory* osztálya.).



3.2. ábra. Labirintus entitásainak kapcsolata

3.1.3. Kliens kód

A 3.2. kódrészlet a *Program* elnevezésű osztályt mutatja be, mely egyetlen lényeges felelőssége a 10. sorban megtalálható *Game* példányosítása és elindítása a *Play()* metóduson keresztül. A *játék* osztály konstruktora bemeneti paraméterként megkapja a létrehozandó *labirintus* bal felső koordinátáját (3. sor, 2. oszlop), illetve a megnyitandó *labirintus* állomány elérési útját[5]. Mivel az alkalmazáson belül szükségünk lesz valamilyen véletlen szám generátorra, ezért ezt ezen a ponton hozzuk létre, és adjuk át a *játéknak*. Figyeljünk arra, hogy pseudo-random generátort mindig lehetőség szerint a főszálban és csak egy helyen hozzuk létre, ahogyan erről a 2. fejezetben már szó volt.

```
1 public class Program
2 {
3   private static readonly int WINDOW_WIDTH = 36;
4   private static readonly int WINDOW_HEIGHT = 16;
5
```

[4] Kliensnek tekinthetjük azt a fejlesztőt, aki az adott feladat számára elkészített alkalmazást egy forráskódban felhasználja. Ez a legtöbb esetben mi magunk vagyunk, akik egy egység vagy integrációs tesztből, avagy egy *Main()* metódusból létrehozzuk és elindítjuk az implementált algoritmusokat.

[5] A megadott elérési út relatív, a projekt futásakor az elindított alkalmazás könyvtárából nézve egy *maze* könyvtárban lévő *track-1.txt* állományt fog keresni a program. Ezt külső erőforrásként a projekthez csatolhatjuk Visual Studioban. A projekt gyökerében vegyünk fel egy *maze* könyvtárat, majd azon belül a nevezett file-t (vagy hozzuk létre kívülről, és az "Include In Project" paranccsal helyi menüből adjuk hozzá, miután a *Solution Explorerben* a "Show All Files" kapcsolót aktiváltuk.). A *track-1.txt* állomány tulajdonságai között (*Properties Window*) a *Build Action* értéke legyen "Content", míg a *Copy to Output Directory* értéke "Copy always" vagy "Copy if newer" legyen.

```
6   private static void Main(string[] args)
7   {
8     Console.SetWindowSize(Program.WINDOW_WIDTH, Program.WINDOW_HEIGHT);
9     Console.Title = "Maze";
10    new Game(new Random(), 3, 2, @"maze\track-1.txt").Play();
11  }
12 }
```

3.2. kód. Labirintus tesztelése

3.1.4. Labirintus

A *labirintus* modellezése alkalmazásunkban nem szól másról, mint a falak és a folyosók definiálásáról. Erre számos megoldás kínálkozik a különféle vektorgrafikus modellezéstől a ritka mátrixokig. Most válasszunk egy egyszerű és könnyen feldolgozható módszert: modellezzük egy két dimenziós logikai mátrixszal a *labirintust*, ahol igaz értéket a mátrix egy cellája ott vesz fel, ahol fal van. A C# támogatja külön típussal az NxM-es mátrixok létrehozását (valójában ez egy egy dimenziós tömb lesz a háttérben), használjuk ezt ki[6]. Ha olyan típust készítünk, mely a háttérben valamilyen sorozat adatszerkezetet tartalmaz, lehetőség szerint ezen adatszerkezet legyen végleges (C# szintaxis alapján ez a **readonly** kulcsszóval érhető el), és a konstruktorban hozzuk létre, átadott paraméterek által definiálva. Nem javasolt magát az adatszerkezetet átadni a konstruktornak, mert ezzel az osztály kívülről egyszerre túl nagy - általában kezeletlen - felelősséget kap. A *Maze* osztály ezek szerint megkapja a felülnézeti térkép szélességét és magasságát, melyet felhasználhat a modellje létrehozására. Mivel érték típusú alapeleme van a mátrixnak, létrehozás után minden elem folyosónak fog számítani (*false* érték). Gondolhatunk már ezen a ponton arra, hogy a mátrix megjelenítésekor szükségünk lesz arra, hogy hol helyezzük el a képernyőn, ezért a bal felső koordináta párt is átadhatjuk konstruktoron keresztül. Az osztály minden mezője végleges lesz, ami egyelőre nem cél, de későbbi fejezetekben erre majd külön törekedni fogunk.

A 3.3. forráskód a *Maze* osztály azon részletét mutatja be, melyben kivételkezelés használata nélkül megnyitjuk a *labirintus* állományát. A megnyitási művelet a *Maze* osztály (vagy egy jelen példában nem bemutatott *MazeFactory* osztály) felelőssége, ezért nem javasolt ezt más helyre szervezni a kódban. A *Load()* osztályszintű metódus visszatér a *labirintus* példányával, ezért a konstruktor láthatósága bátran lehet **private**.

Figyeljük meg, hogy a forráskód olvashatósága érdekében literálokat nem helyezünk el a kódban, ezért a 37. sorban a *WALL* konstanst, illetve a 50. sorban a *SEPARATOR* konstanst használtuk ezek helyett. Az algoritmusban teljesen lényegtelen, hogy az állományban *'X'* karakter jelöli a falat. Az algoritmus számára csak az a fontos, hogy amennyiben a soron következő beolvasott karakter *fal*, akkor állítsuk be a modellben is a fal helyét. Érdemes lehet a még a 51. sorban megtalálható szám literálok kivezetése is, itt azonban sokkal inkább az állomány feldolgozásának elegánsabb módjára lenne szükség (nem kellene eltárolni a file-ban a két értéket, dinamikusan is fel lehetne térképezni), azonban jelen alkalmazásban felülemelkedünk - a feladat leírásában szándékosan meghatározottak szerint - a filekezelés hibabiztos megvalósításának hiányán.

[6] Nem minden magas szintű nyelvben van erre beépített típus. Ez esetben használjunk tömb a tömbben adatszerkezetet (*fűrészfogas tömböt*), és figyeljünk arra, hogy minden sorban azonos számú oszlop legyen, illetve megvan a lehetőségünk természetesen arra is, hogy egy N*M elemű egy dimenziós tömböt egy külön típusba elcsomagolunk, és definiálunk a típusnak egy (n, m) paraméteres lekérő metódust, melyben valójában az (n*M+m) indexű elemet adjuk vissza.

```
1  public class Maze
2  {
3    private const char SEPARATOR = ';';
4    private const char WALL = 'X';
5
6    private readonly int left;
7    private readonly int top;
8    private readonly bool[,] fields;
9
10   private Maze(int left, int top, int width, int height)
11   {
12     this.left = left;
13     this.top = top;
14     this.fields = new bool[height, width];
15   }
16
17   private void SetWall(int row, int column)
18   {
19     this.fields[row, column] = true;
20   }
21
22   public bool IsWall(int row, int column)
23   {
24     return this.fields[row, column]; // !!!
25   }
26
27   public static Maze Load(int left, int top, String fileName)
28   {
29     StreamReader reader = new StreamReader(File.Open(fileName, FileMode.Open));
30     Maze maze = CreateMaze(left, top, reader.ReadLine());
31     String line = "";
32     int row = 0;
33     while ((line = reader.ReadLine()) != null)
34     {
35       for (int column = 0; column < line.Length; column++)
36       {
37         if (line[column] == WALL)
38         {
39           maze.SetWall(row, column);
40         }
41       }
42       row++;
43     }
44     reader.Close();
45     return maze;
46   }
47
48   private static Maze CreateMaze(int left, int top, String coordinateLine)
49   {
50     String[] coords = coordinateLine.Split(SEPARATOR);
51     return new Maze(left, top, Int32.Parse(coords[0]), Int32.Parse(coords[1]));
52   }
53
54 }
```

3.3. kód. Labirintus betöltése

A 3.3. forráskód 22. sorában létrehoztunk egy korábban még nem használt metódust *IsWall()* néven, mert várhatóan igény lesz rá. Bár sokan nem így járnak el fejlesztés közben, meg kell jegyezni, hogy lehetőleg kerülni kell az "előredoglozást", vagyis pl. ne készítsük el azokat az *accessor* vagy *mutator* elemeket[7], melyekre aktuálisan még nincsen szükség (hiába csábít ezen metódusok elkészítésének egyszerűsége (értsd. gép is pillanatok alatt legenerálja), egy alkalmazást sosem a begépelt karakterek számával értékelünk). Visszatérve az *IsWall()* metódusra, a forráskódban lévő felkiáltójelek arra utalnak, hogy a metódust a későbbiekben módosítani fogjuk, ez csak az első verziója.

[7] Accessor elemek tipikusan a *getXY()* metódusok, illetve az XY *property*-k **get** blokkja, *mutatornak* pedig a *setXY()* metódusokat, illetve az XY *property*-k **set** blokkját hívjuk. Nem feltétlenül, de ezek legtöbbször csak olvasnak, illetve írnak egy mezőt az adott példányban.

3.1.5. Játékos

Ideje létrehozni a *játékos* osztályát, melynek példánya a *labirintus* egy adott sorának adott oszlopában található. Ezen mezői az osztálynak értelemszerűen *mutable* elemek, vagyis változtathatóak. Ne készítsünk hozzájuk *accessor* és *mutator* elemeket! Látni fogjuk, semmi szükség nem lesz rájuk. Nem az a célja az osztálynak, hogy vissza tudjon adni egy koordináta párt! Gondolkodjunk el azon egy percet, mire is van szükségünk akkor, amikor egy *játékost* felhasználunk az alkalmazásban: mozgatni szeretnénk négy irányban, illetve értelemszerűen megjelenítenénk a térképen az aktuális pozícióján. Magyarán szükségünk lesz egy *Move()* és egy *Draw()* metódusra. Utóbbi láthatóságát pedig hamar áttehetjük **private**-ra, mivel csak akkor szükséges kirajzolni a figurát, ha elmozdult, így a mozdulás fogja kiváltani az újrarajzolását.

A mozgás metódusának vezéréléséhez szükségünk lesz egy olyan adatszerkezetre, melyet átadva egyértelműsítjük, hogy a négy irány melyikébe szeretnénk elmozdítani a *játékost*. E problémára remekül megfelel a C# *enum* érték típusa, melyet önálló top-level elemként definiáljunk (lásd. 3.4. kód).

```
1 public enum Direction
2 {
3   LEFT,
4   RIGHT,
5   UP,
6   DOWN
7 }
```

3.4. kód. Irányok érték típusa

Szükségünk van még pár apró ötletre az implementációhoz. Hogyan fogjuk a "mozgást" szimulálni? Adott koordinátán adott színnel a *játékos* '@' jele látszik a képernyőn. Ha el tudunk mozdulni pl. balra (mert az aktuális helyzetet alapul véve a *labirintusban* szabad út van abba az irányba), akkor mielőtt megváltoztatjuk a példány állapotát, "kirajzoljuk" az eredeti pozíciójában a *labirintus* háttér-, majd a változtatás után a saját színével. Más megoldás is lehetséges (pl. szóközt is rajzolhatnánk "törlés" hatást kiváltva). A *játékos* mozgatásához szükségünk van a *labirintus* példányára, hiszen ezen keresztül tudjuk ellenőrizni azt, hogy ne ütközzünk falba. A jelenlegi aggregatív kapcsolatokat figyelembe véve, a *labirintus* példányát át kell adnunk a *Move()* metódusnak (lásd. 3.5).

```
1  public class Player
2  {
3    private const char SHAPE = '@';
4
5    private readonly ConsoleColor background;
6    private int row;
7    private int column;
8
9    public Player(ConsoleColor background, int row, int column)
10   {
11     this.background = background;
12     this.row = row;
13     this.column = column;
14   }
15
16   public void Move( Maze maze, Direction direction )
17   {
18     this.Draw(this.background, maze.Left, maze.Top);
19     switch (direction)
20     {
21       case Direction.LEFT:
22         if (!maze.IsWall(this.row, this.column - 1))
23         {
```

```
24            this.column--;
25          }
26          break;
27        case Direction.RIGHT:
28          if (!maze.IsWall(this.row, this.column + 1))
29          {
30            this.column++;
31          }
32          break;
33        case Direction.UP:
34          if (!maze.IsWall(this.row - 1, this.column))
35          {
36            this.row--;
37          }
38          break;
39        case Direction.DOWN:
40          if (!maze.IsWall(this.row + 1, this.column))
41          {
42            this.row++;
43          }
44          break;
45      }
46      this.Draw(ConsoleColor.Green, maze.Left, maze.Top);
47    }
48
49    private void Draw( ConsoleColor color, int left, int top )
50    {
51      Console.ForegroundColor = color;
52      Console.SetCursorPosition(left + this.column, top + this.row);
53      Console.Write(SHAPE);
54    }
55
56  }
```

3.5. kód. Játékos modellje

3.1.6. Labirintus átdolgozása

Ezen a ponton szükségessé válik a *Maze* osztály *IsWall()* metódusának módosítása. Ha megfigyeljünk a *Move()* metódusban az említett alprogram hívásait, feltűnhet, hogy ha a *labirintus* szélén vagyunk, a paraméterben átadott koordináták kiesnek a címezhető tartományból (és ez futás idejű kivételt eredményez). Ez egy tipikus szituáció, és a későbbiekben erre a kivételkezelés remek megoldást nyújthat, azonban most előre ellenőrizzük a tartomány helyességét. Eme ellenőrzésnek az *IsWall() accessor* metódus adhat helyet, egyszerűen visszaadva falat (**false** értéket), amennyiben a tartomány már nem értelmezett (lásd. 3.6. kód 28-29. sor).

Az *IsWall()* metódusra szükségünk lesz ahhoz is, hogy egy *játékost* létrehozzunk. Az eredeti gondolatmenetet követve, a létrehozás felelőssége a *labirintus* példányában lesz, lévén a már betöltött felülnézeti térkép tud visszaadni egy véletlen, ámbár szabad folyosó koordinátát a *játékos* számára.

```
1  public class Maze
2  {
3    [..]
4
5    private int Width
6    {
7      get { return this.fields.GetLength(1); }
8    }
9
10   private int Height
11   {
12     get { return this.fields.GetLength(0); }
13   }
```

```
14
15    public Player CreatePlayer( Random random, ConsoleColor background )
16    {
17      int row = 0;
18      int column = 0;
19      do {
20        row = random.Next(this.Height);
21        column = random.Next(this.Width);
22      } while (IsWall(row, column));
23      return new Player(background, row, column);
24    }
25
26    public bool IsWall(int row, int column)
27    {
28      bool valid = this.isValid(row, column);
29      return !valid || (valid && this.fields[row, column]);
30    }
31
32    private bool isValid(int row, int column)
33    {
34      return this.IsValidRow(row) && this.IsValidColumn(column);
35    }
36
37    private bool IsValidRow(int row)
38    {
39      return row >= 0 && row < this.Height;
40    }
41
42    private bool IsValidColumn(int column)
43    {
44      return column >= 0 && column < this.Width;
45    }
46
47    [..]
48
49  }
```

3.6. kód. Játékos létrehozása

A tartomány ellenőrzése nem egy bonyolult matematikai művelet, azonban mégis öt egysoros metódus keletkezett e célból a 3.6. kódot áttekintve (kettő ezek közül **get** *accessor* metódusa C# *property*-knek). Gondolkozzunk el egy kicsit azon, mennyi időbe telne értelmezni ugyanezen validációt (illetve mennyi idő lenne hibát javítani ebben), ha csupán egyetlen metódust készítünk ugyanebből a célból, ahogyan ez a 3.7. kódrészletben látható.

```
1  public bool IsValid(int row, int column)
2  {
3    return row >= 0 && row < this.fields.GetLength(0) && column >= 0 && column <
         this.fields.GetLength(1);
4  }
```

3.7. kód. Fal ellenőrzésének alternatívája

A *labirintus* kirajzolása egy egyszerű mátrix bejárás műveletével ér fel (lásd 3.8). Minden egyes cellát megvizsgálunk, és ahol fallal találkozunk a modellben, oda falat, minden más helyre pedig szóközt rajzolunk. E kirajzolás során karaktergrafikus pozíciót mindig csak az új sor elején változtatunk (*SetCursorPosition()* metódus), azonban mindent felülírunk, ami eredetileg a képernyőn volt korábban (a szóközök miatt). Ha egy hasonló problématérben ez nem előnyös, akkor csak a falakat rajzoljuk ki, és a pozíció állítást minden falrajzolás előtt eszközöljük.

```
1  public class Maze
2  {
3    [..]
4
```

```
5   public int Left
6   {
7     get { return this.left; }
8   }
9
10  public int Top
11  {
12    get { return this.top; }
13  }
14
15  public void Show()
16  {
17    Console.ForegroundColor = ConsoleColor.Blue;
18    for (int row = 0; row < this.Height; row++)
19    {
20      Console.SetCursorPosition(this.left, this.top + row);
21      for (int column = 0; column < this.Width; column++)
22      {
23        Console.Write(this.IsWall(row, column) ? WALL : ' ');
24      }
25    }
26  }
27
28  [..]
29
30 }
```

3.8. kód. Labirintus megjelenítése

3.1.7. Játék

Miután minden építőkövet megalkottunk, utolsó lépésben a *játék* vezérlését szükséges elkészítenünk. Ez a felelősség a *Game* osztály *Play()* metódusára tartozik, és az alábbi lépésekből áll:

- Képernyő törlése háttérszínnel
- *Labirintus* megjelenítése a kívánt pozíción
- *Játékos* megjelenítése
- Egy ciklusban a leütött kurzor karakterek figyelése és lekezelése addig, amíg az ESC billentyű lenyomásra nem kerül

```
1  public class Game
2  {
3    private static readonly ConsoleColor BACKGROUND = ConsoleColor.Gray;
4
5    private readonly int left;
6    private readonly int top;
7    private readonly Maze maze;
8    private readonly Player player;
9
10   public Game(Random random, int left, int top, String mazeFileName)
11   {
12     this.left = left;
13     this.top = top;
14     this.maze = Maze.Load(left, top, mazeFileName);
15     this.player = this.maze.CreatePlayer(random, BACKGROUND);
16   }
17
18   public void Play()
19   {
20     Console.BackgroundColor = BACKGROUND;
21     Console.Clear();
```

```
22      this.maze.Show();
23      this.player.Move(this.maze, Direction.LEFT);
24      ConsoleKey key;
25      do
26      {
27        key = Console.ReadKey(true).Key;
28        switch (key)
29        {
30          case ConsoleKey.LeftArrow:
31            this.player.Move(this.maze, Direction.LEFT);
32            break;
33          case ConsoleKey.RightArrow:
34            this.player.Move(this.maze, Direction.RIGHT);
35            break;
36          case ConsoleKey.UpArrow:
37            this.player.Move(this.maze, Direction.UP);
38            break;
39          case ConsoleKey.DownArrow:
40            this.player.Move(this.maze, Direction.DOWN);
41            break;
42        }
43      } while (key != ConsoleKey.Escape);
44    }
45
46  }
```

3.9. kód. Játék összeépítése

A *labirintus* "játék" elkészítése nem tartalmazott olyan algoritmusokat, melyekhez komolyabb programozási tételeket lett volna szükséges felidéznünk. Egyszerű vezérlési szerkezetek, mátrix bejárás, egy-egy hátultesztelős ciklusra jó példa. Annak, aki erre fogékony, megláthatja a karaktergrafikában rejtőző lehetőségeket (néhány ellenség, egy kis szálkezelés, és már kész is a *Wizard of Wor*[7] klasszikusa), de még ennél is fontosabb talán átnézni az elkészült osztályok egyszerűségét és felelősségét.

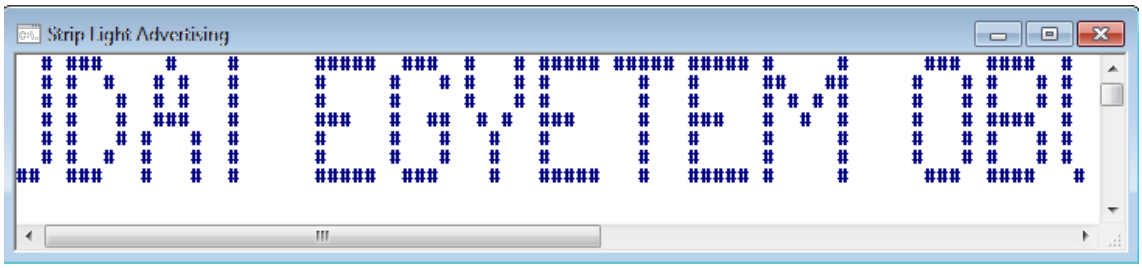
3.2. Fényreklám

Előfeltétel

osztályrelációk, egységbezárás, többalakúság, konstruktor hívási lánc, tömb literálok

Készítsünk karaktergrafika segítségével olyan "fényreklámot", mely a megadott szöveget hét egység magasan "felnagyítva" megjeleníti, majd vízszintes irányban körkörösen mozgatja. A balról eltűnő "pixelek" jobb oldalt azonnal megjelennek. A karakterek mindegyike 7 egység magas legyen, de szélességük típusonként eltérhet (pl. az I betű 3, míg az M betű 7 széles lehet). A fényreklám mozgatása "pixelenként" történik, nem karakterenként ("pixel" alatt értve egy pontját a karaktergrafikus képernyőnek)! A program az ESC billentyű leütésének hatására fejeződjön be. Az implementáció során olyan objektum-orientált megoldást válasszon, mely a fényreklám aktuális megjelenítését "mátrixnyomtató" elven jeleníti meg a képernyőn (sorfolytonosan, a kurzor pozícióját direktben nem módosítva).

Vannak olyan feladatok, melyek egy-egy példa ábra nélkül még sokadik elolvasásra sem tűnnek egyértelműnek. Ez természetesen (sokszor) visszavezethető a feladatot specifikáló személy gyengébb képességeire, azonban mindig megéri rajzolni pár ábrát, majd azt egyeztetni a "megrendelővel", mielőtt a tervezés és implementálás megkezdődik. Vizualizálni egy-egy problémát vagy annak felbontását ugyanolyan mérnöki feladat, mint megtervezni annak implementációs lépéseit.



3.3. ábra. Fényreklám

A 3.3. ábra ezúton is a már elkészült programból mutat egy képernyőképet, ahol az "OBUDAI EGYETEM" szöveg már éppen az U betű felénél tart a "mozgásban". Teljesen önkényes módon - az eredeti feladatot üzletileg nem módosítva - megállapodhatunk néhány kiegészítő feltételben, mely a választott implementációra hatással lesz:

- A karakterek modelljét két-dimenziós *byte* tömbökben fogjuk leírni, literálként, és két-dimenziós logikai tömbökben fogjuk tárolni (*byte* tömb literál kevesebb helyet foglal, illetve átláthatóbb mint a logikai tömb literál).
- Nem minden karaktert fogunk lemodellezni, csupán néhányat, melyből már az alkalmazás működése teljes egészében ellenőrizhető.
- A reklám háttér- és előtérszínét paraméterezhetővé tesszük, illetve a futó reklámszöveg mindig a képernyő bal felső sarkában fog megjelenni.

A programtervezési módszerek között tucatjával találunk olyan szabályokat, javaslatokat, melyek egymásnak akár ellentmondásai is lehetnek. Ilyen ezek közül az implementáció megtervezése a kódolási munka előtt, illetve ezen tervezés részleges vagy egészét érintő mellőzése. Most az utóbbinak alapjaival ismerkedünk meg, és szándékosan nem foglalkozunk azzal, hogy milyen osztályokra és felelősségekre lesz a későbbiekben szükségünk.

3.2.1. Karakterek modellezése

A karakterek modellezésével kezdjük a munkát. Megállapodás szerint literálként fogjuk leírni a karakterek "pixeleit" egy két-dimenziós *byte* tömb segítségével (és nem pl. állományból olvassuk be). A *byte* tömb azért célszerű megoldás, mert a forráskódban a tömb literál felveheti azt az alakot, ami hasonlít a karakter majdani futás idejű vizuális képére (lásd. 3.10. kódrészletet, mely egy 'T' betű alakját veszi fel).

```
1 new byte[][]{
2  new byte[]{ 1, 1, 1, 1, 1 },
3  new byte[]{ 0, 0, 1, 0, 0 },
4  new byte[]{ 0, 0, 1, 0, 0 },
5  new byte[]{ 0, 0, 1, 0, 0 },
6  new byte[]{ 0, 0, 1, 0, 0 },
7  new byte[]{ 0, 0, 1, 0, 0 },
8  new byte[]{ 0, 0, 1, 0, 0 } };
```

3.10. kód. T betű leírása literálként

Az így leírt modellt azonban logikai tömbként érdemes az alkalmazásban kezelni (ezzel burkoltan elérve azt, hogy lényegtelen legyen az, hogy a karakter "képe" milyen forrásból származik), célszerű ehhez majd valamilyen átalakítót készíteni. Mielőtt azonban ennek részletit kidolgoznánk, nézzük meg hogy miként szeretnénk az alkalmazásunkat használni? Célszerűen megadunk egy karakterláncot a reklám létrehozásakor, majd megjelenítjük azt (esetleg elindítjuk). Mindezen problémát persze visszavezetjük egy karakter létrehozására, mely során e karaktert adjuk át egy olyan konstruktornak, mely a szükséges modellt létrehozza számunkra. E felhasználási tervre mutat példát a 3.11. kódrészlet.

```
1 Advertisement ad = new Advertisement("OBUDAI EGYETEM");
2 Console.WriteLine(ad);
3 [..]
4 ad.Play();
5 [..]
6 CharacterModel letterA = new CharacterModel('A');
7 Console.WriteLine(letterA);
8 [..]
```

3.11. kód. Felhasználsi terv

3.2.2. Karakterek csomagoló osztálya

A karakterek már le tudjuk írni egy *byte* tömb segítségével, azonban célszerűen nem egy nagy *switch case* blokkban kellene a literál és a modell összerendelését végrehajtani (mivel minden *switch case* ág szerkezetileg ugyanazt a kódot tartalmazná, mely redundancia, és az objektum-orientált szemlélet szerint minden redundancia kerülendő). Hozzunk létre egy olyan osztályt, mely párosítja a karaktert (*char* típus) és a modelljét (*byte[][]* típus). Erre mutat egy példát a 3.12. kódrészletben bemutatott *CharacterData* osztály. Az osztály példányai *immutable*-ök, és egységbe zárják az említett két-dimenziós *byte* tömböt

és a karaktert (9. és 10. kódsor). Figyeljük meg, hogy az osztálynak **private** konstruktora van, és példányokat csak és kizárólag osztályszintű végleges mezők inicializálásakor (**static** és **readonly** módosítóval ellátott mezők, lásd. 3.12. kódrészlet 5. és 6. sora), az osztályon belül hozunk létre. Milyen típus leírására lehetnek jellemzőek ugyanezen tulajdonságok? Az *enum* típus valami nagyon hasonlót valósít meg, mindamellett természetesen, hogy C# nyelven az *enum valuetype*, és ennyire nem tud zárt lenni, és ennyi felelősséget sem tud tartalmazni, mint az itt megvalósított *CharacterData* osztály[8].

```
1  public class CharacterData
2  {
3    public const byte HEIGHT = 7;
4
5    public static readonly CharacterData A_MODEL = new CharacterData('A', [..]);
6    public static readonly CharacterData T_MODEL = new CharacterData('T', [..]);
7    [..]
8
9    private readonly byte[][] data;
10   private readonly char letter;
11
12   public bool[][] Model
13   {
14     get
15     {
16       bool[][] result = new bool[this.data.Length][];
17       for (int i = 0; i < this.data.Length; i++)
18       {
19         result[i] = new bool[this.data[i].Length];
20         for (int k = 0; k < this.data[i].Length; k++)
21         {
22           result[i][k] = this.data[i][k] == 1;
23         }
24       }
25       return result;
26     }
27   }
28
29   public char Letter
30   {
31     get { return this.letter; }
32   }
33
34   private CharacterData(char letter, byte[][] data)
35   {
36     this.letter = letter;
37     this.data = data;
38   }
39 }
```

3.12. kód. Karakterek adatok

Egészítsük ki a 3.12. kódrészletet azon karakterek létrehozásával, melyek modellezését szeretnénk megvalósítani (pl. az 'OBUDAI EGYETEM' felirat elkészítéséhez szükséges karaktereket gyártsuk le). A kódrészlet 5. és 6. sorában a [..] részlet helyére értelemszerűen a két-dimenziós *byte* tömbök literálja kerül, míg a 7. sorban jelzett [..] szimbólum helyére a még megvalósítandó karakterek leírásait írhatjuk be. Figyeljünk arra, hogy egy karakternek a feladat leírása értelmében 7 "pixel" magasnak kell lennie (ezt kivezethetjük egy konstansba), azonban a szélességre nincs ilyen korlátozásunk.

A *CharacterData* osztálynak van két nyilvános tulajdonsága is, melyek közül a *Model* elnevezésű valósítja meg a tárolt *byte* tömb és a célszerű logikai tömb közötti konverziót.

[8] Viszont ha a C# *enum* típusa mellett megismerkedünk pl. a Java *enum* típusával, akkor látni fogjuk, hogy utóbbi esetében pontosan egy olyan szintaktikai cukorkáról van szó, mint amit mi magunk a *CharacterData* osztályban kézzel megvalósítottunk (a felsorolás típusokra az 5.1 fejezetben még vissza fogunk térni).

3.2.3. Karakter törzs

Most, hogy előállítottunk egy olyan osztály (és példányokat!), melynek felelőssége egységbe zárni a karaktert és a hozzá tartozó karaktergrafikus modellt, hozzunk létre egy tömböt, melybe ezen osztály példányait elhelyezzük. A sorozaton végrehajtott *lineáris keresés*[1] segítségével fogjuk tudni kiváltani a korábban említett *switch case* által okozott redundanciát. Ezen a ponton igény keletkezett egy új mezőre (*CharacterData[]*) és az ezen végrehajtott felelősségre (lineáris keresés): ez egy új osztály létrehozásáért kiállt! Ezt mutatja be a 3.13. kódrészlet. Nem kellene szükségszerűen osztály szinten definiálni a mezőt és a műveletet (sőt, mindez egység tesztelési szempontból nem előnyös), az ún. *Singleton design pattern* ("egyke tervezési minta") pontosan ilyen célra lenne ideális. Minderről azonban a 7.1. fejezetben még szót ejtünk.

```
1  public class CharacterTrunk
2  {
3    private static readonly CharacterData[] CHARACTERS = new CharacterData[]{
4      CharacterData.A_MODEL,
5      CharacterData.B_MODEL,
6      CharacterData.D_MODEL,
7      CharacterData.O_MODEL,
8      CharacterData.I_MODEL,
9      CharacterData.U_MODEL,
10     CharacterData.E_MODEL,
11     CharacterData.G_MODEL,
12     CharacterData.M_MODEL,
13     CharacterData.SPACE_MODEL,
14     CharacterData.T_MODEL,
15     CharacterData.Y_MODEL};
16
17   public static bool[][] GetModelContent(char letter)
18   {
19     bool find = false;
20     int i = 0;
21     while (i < CHARACTERS.Length && !find)
22     {
23       find = (CHARACTERS[i].Letter == letter);
24       i++;
25     }
26     return CHARACTERS[i - 1].Model;
27   }
28
29 }
```

3.13. kód. Karakter törzs

3.2.4. Karakter modell

Minden adott ahhoz, hogy egy karakter megadásával létre tudjuk hozni a saját *CharacterModel* példányunkat, melyet a tervezett felhasználás során már említettünk. Készítsük el ezen - még nem létező osztály - felhasználásának kódsorait (lásd. 3.14. kódrészlet). Egység tesztnek mindez természetesen még nem nevezhető.

```
1  private static void TestCharacterModel(Random random)
2  {
3    CharacterModel a = new CharacterModel('A');
4    Console.WriteLine(a);
5    Console.WriteLine(a.GetRow(0));
6    Console.WriteLine(a.GetRow(1));
7    Console.WriteLine(a.GetRow(2));
8    CharacterModel b = new CharacterModel('B');
9    Console.WriteLine(b);
10   CharacterModel randomModel = CharacterModel.BuildRandomModel(random, 3, 6);
11   Console.WriteLine(randomModel);
```

```
12 }
```

3.14. kód. Karakter modell tesztelése

Megfogalmazásra került egy *GetRow(index)* metódus, melynek felelőssége a karakter modellje megadott sorának visszaadása. Az eredeti feladatkiírásban az szerepel, hogy úgy kell megjeleníteni a karaktereket, ahogyan a mátrixnyomtató "nyomtatja" a sorokat. Vagyis szükségünk lesz arra, hogy pl. az 'OBUDAI EGYETEM' legfelső "sorának" a "pixeleit" visszaadjuk. Ehhez a problémát visszavezetjük egy karakter (pl. az 'O') legfelső sorának visszaadására. A *BuildRandomModel()* osztályszintű metódus nem az eredeti feladatkiírás része, pusztán egy apró játék a lehetőségekkel. Figyeljük meg az osztályban megtalálható két konstruktor szerepeit. A *bool[][]* tömböt fogadó konstruktor láthatósága **private**, mivel egy ilyen felelősséget nem javasolt kifelé kinyitni (Hiába végleges a mező, *null* értékkel meghívható lenne. Mivel a láthatósága miatt a konstruktor az osztály belügye maradt, ezért nem kell félni attól, hogy felügyelet nélkül ezen a ponton "megtámadják" az osztályunkat.). A másik konstruktor nyilvános, és az elvárt *char* típust várja paraméterül. Mivel egy osztály egy felelősség (az objektum-orientáltság egyik fontos szabálya), ezért mindig (szinte mindig) törekedni kell arra, hogy az *overload* konstruktorok egymást hívják (ha egy adott osztályt kétféleképpen is létre lehet hozni, és ezen létrehozásokat nem lehet közös tőre vezetni, akkor gyanús hogy az osztályban legalább kettő különböző felelősség bújt el). Így van ez a bemutatott kódban is, ahol 16. sorban deklarált nyilvános konstruktor a 17. sorban (első utasításaként) áthív a 21. sorban deklarált *private* konstruktorba. A *content* mező értékadás művelete egyetlen helyen fog csak szerepelni a kódban, mely redundancia mentesebb megoldás, mintha ezen értékadás a nyilvános konstruktorban is megvolna (*this.content = CharacterTrunk.GetModelContent(type);*).

Az osztály *ToString()* metódusára validáció végett van szükség (a végleges alkalmazásban egy karaktert így nem kell megjelenítenünk). Addig nem szabad az alkalmazás fejlesztésében tovább lépni, amíg az eleddig megvalósított programrészek nem működnek az elvártnak megfelelően. Itt jegyzendő meg az a remélhetőleg természetesnek tekinthető megállapítás, hogy a forráskódba begépelt soroknak egytől egyik szintaktikailag helyesnek kell lennie. Nem lehet olyan kódrészletet tovább "fejleszteni", melyben korábban félbehagyott, avagy szintaktikailag nem helyes részek találhatóak. A modern IDE[9] környezetek a kezünk alá dolgoznak e téren, és a Visual Studio is évről évre javít ezen a területen.

```
1  public class CharacterModel
2  {
3    private const char PIXEL = '#';
4
5    private readonly bool[][] content;
6
7    public int Width
8    {
9      get
10     {
11       return this.content[0].Length;
12     }
13   }
14
15   public CharacterModel(char type)
16     : this(CharacterTrunk.GetModelContent(type))
17   {
18   }
19
20   private CharacterModel(bool[][] content)
21   {
22     this.content = content;
```

[9] Integrated Development Environment

```
23    }
24
25    public String GetRow(int row)
26    {
27      StringBuilder builder = new StringBuilder();
28      for (int column = 0; column < content[row].Length; column++)
29      {
30        builder.Append(content[row][column] ? PIXEL : ' ');
31      }
32      return builder.ToString();
33    }
34
35    public override String ToString()
36    {
37      StringBuilder builder = new StringBuilder();
38      for (int row = 0; row < content.Length; row++)
39      {
40        builder.AppendLine(this.GetRow(row));
41      }
42      return builder.ToString();
43    }
44
45    public static CharacterModel BuildRandomModel(Random random, int minWidth, int
          maxWidth)
46    {
47      int width = random.Next(minWidth, maxWidth + 1);
48      bool[][] content = new bool[CharacterData.HEIGHT][];
49      for (int i = 0; i < CharacterData.HEIGHT; i++)
50      {
51        content[i] = new bool[width];
52        for (int k = 0; k < width; k++)
53        {
54          content[i][k] = random.Next(2) == 1;
55        }
56      }
57      return new CharacterModel(content);
58    }
59
60  }
```

3.15. kód. Karakter modell

3.2.5. Reklám

Miután egy karaktert már le tudunk modellezni, egy szinttel feljebb léphetünk az implementációban. A 3.16. kódrészlet leírja, hogyan szeretnénk létrehozni egy reklámszöveget, illetve azon a mozgatást miként tudjuk elvégezni műveleti szinten (egyelőre animáció nélkül). Ha a reklámszöveget háromszor visszaléptetem, akkor az a terv, hogy balra elkezd kiúszni 3x7 "pixelnyi" oszlop, majd ugyanezen 21 "pixel" a jobb oldalt megjelenik.

A *reklám* osztály felelőssége a karaktermodellek sorozatának, illetve az aktuális "eltolás" értékének kezelése lesz. Előbbi végleges, míg utóbbi változtatható állapot.

```
1  private static void TestAdvertisement()
2  {
3    Advertisement ad = new Advertisement("OBUDAI EGYETEM");
4    Console.WriteLine(ad);
5    ad.StepBack();
6    ad.StepBack();
7    ad.StepBack();
8    Console.WriteLine(ad);
9  }
```

3.16. kód. Fényreklám tesztelése

A konstruktorban megszerzett karakterláncot mint karakter tömböt ha feldolgozzuk, a karakter modelljeink már elő is állnak. Későbbi műveletek során hasznos lesz, ha

ismerjük a teljes reklámszöveg "pixelekben" mért szélességét, ezért az *összegzés tételét*[1] alkalmazva még ugyanebben a ciklusban kiszámoljuk ezt is (Minden karakter között egy "pixel" széles üres helyet hagyunk ki, ezért minden karakter szélességéhez hozzáadunk a ciklusmagban egyet (lásd. 3.17. kódrészlet 33. sor). Mivel az utolsó karakter után viszont nem hagyunk ki üres helyet, ezért a ciklus végén az utoljára hozzáadott értéket levonjuk (lásd. 3.17. kódrészlet 35. sor).).

A megjelenítés algoritmusán érdemes egy kicsit elgondolkodni. Maga az algoritmus nem mondható nehéznek, a mögöttes "matematika" sem egy *rakéta tudomány*, mégis sok apró számítást kell elvégeznünk, és ha nem tudjuk az algoritmus felépítését logikusan megtervezni, a hibajavítás borzasztóan nehéz lesz[10]. A mátrixnyomtató sorról sorra halad a megjelenítéssel, ezért a külső ciklusunk ezen a hét darab soron fog végigfutni (lásd. 3.17. kódrészlet 65. sor). A belső ciklusban az eredeti, eltolás nélküli sort fogjuk legyártani (ehhez felhasználjuk a már a *CharacterModel* osztályban implementált *GetRow(index)* metódust), majd némi matematikai művelettel a benne megtalálható karaktereket eltoljuk a *ShiftLine()* metódus segítségével.

```
1  public class Advertisement
2  {
3    private const char LETTER_SEPARATOR = ' ';
4
5    private int shift;
6    private readonly CharacterModel[] models;
7    private readonly int width;
8
9    private int Width
10   {
11     get
12     {
13       return this.width;
14     }
15   }
16
17   public int Height
18   {
19     get
20     {
21       return CharacterData.HEIGHT;
22     }
23   }
24
25   public Advertisement(String label)
26   {
27     this.shift = 0;
28     this.models = new CharacterModel[label.Length];
29     int width = 0;
30     for (int i = 0; i < label.Length; i++)
31     {
32       this.models[i] = new CharacterModel(label[i]);
33       width += this.models[i].Width + 1;
34     }
35     this.width = width - 1;
36   }
37
38   public void StepForward()
39   {
40     if (this.shift < this.Width)
41     {
42       this.shift++;
```

[10] Képzeljük el a következőt: megjelenik valami a képernyőn az első tesztelés során, ami nem az elvártnak megfelelő. Jobb esetben felismerjük az eredeti karaktereket (hiszen azok megjelenítését már leteszteltük!), rosszabb esetben úgy elcsúsznak a sorok, hogy felismerhetetlenek lesznek. Utóbbi esetben két lehetséges irány van: ha az algoritmusunk felépítése logikus, szépen részfeladatokra bontott, akkor a hibajelenségből tudunk következtetni a hiba helyére. Ellenkező esetben ki kell törölnünk az algoritmust, és készíteni egy újat (utóbbi nagyon sok felesleges időt elvesz, ami nem mindig áll rendelkezésünkre, mégis van olyan szituáció, mikor valóban csak ez tud célra vezetni).

```
43      }
44      else
45      {
46        this.shift = 0;
47      }
48    }
49
50    public void StepBack()
51    {
52      if (this.shift > 0)
53      {
54        this.shift--;
55      }
56      else
57      {
58        this.shift = this.Width;
59      }
60    }
61
62    public override string ToString()
63    {
64      StringBuilder builder = new StringBuilder();
65      for (int k = 0; k < CharacterData.HEIGHT; k++)
66      {
67        StringBuilder line = new StringBuilder();
68        for (int i = 0; i < this.models.Length; i++)
69        {
70          line.Append(this.models[i].GetRow(k)).Append(LETTER_SEPARATOR);
71        }
72        builder.AppendLine(ShiftLine(line.ToString()));
73      }
74      return builder.ToString();
75    }
76
77    private string ShiftLine(string line)
78    {
79      return line.Substring(line.Length - this.shift, this.shift) + line.Substring(0,
          line.Length - this.shift);
80    }
81
82  }
```

3.17. kód. Fényreklám

Sok más jó megoldás is létezik a megjelenítésre, sőt a 3.17. forráskódban bemutatott algoritmust is lehetne optimalizálni (pl. nem kellene minden animált lépésben kiszámolni az eltolás nélküli sorokat). E feladatok már az olvasóra várnak.

3.2.6. Animáció

Utolsó részfeladatunk magának az animációnak az elkészítése. Ehhez valamilyen időközönként periodikusan változtatjuk a reklámfény állapotát (eltolásának értékét), majd mindig megjelenítjük az eredményt. A szemünk előtt a statikus állóképek apró eltéréseiből egy "mozgókép" fog előállni, pontosan ugyanúgy, ahogyan ez a moziban működik. A 3.18. forráskód részlet szerint tervezzük meghívni a már példányosított fényreklám *Play()* metódusát, melynek átadjuk a háttér- és az előtér színét, valamint a várakozás mértékét (valamilyen időegységben, ami most egyébként lényegtelen).

```
1  private static void TestAdvertisementMotion()
2  {
3    Advertisement ad = new Advertisement("OBUDAI EGYETEM ");
4    ad.Play(ConsoleColor.White, ConsoleColor.DarkBlue, 100);
5  }
```

3.18. kód. Animáció tesztelése

A megvalósított metódusra a 3.19. forráskód mutat egy példát. Mivel billentyű leütésére nem várakozhatunk (ez blokkoló utasítás), a billentyűzet puffert kell mindig ellenőriznünk (*Console.KeyAvailable*), hogy megtalálható-e benne az ESC karakter, hiszen ez jelzi számunka az animáció, és egyben az alkalmazásunk végét is.

```
1  public class Advertisement
2  {
3    [..]
4
5    public void Play(ConsoleColor backgroundColor, ConsoleColor foregroundColor, int speed)
6    {
7      Console.WindowHeight = this.Height + 2;
8      Console.WindowWidth = this.Width + 2;
9      Console.ForegroundColor = foregroundColor;
10     Console.BackgroundColor = backgroundColor;
11     Console.Clear();
12     bool exitCycle = false;
13     do
14     {
15       Console.SetCursorPosition(0, 0);
16       Console.WriteLine(this.ToString());
17       this.StepBack();
18       if (Console.KeyAvailable)
19       {
20         exitCycle = (Console.ReadKey(true).Key == ConsoleKey.Escape);
21       }
22       else
23       {
24         Thread.Sleep(speed);
25       }
26     } while (!exitCycle);
27   }
28
29 }
```

3.19. kód. Animáció

A fényreklám elkészítése során - bár szándékosan nem volt ez kimondva - az ún. TDD[11] módszere szerint próbáltunk eljárni. Ugyan még egység teszteket nem írtunk, az a szemlélet, miszerint mindig csak az aktuálisan megvalósítandó részfeladatra összepontosítunk, és előbb minden esetben azt írjuk fel, hogyan fogjuk, hogyan tervezzük felhasználni az implementálandó részeket, mind-mind a TDD technikái közé tartozik. Ha az elején elkezdjük megtervezni a fényreklám osztályt, vajon megterveztük-e volna a *CharacterTrunk*, vagy a *CharacterData* típusokat, esetleg ezek helyett valami sokkal bonyolultabbat helyeztünk volna el egy UML diagrammon?

[11] Test-Driven Development

3.3. Fénycsíkos menü

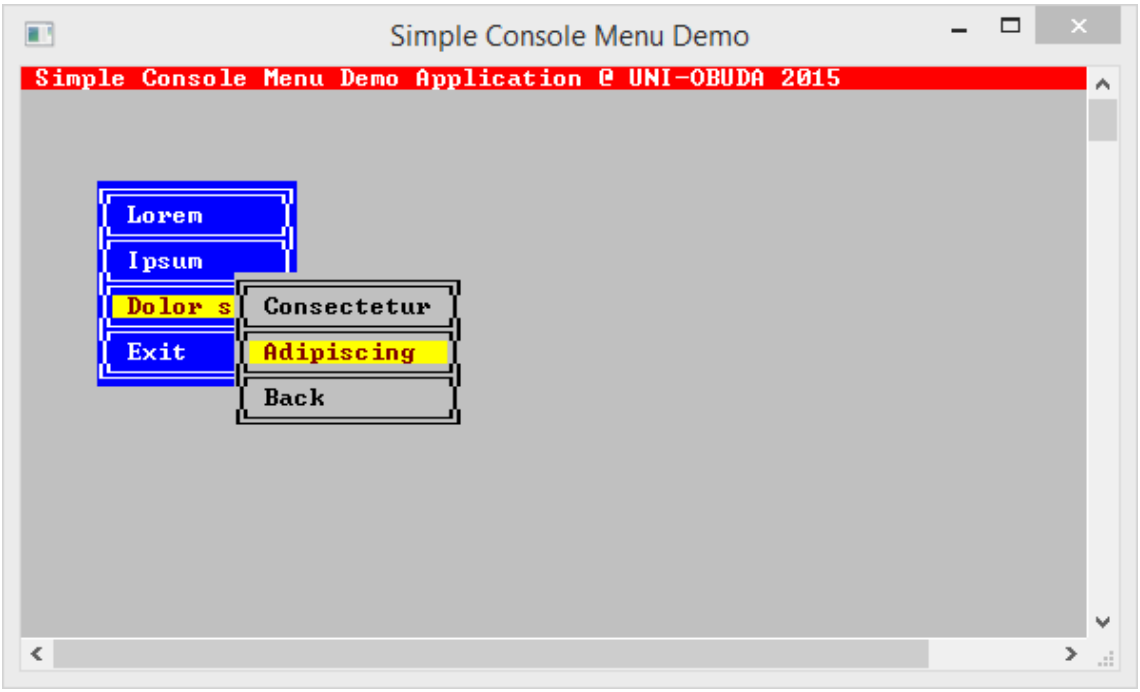
Előfeltétel

programozási tételek, osztályrelációk, egységbezárás, többalakúság, konstruktorok felelőssége

Készítsünk ún. fénycsíkos menüt, mely újrafelhasználható komponense lehet bármely későbbi menüvezérelt alkalmazásnak. Az alkalmazás a képernyő tetszőleges pontján egy halmaznyi menüfeliratot meghatározva képes legyen megjelenni, és a kurzor billentyűk segítségével (fel és le) lehessen a menüpontok között "mozogni". Amelyik menüponton az ENTER billentyű leütésre kerül, az legyen a kiválasztott elem, melyet adjon vissza az alkalmazást a menüt felhasználó komponensnek.

Néhány évvel korábban karaktergrafikus keretalkalmazásokkal és ezek ablakozó rendszereivel gyakran lehetett találkozni. Ma már csak elvétve egy-egy gyógyszertár vagy egy eldugott benzinkúton köszönnek vissza. Ha az ügyintéző egér használata nélkül nagyon gyorsan és nagyon sokat nyomogatja a tabulátor és az ENTER billentyűket, szinte biztosak lehetünk benne hogy egy ilyen rendszert használ éppen.

Egy ilyen rendszert elkészíteni nem nehéz, ráadásul igen felhasználóbarát képet ad. Minimális plusz fejlesztési munka szükséges egy fénycsíkos menü elkészítéséhez ahhoz képest, hogy egyszerű menüpontokat megjelenítünk, és pl. számok alapján kell a felhasználónak választania. A 3.4. ábra egy lehetséges implementáció képét mutatja. Egy ilyen alkalmazásban bőven van helye kreativitásnak: paraméterezhetővé tehetjük a menüben alkalmazott színeket, árnyékokat, és definiálhatunk billentyűkódokat mint alternatív vezérlési mód. A jegyzet nem fog teljességre törekedni, de pl. a színeket a bemutatott megoldás is konfigurálhatóvá teszi.



3.4. ábra. Fénycsíkos menü

3.3.1. Entitások azonosítása

Ezúttal tervezzünk/gondolkodjunk egy kicsit előre, mielőtt elveszünk az implementáció részleteiben. Egy fénycsíkos menüben *menüpontok* vannak, melyek mindegyike egy-egy felirattal és azonosítóval kell hogy rendelkezzen annak érdekében, hogy a felhasználó számára is olvasható legyen, és a kód is tudja egyértelműen azonosítani. Az ilyen osztályok sokszor jobban hasonlítanak egy-egy struktúrára, mintsem önálló felelősséggel rendelkező osztályra (lásd. 3.20. kódrészlet), viszont megnyitják a továbbfejlesztésre való lehetőséget, amire a fejezet végén még visszatérünk.

```
1  public class MenuItem
2  {
3
4    private readonly int id;
5    private readonly String label;
6
7    public int Id
8    {
9      get { return this.id; }
10   }
11
12   public String Label
13   {
14     get { return this.label; }
15   }
16
17   public int Length
18   {
19     get { return this.label.Length; }
20   }
21
22   public MenuItem(int id, String label)
23   {
24     this.id = id;
25     this.label = label;
26   }
27
28 }
```

3.20. kód. Menüpont (MenuItem.cs)

Írjuk össze azokat az adatokat és felelősségeket, melyek egy egyszerű menü modellezéséhez szükségesek. Az összeírt elemeket a 3.1. táblázat tartalmazza, többet is, mint amit egyébként a bemutatott implementáció tartalmazni fog.

Adat/állapot	Művelet/felelősség
Háttérszín	Előző menüpontra ugrás
Előtérszín	Következő menüpontra ugrás
Megjelölt elem háttérszíne	Vezérlés
Megjelölt elem előtérszíne	Megjelenítés
Bal felső sarok X koordinátája	Menüpont hozzáadása
Bal felső sarok Y koordinátája	Menüpont törlése
Menüelemek	Aktuális elem megjelölése (színcsere)
Árnyék létezése	...
...	

3.1. táblázat. Egyszerű menü modellje

Egy adott osztályban az adattagok száma átlagosan hét, vagy annál inkább kevesebb. Ez egy gyakorlati szabály, mely körülbelül azt mondja ki, hogy elég valószínűtlen, hogy

valaminek, aminek hétnél több adattagja van, a mögött egyetlen felelősség bújik meg. Van egy másik szabály is, mely fontos lehet: az egy osztályban lévő adattagok egymástól csak azonos mértékben függhetnek. Ez azt mondja ki, hogy nem lehet olyan adattagokat kiválasztani, melyeknek egymáshoz kicsivel is több közül van, mint a többi adattaghoz. Természetesen ez is egy olyan szabály, melytől a gyakorlat sokszor eltér. Ennek ellenére ha megvizsgáljuk a felsorolt adattagokat, akkor azt láthatjuk, hogy ezek között jócskán találunk összetartozó elemeket:

- Szín sablon
  - Normál szín
    - Háttérszín
    - Előtérszín
  - Megjelölt szín
    - Háttérszín
    - Előtérszín
- Pozíció
  - Bal felső sarok X koordinátája
  - Bal felső sarok Y koordinátája
- Menüelemek
- Árnyék létezése *(a bemutatott implementációban nem fog szerepelni)*
- ...

Az összetartozó adattagoknak tudunk egy csoport nevet adni, és ha ezt megtettük, akkor ezzel már létre is hoztunk számára egy új típust. A *pozíciókat* tartalmazó osztályban két olyan adattagot zárunk egységbe, melyek között az adott osztályban nézve nincsen már kiemelten szorosabb kapcsolat, mint az osztály többi mezőjével szemben (mivel az osztályban rajtuk kívül nincs több mező), és ahol eredetileg szerepeltek az adattagok (menü osztály modellje), ott mostantól csupán egy *pozíció* példány lesz megtalálható, mely önmagában pontosan akkora "függésben" van a menüelemektől, mint az árnyék létezésétől. A *színsablonokat* tartalmazó osztályban eredetileg négy adattagot zárnánk egységbe, azonban ezek között újabb nem azonos szinten lévő függést észlelhetünk: a megjelölt elem előtérszíne és a megjelölt elem háttérszíne szorosabb kapcsolatban van, mint bármelyik elem pl. a normál előtérszínnel összehasonlítva[12]. Az új típusok bevezetésével az eredeti menü osztályban felsorolt adattagok száma 3-4 elemre csökkent, mely már egy sokkal reálisabb érték.

Vizsgáljuk meg az új típusokban a mezők elnevezéseit is! Amennyiben a kijelölt menüelem háttérszíne a *menüt* modellező osztályban maradt volna, a mező neve pl. *"highlightedBackgroundColor"* lehetett volna. Abban a pillanatban, hogy mindezt áthelyezzük egy olyan osztályba, mely a színek/sablonok modellezéséről szól, a név nem csak hogy nem kell, hogy ilyen hosszú legyen, hanem mindez redundáns is volna! Lévén egy színeket tartalmazó osztályban nem kell a mezőket "color" postfix-szel ellátni ahhoz, hogy

[12] Itt más csoportosítás is felismerhető: lehetne az előtérszíneket és a háttérszíneket is csoportosítani, azonban ha a művelet oldalt is felrajzoljuk (amit most az egyszerűség kedvéért nem tettünk meg), látnánk hogy közös művelete a normál színek, illetve a megjelölt színek aktiválásának lesz.

mindenki számára egyértelmű legyen, miről is van szó. Az aggregációkon keresztül ahogyan elérjük az elemeket, kiolvasható lesz minden számunkra szükséges elem (pl. a *menü* osztály egy példánymetódusából a *this.template.Normal.Background* egyértelműen megcímezhetne egy színt [13]).

3.3.2. Pozíció és Menü színek

```
1  public class Position
2  {
3    private readonly int top;
4    private readonly int left;
5
6    public Position(int top, int left)
7    {
8      this.top = top;
9      this.left = left;
10   }
11
12   public void SetCursor(int row)
13   {
14     Console.SetCursorPosition(this.left, this.top + row);
15   }
16
17 }
```

3.21. kód. Pozíció (Position.cs)

Az implementációt folytathatjuk a *pozíciót* (lásd. 3.21. kódrészlet) illetve a *menü színeket* (lásd. 3.22. kódsor) leíró osztályokkal. Megfigyelhetjük, hogy egyik adattaghoz sem készítünk nyilvános tulajdonságot az osztályokban. Bár elképzelhető, hogy az implementáció során szükség lesz rájuk, előre nem dolgozunk e téren (nem lesz rájuk szükség, lévén a vezérlést műveleten keresztül fogjuk kiváltani, és ehhez mindkét osztályban találunk metódusokat (*SetCursor()* illetve SetColors() néven)).

```
1  public class MenuColor
2  {
3
4    public static readonly MenuColor DEFAULT_NORMAL = new MenuColor(ConsoleColor.Blue,
         ConsoleColor.White);
5    public static readonly MenuColor DEFAULT_HIGHLIGHTED = new
         MenuColor(ConsoleColor.Yellow, ConsoleColor.DarkRed);
6
7    private readonly ConsoleColor background;
8    private readonly ConsoleColor foreground;
9
10   public MenuColor(ConsoleColor background, ConsoleColor foreground)
11   {
12     this.background = background;
13     this.foreground = foreground;
14   }
15
16   public void SetColors()
17   {
18     Console.BackgroundColor = this.background;
19     Console.ForegroundColor = this.foreground;
20   }
21
22 }
```

3.22. kód. Szín (MenuColor.cs)

[13] A fejezetben bemutatott példa implementációban a *Background* tulajdonság nem létezik a *MenuColor* osztályban.

3.3.3. Sablon

A normál és a kijelölt színeket leíró osztály példányait a *sablon* osztályban zárjuk egységbe, ahogyan ezt a 3.23. kódrészlet bemutatja. Az osztályt példányosítani lehet a négy szín, vagy a két menü szín példány átadásával is. Előbbi gyakoribb használata esetén elérhetjük, hogy a *MenuColor* osztály léte pusztán a menüt kezelő alkalmazás belügye legyen, hiszen kifelé nem szükséges az osztályt elérni.

Mind a *MenuColor*, mind a *MenuTemplate* osztályba fel lett véve néhány konstans, mely az alapértelmezett színek használatának megkönnyítését szolgálja. Attól, hogy egy alkalmazásban lehetőséget adunk arra, hogy ezernyi ponton konfiguráljuk a működést, nem jelenti azt hogy az API-t felhasználók számára tíz paraméteres konstruktorokat kell szolgáltatnunk, avagy halmaznyi inicializáló metódus meghívását kell kikényszerítenünk. A metódus (illetve konstruktor) paraméterek számára is létezik gyakorlati szabály: ha már öt paramétere van az alprogramnak, gondolkodjunk el azon, hogy ezek közül melyikeket tudjuk csoportosítani, illetve hogy bizonyosan mindegyiket ezen metódusnak szeretnénk-e átadni.

```
1  public class MenuTemplate
2  {
3
4    public static readonly MenuTemplate DEFAULT_TEMPLATE = new
         MenuTemplate(MenuColor.DEFAULT_NORMAL, MenuColor.DEFAULT_HIGHLIGHTED);
5
6    private readonly MenuColor normal;
7    private readonly MenuColor highlighted;
8
9    public MenuColor Normal
10   {
11     get { return this.normal; }
12   }
13
14   public MenuColor Highlighted
15   {
16     get { return this.highlighted; }
17   }
18
19   public MenuTemplate(ConsoleColor normalBackground, ConsoleColor normalForeground,
         ConsoleColor highlightedBackground, ConsoleColor highlightedForeground)
20     : this(new MenuColor(normalBackground, normalForeground), new
           MenuColor(highlightedBackground, highlightedForeground))
21   {
22   }
23
24   public MenuTemplate(MenuColor normal, MenuColor highlighted)
25   {
26     this.normal = normal;
27     this.highlighted = highlighted;
28   }
29
30 }
```

3.23. kód. Szín sablon (MenuTemplate.cs)

3.3.4. Menü

Minden építőkő elkészült ahhoz, hogy a *menüt* leíró osztályt elkészítsük (lásd. 3.24. kód). Az implementációban a dinamikus elemszámú listák használata szándékosan mellőzve van, erre későbbi feladatok kapcsán fog a jegyzet visszatérni (lásd. 4.1.4. fejezet). Az osztály konstruktorban megkaphatja, hogy maximum mennyi menüelemet tartalmazhat, majd az *Add()* metódus iteratív hívásával felvehetjük az egyes tételeket.

A *MenuItem* és a *Position* osztályok használata teljesen el van fedve az implementációban, kifelé ezen osztályok alapvetően nem látszódnak, és amennyiben megelégszünk az alapértelmezett színekkel, a sablon definiálására sem lesz gondunk. Figyeljük meg, hogy hiába van az osztálynak három konstruktora, mindegyik egyféleképpen hozza létre a példányt, hiszen a konstruktorok egymást hívják, és egyetlen redundáns kódsort sem tartalmaznak.

```
1  public class SimpleMenu
2  {
3    private static readonly int MAXIMUM_MENU_ITEMS = 10;
4
5    private readonly MenuTemplate template;
6    private readonly Position position;
7    private readonly MenuItem[] items;
8    private int index;
9    private int current;
10
11   public SimpleMenu(int top, int left)
12     : this(top, left, MenuTemplate.DEFAULT_TEMPLATE)
13   {
14   }
15
16   public SimpleMenu(int top, int left, MenuTemplate template)
17     : this(top, left, template, MAXIMUM_MENU_ITEMS)
18   {
19
20   }
21
22   public SimpleMenu(int top, int left, MenuTemplate template, int maxMenuItem)
23   {
24     this.position = new Position(top, left);
25     this.template = template;
26     this.items = new MenuItem[maxMenuItem];
27     this.index = 0;
28     this.current = 0;
29   }
30
31   public void Add(int id, String label)
32   {
33     if (this.index < this.items.Length)
34     {
35       this.items[this.index++] = new MenuItem(id, label);
36     }
37   }
38
39 }
```

3.24. kód. Menü modellje (SimpleMenu.cs)

3.3.5. Megjelenítés

A modellezett adataink készen állnak arra, hogy vizuálisan megjelenítsük a *menüt* a karaktergrafikus képernyőn. A *menü* szélességét és magasságát meghatározza a benne szereplő *menüpontok* felirata. A magasság egyenes arányos a *menüpontok* számával, míg a szélességhez a leghosszabb *menüpont* kiválasztására van szükségünk (a *maximumkiválasztás tételét*[1] a 3.25. kódrészlet 19. sorában definiált *GetTheLargestLength()* metódus tartalmazza). A rajzolás bár elfoglal néhány kódsort, valójában csupán végig kell szaladnunk a *menüpontokon*, és sorban megjeleníteni őket. Ügyelni kell a menü tetejének és aljának megjelenítésére, és a menüpontok közötti vonalakra. Amennyiben éppen az aktuális/megjelölt *menüpontot* "rajzoljuk", ne felejtsük el az előtér és a háttér színét megváltoztatni (lásd. a 3.25. kódrészlet 54. sora). Bár a megjelenítés a *Draw()* metódus meghívásával fog megvalósulni, valójában hét metódus készült-e célra. Minden olyan kódsort, melynek egy közös nevet tudunk adni, szervezzük ki egy **private** metódusba. Ezzel

átlátható, és ami talán még fontosabb: javítható, karbantartható kódunk lesz. Ha valahol hiba van, hetekkel később is tudni fogjuk, hogy hol és miként javítsuk ki. Ami az implementáció készítésekor plusz perceket "vesz el tőlünk", az a hibajavítás során órákat jelent!

```
1  private void Draw()
2  {
3    this.template.Normal.SetColors();
4    int maxLength = this.GetTheLargestLength() + 2;
5
6    this.DrawTop(maxLength);
7    int row = 0;
8    for (int i = 0; i < this.index; i++)
9    {
10     this.DrawMenuItem(maxLength, i, ++row);
11     if (i != this.index - 1)
12     {
13       this.DrawMiddle(maxLength, ++row);
14     }
15   }
16   this.DrawBottom(maxLength, ++row);
17 }
18
19 private int GetTheLargestLength()
20 {
21   int ret = 0;
22   if (this.index > 0)
23   {
24     ret = this.items[0].Length;
25     for (int i = 1; i < this.index; i++)
26     {
27       int current = this.items[i].Length;
28       if (ret < current)
29       {
30         ret = current;
31       }
32     }
33   }
34   return ret;
35 }
36
37 private void DrawTop(int maxLength)
38 {
39   this.position.SetCursor(0);
40   Console.WriteLine("O" + "".PadLeft(maxLength, '-') + "O");
41 }
42
43 private void DrawMenuItem(int maxLength, int index, int row)
44 {
45   this.position.SetCursor(row);
46   this.template.Normal.SetColors();
47   Console.Write("|");
48   this.SetColors(index);
49   Console.Write(" " + this.items[index].Label.PadRight(maxLength - 1, ' '));
50   this.template.Normal.SetColors();
51   Console.Write("|");
52 }
53
54 private void SetColors(int index)
55 {
56   (index == this.current ? this.template.Highlighted : this.template.Normal).SetColors();
57 }
58
59 private void DrawMiddle(int maxLength, int row)
60 {
61   this.position.SetCursor(row);
62   Console.WriteLine("O" + "".PadLeft(maxLength, '-') + "O");
63 }
64
65 private void DrawBottom(int maxLength, int row)
66 {
67   this.position.SetCursor(row);
68   Console.WriteLine("O" + "".PadLeft(maxLength, '-') + "O");
69 }
```

3.25. kód. Megjelenítés (SimpleMenu.cs)

3.3.6. Vezérlés

A *SimpleMenu* osztályból már csupán a vezérlés nyilvános metódusa hiányzik (mely egyébként magát a rajzolást is elvégzi, ezért volt a *Draw()* metódus **private** láthatóságú). Egy hátultesztelős ciklusban a leütött billentyűkre figyelünk, kiemelten a fel- és lefele irányú kurzor billentyűkre, illetve az ENTER-re. Ha előbbiek leütése (vagy folytonos tartása) bekövetkezik, akkor az aktuális fénycsíkkal jelzett menüpont indexét megváltoztatjuk. A fénycsíkos menük általában "körbe forognak", vagyis a menü aljáról illetve tetejéről át lehet "ugrani" az ellenkező pólusra.

```
1  public MenuItem Process()
2  {
3    ConsoleKey key;
4    do
5    {
6      this.Draw();
7      key = Console.ReadKey(true).Key;
8      switch (key)
9      {
10       case ConsoleKey.UpArrow:
11         this.Up();
12         break;
13       case ConsoleKey.DownArrow:
14         this.Down();
15         break;
16     }
17   } while (key != ConsoleKey.Enter);
18   return this.items[this.current];
19 }
20
21 private void Down()
22 {
23   this.current = this.current < this.index - 1 ? this.current + 1 : 0;
24 }
25
26 private void Up()
27 {
28   this.current = this.current > 0 ? this.current - 1 : this.index - 1;
29 }
```

3.26. kód. Menüvezérlés (SimpleMenu.cs)

3.3.7. Kliens kód

A 3.27. kódrészlet egy lehetséges felhasználását mutatja be az elkészült fénycsíkos menü alkalmazásnak. Négy menüpontot definiálunk első körben, melyek közül az utolsó aktiválására kiléphetünk a programból, a harmadik esetén pedig egy gyermek menü fog megjelenni, újabb három választási lehetőséggel. Természetesen utóbbihoz semmi plusz kódolásra az API-ban nem volt szükség, pusztán egy új menüt példányosítottunk.

```
1 private static readonly int WINDOW_WIDTH = 70;
2 private static readonly int WINDOW_HEIGHT = 25;
3
4 private static void TestSimpleMenu()
5 {
6   DrawFrame();
```

```
7
8   SimpleMenu menu = new SimpleMenu(5, 5);
9   menu.Add(10, "Lorem");
10  menu.Add(20, "Ipsum");
11  menu.Add(30, "Dolor sit");
12  menu.Add(40, "Exit");
13
14  MenuTemplate template = new MenuTemplate(ConsoleColor.Gray, ConsoleColor.Black,
        ConsoleColor.Yellow, ConsoleColor.DarkRed);
15  SimpleMenu childMenu = new SimpleMenu(9, 14, template);
16  childMenu.Add(31, "Consectetur");
17  childMenu.Add(32, "Adipiscing");
18  childMenu.Add(33, "Back");
19
20  MenuItem item;
21  do
22  {
23    item = menu.Process();
24    switch (item.Id)
25    {
26      case 10:
27      case 20:
28        Program.WriteMessage(1, 1, item.Label);
29        break;
30      case 30:
31        MenuItem childItem;
32        do
33        {
34          childItem = childMenu.Process();
35          switch (childItem.Id)
36          {
37            case 31:
38            case 32:
39              Program.WriteMessage(1, 1, childItem.Label);
40              break;
41          }
42        } while (childItem.Id != 33);
43        break;
44    }
45
46  } while (item.Id != 40);
47 }
48
49 private static void DrawFrame()
50 {
51   Console.BackgroundColor = ConsoleColor.Gray;
52   Console.ForegroundColor = ConsoleColor.White;
53   Console.Clear();
54
55   Console.SetWindowSize(Program.WINDOW_WIDTH, Program.WINDOW_HEIGHT);
56   Console.Title = "Simple Console Menu Demo";
57   Console.BackgroundColor = ConsoleColor.Red;
58   Console.ForegroundColor = ConsoleColor.White;
59   Console.SetCursorPosition(0, 0);
60   Console.Write(" Simple Console Menu Demo Application @ UNI-OBUDA
       2015".PadRight(Program.WINDOW_WIDTH));
61   Console.BackgroundColor = ConsoleColor.Gray;
62 }
63
64 private static void WriteMessage(int x, int y, string label)
65 {
66   Console.BackgroundColor = ConsoleColor.Gray;
67   Console.ForegroundColor = ConsoleColor.Black;
68   Console.SetCursorPosition(x, y);
69   Console.Write(label.PadRight(30));
70 }
```

3.27. kód. Menü tesztelése (Program.cs)

A fénycsíkos menü továbbfejlesztése számos lehetőséget rejt magában. Öröklés alkalmazásával kialakíthatunk egy olyan hierarchiát a menüpontokból, melyek egyike lehetne jelölő négyzetes vagy rádió gombos viselkedésű. Abban is lenne fantázia, ha az almenüket, melyek egy-egy menüpont kiválasztásakor megjelennek aggregálnánk a szülő menühöz.

Mindez a *Composite design pattern* (lásd. 7.2. fejezet) alkalmazására egy remek példa volna!

3.4. Gyakorló feladatok

Készítse el a régebben népszerű *Szerencsekerék* televíziós vetélkedő interaktív változatát karaktergrafikus ábrázolásmóddal. A feladványokat egy szöveges állomány tartalmazza (minden sorban pontosvesszővel elválasztva egymástól a kategóriát és a feladványt). A játék indulásakor véletlenszerűen válasszon ki egy feladványt, és jelenítse meg az átfordítatlan karaktereket a képernyőn egy-egy 3x3-mas keretben (az üres közép jelezze az átfordítatlan állapotot). Figyeljen arra, hogy a feladvány legfeljebb három sorban, és középre rendezve jelenjen meg! A játékot játszó felhasználó sorban elkezdi leütni a karaktereket. A leütések száma megjelenik a képernyő sarkán, illetve találat esetén a betűk "átfordulnak" (megjelennek a nekik szánt keretben).

Készítsen el egy rulettasztalt karaktergrafikus ábrázolásmód segítségével. Az asztalon kurzor karakterek segítségével lehessen "mozogni" és tétet tenni. Az a szám/mező, melyen a mozgó "kurzor" megtalálható, invertált háttérszínnel rendelkezzen. ENTER hatására a kiválasztott mezőn lehessen tétet tenni egy "felugró" ablakban, bekérve a tét összegét (a tét összege ne lehessen nagyobb mint a játékos egyenlege). ESC leütésekor a gép sorsoljon ki egy számot, majd a játékos esetleges nyeresége legyen elkönyvelve az új játék indulása előtt.

4. fejezet

Nyelvi ismeretek

Ismeretszerzés

láncolt listák, dinamikus elemszámú sorozatok, kivételkezelés, kivételek hierarchiája, eseménykezelés az objektum-orientált programozásban, eseménykezelés delegate-ek segítségével

A fejezetben olyan nyelvi elemek kerülnek bemutatásra, melyek esetén a szintaktika ismerete sokszor nem elégséges ahhoz, hogy megfelelően alkalmazzuk a gyakorlatban őket.

Egy-egy magas szintű nyelvben megtalálható programozási technika, *szintaktikai cukorka* nem azért létezik, mert valaki íróasztal mellett megtervezte, hanem "egyszerűen" az igény szülte a létét. Mielőtt nem létezett, a nyelv korábban létező lehetőségeivel implementálták. Egy idő után számos mérnök elkészítette a saját implementációját a célból, melyet akár library-kba szervezett, publikált, open-source világban a közösség részévé tett. Lassan elkezdett kialakulni egy ún. *best-practice* az adott probléma megoldására, és elterjedt a sokféle - bár egymáshoz egyre jobban hasonlító - megoldás közül néhány különösen szerencsés példány. Azon a ponton, mikor egy programozási nyelv fejlődése során beemel egy-egy ilyen implementációt a nyelvi elemek közé, és akár ellátja azt egy szintaktikai cukorkával, elérkezünk ahhoz a ponthoz, mikor a fejlesztő mindezt már természetesnek véli, és úgy építkezik rá, mint a nyelv alapműveleteire. Ezzel alapvetően nincs is semmi probléma, hiszen ez teszi lehetővé azt, hogy az egyre magasabb szinten megfogalmazott problématér átlátható legyen az ember számára, és ebből kifolyólag képes legyen olyan összetett problémák megoldására is, melyet korábban nem látott át. Azonban ahhoz, hogy egy ismeretlen problémához mérnöki megközelítés szerint tudjunk hozzányúlni, érdemes egy-két szinttel ezen szintaktikai cukorkák "alá" látni. A fejezet ebben a szemléletben íródott.

4.1. Láncolt listák

Előfeltétel

programozási tételek, osztályrelációk, rekurzív algoritmusok

A sorozatok alapvető fontosságúak a számítástechnikai algoritmusok készítése során. Vannak dolgok, melyekben az emberek még jobbak a gépeknél, és bizony van jó pár dolog (és egyre több), amiben a gépek legyőznek minket. A humánpolitikai és etikai oldalával a kérdésnek nem foglalkozva, ezen utóbbi esetekben a gépek térnyerése borítékolható, illetve már javában tart. Egy adott művelet iteratív elvégzése tipikusan olyan feladat, melyet egy gép "hibátlanul" és fáradhatatlanul el tud végezni, nem véletlen hát hogy a *programozási tételek*[1] is mind eme sorozaton végzett algoritmusokra épülnek.

Eddig minden esetben fix elemszámú tömb adatszerkezettel oldottuk meg a felmerülő problémákat, azonban vélhetően már megfogalmazódott az igény a dinamikus elemszámú sorozatokra. Ezek tipikus megvalósítása a *láncolt lista*[2], melyben pl. minden elem tartalmaz egy hivatkozást a következő elemre, ezáltal a sorozat fejétől kezdve az adatszerkezet bejárható és a kívánt műveletek elvégezhetőek.

Bár csábító az elemek számának korlátlan bővítési lehetősége, ne legyenek vágyálmaink: a *láncolt listák* kezelése teljesítményben sokszor alulmarad a fix elemszámú tömbökéhez képest, de még talán ennél is fontosabb az a jelenség, miszerint ha tudjuk előre, hogy mennyi elemre van szükségünk, akkor a fix elemszámú tömbök alkalmazása nem pusztán optimalizáció, hanem egy jelzés eme információra, tényállásra!

A láncolt lista bővítése új elemmel, elemekkel, illetve tetszőleges láncszemek törlése mind-mind olyan műveletek, melyek rendkívül gyorsan és hatékonyan elvégezhetőek. Feltéve persze hogy az akció "helyének" referenciája rendelkezésre áll számunkra. Ugyanis itt jön az adatszerkezet legnagyobb hátránya: az adatok elérése borzasztó lassú. Különféle technikák léteznek ennek javítására, pl. az oda-vissza láncolt listák, de ebbe a témakörbe tartoznak a fák és a speciális gráfok is.

4.1.1. Dominó

Készítsünk el egy *dominó sort* reprezentáló adatszerkezetet láncolt lista segítségével. A legelső dominó kulcsszerepet játszik (hiszen ezen keresztül meglökhető az egész sorozat), és az egymást követő dominók kapcsolatban vannak egymással (egy irányban). E kapcsolat az osztályrelációk szintjén az *aggregáció* lesz, vagyis mindegyik dominó példány tartalmazni fogja a rákövetkező dominó referenciáját (lásd. 4.1. kódrészlet). A *ToString()* metódus (18. sor) egy apró rekurzió, melynek kilépési feltétele az, ha már a következő dominó referenciája nem található (ez esetben leáll a rekurzív hívás, és sorban visszatérnek a kiértékelt *String* literálok a hívás helyéhez).

```
1 public class Domino
2 {
3   private readonly int value;
4
5   private Domino next;
6
7   public Domino Next
8   {
9     get { return this.next; }
```

```
10      set { this.next = value; }
11    }
12
13    public Domino(int value)
14    {
15      this.value = value;
16    }
17
18    public override string ToString()
19    {
20      return this.value + " -> " + (this.next != null ? this.next.ToString() : "NIL");
21    }
22  }
```

4.1. kód. Dominó

A 4.2. kódrészlet bemutatja a dominó lánc létrehozását. A kód még finom túlzással sem mondható könnyen olvashatónak, ráadásul nélkülöz mindenféle hibakezelést (a *dot notation* láncolt használata a *NullArgumentException* tipikus keletkezési helye).

```
1  Domino head = new Domino(1);
2  head.Next = new Domino(1);
3  head.Next.Next = new Domino(2);
4  head.Next.Next.Next = new Domino(3);
5  Console.WriteLine(head);
6
7  Domino tail = head.Next.Next.Next;
8  Domino domino = new Domino(5);
9  tail.Next = domino;
10 Console.WriteLine(head);
```

4.2. kód. Lánc létrehozása

A Domino osztály apró kiegészítésével egy kicsit olvashatóbbá tehetjük a lánc létrehozását/bővítését (lásd. 4.3. kódrészlet). A technika nagyon hasonló ahhoz, ahogyan a *StringBuilder* osztályt bővítjük, ott azonban természetesen nincsen láncolás, az *Append()* metódus minden esetben **this**-el tér vissza.

```
1  public class Domino
2  {
3    [..]
4
5    public Domino Add(int value)
6    {
7      this.Next = new Domino(value);
8      return this.Next;
9    }
10 }
```

4.3. kód. Lánc elegáns növelése

A dominók olvashatóbb létrehozását a 4.4. kódrészlet mutatja be.

```
1  Domino list = new Domino(1);
2  list.Add(1).Add(2).Add(3).Add(5).Add(8).Add(13);
3  Console.WriteLine(list);
```

4.4. kód. Lánc létrehozása elegánsan

4.1.2. Bővítés, törlés

A *láncolt lista* adatszerkezet legnagyobb előnye, hogy a láncot bárhol megbonthatjuk anélkül, hogy erről a bontás előtti és utáni elemeken kívül "bárkihez" hozzá kellene érnünk.

Értelemszerűen a bontáshoz szükségünk lesz egy segédváltozóra, amibe legelső lépésben a bontás utáni elem referenciáját eltároljuk. Beszúrás esetén (lásd. 4.5. kódrészlet) ezen a ponton létrehozzuk az új elemet, és a bontás előtti elemhez csatoljuk, miközben az új elem következő dominója lesz a leválasztott lista vége. Törlésre több alternatíva is létezik, ezek közül kettőt mutat be a 4.6. kódrészlet. A *DeleteNext()* metódus feltételezi hogy a törlési pont előtti elem referenciájával rendelkezünk, míg a *DeleteByIndex()* képes bármely elemhez képest egy meghatározott távolságban lévő elem kikötésére.

```
1  public class Domino
2  {
3    [..]
4
5    public void Insert(int value)
6    {
7      Domino tmp = this.next;
8      this.next = new Domino(value);
9      this.next.next = tmp;
10   }
11 }
```

4.5. kód. Lánc bővítése

```
1  public class Domino
2  {
3    [..]
4
5    public void DeleteNext()
6    {
7      this.next = this.next.next;
8    }
9
10   public void DeleteByIndex(int relativeIndex)
11   {
12     Domino tmp = this;
13     for (int i = 0; i < relativeIndex - 1; i++)
14     {
15       tmp = tmp.Next;
16     }
17     tmp.Next = tmp.Next.Next;
18   }
19 }
```

4.6. kód. Láncszem kiszedése

A bővítési illetve törlési műveletek felhasználását a 4.7. kódrészlet mutatja be.

```
1 head.Next.Next.Insert(42);
2 Console.WriteLine(head);
3
4 head.DeleteByIndex(2);
5 Console.WriteLine(head);
```

4.7. kód. Beszúrás illetve törlés tesztelése

4.1.3. Lineáris Keresés

Egy láncolt listában a *lineáris keresés*[1] művelete is kicsit máshogy néz ki, mint ahogyan ezt egy fix elemszámú tömbbel elvégeznénk, illetve kézenfekvő lehet a rekurzív megoldás ez esetben is. Mindegyikre ad példát a 4.8. kódrészlet.

```
1 public class Domino
```

```
2  {
3    [..]
4
5    public Domino RelativeFind(int value)
6    {
7      Domino tmp = this;
8      while (tmp.value != value)
9      {
10       tmp = tmp.Next;
11     }
12     return tmp;
13   }
14
15   public Domino Find(int value)
16   {
17     return this.value == value ? this : this.next.Find(value);
18   }
19 }
20
21 [..]
22
23 Console.WriteLine(head.RelativeFind(3));
24 Console.WriteLine(head.Find(3));
```

4.8. kód. Lineáris keresés

A láncolt listák rendkívül rugalmas adatszerkezetek. Gyorsan és a rekurzív megoldások kézenfekvősége végett deklaratívan képesek vagyunk az elvárt követelményeknek leginkább megfelelő típus létrehozására. Nagy láncok esetén lehet készíteni indextáblákat, hashtáblákat. Könnyen és biztonságosan tudjuk több szálon feldolgozni őket, hiszen bárhol szétszedve a láncot két azonos értékű, egymástól azonban tökéletesen független láncot kapunk.

4.1.4. Tömbök láncolt listája

Ha a fix elemszámú tömböknek megvan az a nagyszerű előnyük, hogy bármely elemüket egy egyszerű matematikai művelet segítségével direkt címezni tudjuk, a láncolt listáknak pedig a dinamikus bővíthetőség, törlés lehetősége adott, joggal merül fel a kérdés: miért nem egyesítjük e két előnyös tulajdonságot egy új típusban, és hozzuk létre tömbök láncolt listáját? Elsősorban azért, mert mindezt már megtették helyettünk, ez a típus a C# nyelvben az *ArrayList* (lásd. 4.9. kódrészlet).

```
1  ArrayList list = new ArrayList();
2  list.Add(42);
3  list.Add("Lorem");
4
5  foreach (Object element in list)
6  {
7    Console.WriteLine(element);
8  }
9
10 IEnumerator iterator = list.GetEnumerator();
11 while (iterator.MoveNext())
12 {
13   Console.WriteLine(iterator.Current);
14 }
```

4.9. kód. Tömbök láncolt listája

4.1.5. Type-safe csomagoló osztály

A típus azonban a tömb adatszerkezethez, illetve az általunk készített láncolt listával összehasonlítva egy kiemelten fontos tulajdonsággal nem rendelkezik: nem *type-safe*, vagyis kizárólag annyit tudunk a benne eltárolt elemekről, hogy azok valamilyen *System.Object* leszármazottak, és ennek megfelelően leginkább a *ToString()* metódusukat tudjuk meghívni explicit átalakítás nélkül.

Kicsit körülményesen, de egy *wrapper* osztály elkészítésével megoldhatjuk a *type-safe* viselkedést (lásd. 4.10. kódrészlet), azonban a generikus típusok megismerését követően mindez szükségtelenné fog válni (lásd. 5.3. fejezet).

```
1  public class ArrayListWrapper
2  {
3    private ArrayList list;
4
5    public ArrayListWrapper(int capacity)
6    {
7      this.list = new ArrayList(capacity);
8    }
9
10   public void Add(String value)
11   {
12     this.list.Add(value);
13   }
14
15   public int IndexOf(String value)
16   {
17     return this.list.IndexOf(value);
18   }
19
20   [..]
21 }
```

4.10. kód. Dinamikus lista

A láncolt tömbök *wrapper* osztályának *type-safe* felhasználására mutat példát a 4.11. kódrészlet.

```
1 ArrayListWrapper wrapper = new ArrayListWrapper(10);
2 wrapper.Add("Lorem");
3 wrapper.Add("Ipsum");
4 wrapper.Add("Dolor");
5 Console.WriteLine(wrapper.IndexOf("Ipsum"));
```

4.11. kód. Dinamikus lista

4.2. Kivételkezelés

Előfeltétel

programozási tételek, osztályrelációk, öröklés, overload konstruktorok

A kivételkezelés kapcsán legtöbben a "hiba" szóra asszociálnak, pedig a kifejezés rendkívül találó. Nem véletlen, hogy a technikát nem "hibakezelésnek" hívjuk! A kivételkezelés során valamilyen esemény vagy akció hatására példányosítunk egy olyan "speciális" osztályt, mely a jelenség tényét általában leírja, majd ezen példányt *eldobjuk* a kódblokkot hívó irányába. Egy osztály attól lesz "dobható", ha ősei között szerepel az *Exception* osztály. Hibát lekezelni **try-catch** blokkban tudunk, és a hiba kezelése lehetőséget biztosít számunkra a jelenség futási idejű megoldására.

Ez eddig gyakorlatilag szintaktika. De mit is jelent valójában a kivételkezelés? Talán a legfontosabb dolog, amit érdemes a legelején egyértelműsíteni: nem szükséges minden (üzleti) hibánál (saját) kivételt dobni! Az objektum-orientált világban egy tagfüggvény egyetlen referenciával vagy egyetlen értékkel képes visszatérni (ez igen szabad kezet ad természetesen), miközben tetszőleges számú kivételt dobhat. A C# nyelvben csak ún. *unchecked* kivételek vannak, ezért az egyes kódrészek által dobott vagy dobható kivételek lekezelése teljesen az őket hívó blokk felelősségére hárul (lekezeli, vagy nem kezeli le). A kivétel pontosan addig a pontig fog "utazni" a hívási láncban, amíg valaki nem fog foglalkozni vele. Legrosszabb esetben a futtatókörnyezet. Mindebből következik-e az, hogy a hibakezelésnek mindez egy elegáns és objektum-orientált módja? Természetesen nem. Ebből elsősorban az következik, hogy egy kivétel dobása és lekezelése legalább egy példányosításból, és számos elágazásból áll. Ha a hiba ennél olcsóbb eszközökkel is jelezhető, akkor egy kivétel dobása sokszor túlzás.

Vannak olyan üzleti metódusok, melyek során egy-egy "hibát" lehet egyértelműen és kivételkezelés nélkül jelezni. Pl. egy sorozatban egy elem indexét visszaadó függvény ha -1-el tér vissza, egyértelműen jelzi hogy a megadott elem nem található meg a sorozatban. Ugyanakkor egy tetszőleges egész számokat tartalmazó sorozatban a legkisebb elem kiválasztása során a -1-el nem tudjuk jelezni a szélsőérték hiányát (üres sorozatnak nincsen lekisebb eleme), hiszen elképzelhető hogy egy sorozatban a -1 a legkisebb elem. Utóbbi esetben a kivételkezelés jó megoldás lehet, még akkor is, ha egyébként sok erőforrást felemészt.

Vannak olyan kivételek, melyeket a gyakorlatban *soha* nem kezelünk le. Ilyen kivétel a talán leggyorsabban megismert *NullArgumentException*. Egyrészt logikus kérdés volna a **catch** blokk tartalma. Bár legtöbbször a hibát egy rendszer csupán naplózza, azonban e *NullArgumentException* esetén miképpen javítanánk a hibát, ha egy adott referencia nem létezik. Létrehoznánk? Kétséges, mivel feltehetően ha az adott kontextusban létre lehetne hozni, akkor a *NullArgumentExeption* nem merült volna fel. Ezen kivétel lekezelése súlyos programozói hiba, ugyanis programhibát fel el! Ha valahol a kódban *NullArgumentException* keletkezik, akkor a program "szálljon el" jó messzire, és mindez remélhetőleg egy olyan tesztfázisban történik, ami még közel van a fejlesztéshez (vagyis nem pl. az ügyfélnél jön elő).

Térjünk egy utolsó gondolat erejéig vissza a kivételekre. Miket tekinthetünk kivételeknek? A kivétel egy olyan esemény a programban, mely egy adott üzleti kód futási ágában ritkábban következik be, mint a többi esemény. Pl. képzeljünk el egy banki átutalási tranzakciót, ahol a forrás és a célszámla szerepel mint bemeneti paraméter, az

utalandó összeg mellett. Az esetek 90%-ban a tranzakció sikeresen lezajlik, és a forrás számláról eltűnik, a cél számlán pedig megjelenik az összeg (meg némi banki/állami költség is keletkezik értelemszerűen). Van azonban olyan eset is, mikor a forrásszámlán nincs megfelelő mennyiségű összeg, de a számlatulajdonosnak van másik számlája a banknál, és rendelkezik is meghatalmazással az intézmény felé, hogy másik számlájáról teljesíteni lehessen az átutalást forráshiány esetén. Mindez az esetek csupán 2%-a, azonban végeredményben ugyanúgy sikeres lesz a tranzakció. Kivételes eset természetesen a forráshiány, avagy a célszámla nem létezése is. Az üzleti kód kimenete ezek fényében sikeres vagy sikertelen tranzakció. Egy alkalmazásnak nem elegendő az esetek döntő többségére helyes eredményt adnia. Az esetek 100%-át szükséges lefedni, ez alá adni nem érdemes. De mi történne akkor, ha a ritkább esetek lekezelését elvégző kódrészletek elágazások formájában jelennének meg, és nem kivételkezelés segítségével? Előállna az a furcsa szituáció, miszerint pl. egy 100 soros kódrészletből 15 sor foglalkozik azzal, ami az esetek 90%-a, és az összes többi kódsor csak azért létezik, hogy a maradék 10%-al is foglalkozzunk. Itt most nem a 100 sornyi kódot szeretnénk megspórolni a kivételkezeléssel, sokkal inkább kihangsúlyozni a lényegi 15 sort a többi kódrészlet közül. Ugyanis kódsorok és kódsorok között rangsort kell tudni állítani, nincs jogi és politikai egyenlőség: ugyanaz a művelet egy sok magos számlálós ciklusban luxus, máshol pedig elegáns! Az a kódsor, mely az esetek 90%-ában lefut, bizony nagyobb felelősséget hordoz, sokkal robosztusabbnak kell lennie, mint egy fél évente használt funkciónak (várhatóan a hibajegyek a gyakori esetben előbb kijönnek).

4.2.1. Komponensek kivételkezelése

A kivételek ugyanolyan osztályok, mint korábban készített társaik. Ősei között szerepelnie kell az *Exception* osztálynak, illetve ebből kifolyólag illik az örökölt és *overloadolt* konstruktorokat feltüntetni. Ha alkalmazás specifikus kivétel osztály készítünk, ajánlás az *ApplicationException*-t használni ősnek. Elvben nincs korlátozva az, hogy egy kivétel üzleti metódusok tömegét tartalmazza, azonban ezt értelemszerűen kerüljük el. A kivétel osztály példányai tipikusan rövid életűek. Keletkezésüket követő "pillanatokban" eljutnak arra a pontra, ahol lekezelésük elkezdődik, és innentől kezdve feleslegessé válnak, hiszen legfontosabb felelősségüket teljesítették: továbbították a ritka és kivételes esemény tényét a megfelelő helyre. Joggal merül fel a kérdés, milyen praktikus objektum-orientált módszerek léteznek, melyeket érdemes a kivételosztályok készítésekor is figyelembe venni?

Képzeljünk el egy alkalmazást, mely két "rétegből" tevődik össze: egy adattárolási, *persistence*, illetve egy üzleti, *facade* rétegből. Utóbbi réteg egyik feladata az elképzelt alkalmazásunkban, hogy a bejövő hívásokat lebontsa pl. elemei lekérdezésekké a tárolási réteg felé. A tárolási réteg számos kivételes esettel "találkozhat", pl. a teljesség igénye nélkül a beszúrandó kulcs már szerepel a táblában (*DuplicateKeyException*, lásd. 4.12. kódrészlet), avagy az új érték megsért egy egyediségre vonatkozó megszorítást (*UniqueConstraintException*, lásd. 4.13. kódrészlet).

```
1 public class DuplicateKeyException : PersistenceException
2 {
3   public DuplicateKeyException(String message, String field) : base(message, field) { }
4   public DuplicateKeyException(String message, String field, Exception inner) :
        base(message, field, inner) { }
5 }
```

4.12. kód. Kulcs ismétlődés

```
1 public class UniqueConstraintException : PersistenceException
2 {
3   public UniqueConstraintException(String message, String field) : base(message, field)
        { }
4   public UniqueConstraintException(String message, String field, Exception inner) :
        base(message, field, inner) { }
5 }
```

4.13. kód. Egyedi érték megszorítás

Ha megfigyeljük a bemutatott kivételeket, mindegyik a *PersistenceException*-ből származik (lásd. 4.14. kódrészlet). Ez is egy saját kivétel, azonban nem véletlenül **abstract**! Mindig valamilyen specifikus kivétel keletkezik a tárolási rétegben, mely sokkal jobban leírja a hiba természetét, azonban ezek mindegyikére igaz lesz, hogy köthető valamilyen mezőhöz[1]. Mindazonáltal nem az itt tapasztalt redundancia megszüntetés a legnagyobb haszna a bevezetett leszármazási láncnak!

```
1 public abstract class PersistenceException : ApplicationException
2 {
3   private readonly String field;
4
5   public String Field
6   {
7     get { return this.field; }
8   }
9
10  public PersistenceException(String message, String field) : this(message, field, null)
        { }
11  public PersistenceException(String message, String field, Exception inner)
12    : base(message, inner)
13  {
14    this.field = field;
15  }
16 }
```

4.14. kód. Tárolási réteg kivétele

Nézzünk egy elképzelt üzleti metódust a *persistence* rétegből. A *DashboardService GetInstruments()* metódusa (lásd. 4.15. interface, illetve 4.16. implementáció) visszaadja a paraméterben kapott felhasználóhoz tartozó "műszerek" neveinek listáját[2]. Az implementációban természetesen van pár "kivételes" eset. Ha pl. a felhasználó azonosítója páratlan szám, rögtön dobunk egy *DuplicateKeyException*-t, ellenkező esetben a 42-es azonosító kivételével visszaadunk néhány "műszert". Utóbbi - azon kívül hogy mint ismert[4], *az élet értelme* - egy *UniqueConstraintException* dobással köszön el a metódus további részének végrehajtásától.

```
1 public interface DashboardService
2 {
3   List<String> GetInstruments(int userId);
4 }
```

4.15. kód. Műszerfal service

```
1 public class DashboardServiceImpl : DashboardService
2 {
```

[1] Ez azért nem teljesen igaz minden esetben, de az említett két implementációban igen. A feladat szemléltetési szempontjából ezen most felülemelkedhetünk.

[2] Az implementáció jelenleg lényegtelen. Egy webes "műszerfalat" képzelhetünk el, ahol minden felhasználó saját maga állíthatja össze "widget"-jeit, "műszereit". Ezek listáját adja vissza képzeletbeli alkalmazásunk metódusa, valamilyen perzisztens rétegből

```
3   public List<String> GetInstruments(int userId)
4   {
5     List<String> result = null;
6     if (this.CheckUserId(userId))
7     {
8       result = this.GetResultSet(userId);
9     }
10    return result;
11  }
12
13  private bool CheckUserId(int userId)
14  {
15    if (userId % 2 != 0)
16    {
17      throw new DuplicateKeyException("Lorem ipsum " + userId, "user");
18    }
19    return true;
20  }
21
22  private List<String> GetResultSet(int userId)
23  {
24    List<String> result = new List<String>();
25    if (userId == 42)
26    {
27      throw new UniqueConstraintException("Dolor sit " + userId, "amet");
28    }
29    result.Add("Settings");
30    result.Add("Navigation");
31    return result;
32  }
33 }
```

4.16. kód. Műszerfal service implementáció

Mi történik akkor, amikor a *facade* rétegből valaki meghívja a *GetInstruments()* metódust? A hibátlan lefutások során visszakapja az elvárt karakterlánc listát, azonban néhány kivételes esetben *DuplicateKeyException* illetve *UniqueConstraintException*-t is kaphat. Nem beszélve természetesen arról a további száz kivételről, melyeket még szintén dobhat pl. egy relációs adatbázis, amikor megpróbáljuk szolgálataink alá vonni. A *facade* rétegből (lásd. 4.17. kódrészlet) azonban nem szükséges az összes kivételt felsorolnunk, bőven elegendő a *PersistenceException*-t elkapnunk (lásd. 20. sor), hiszen minden érintett kivétel ennek leszármazottja lesz (és bármi is történik az adatbázis oldalon, általában a megoldás a *rollback*[3] lesz).

```
1  public class DashboardFacade
2  {
3    private readonly Random random;
4    private readonly DashboardService service;
5
6    public DashboardFacade()
7    {
8      this.random = new Random();
9      this.service = new DashboardServiceImpl();
10   }
11
12   public List<String> GetUserInstruments()
13   {
14     List<String> instruments = null;
15     int userId = this.GetCurrentUserId();
16     try
17     {
18       instruments = this.service.GetInstruments(userId);
19     }
20     catch (PersistenceException e)
21     {
22       Console.WriteLine("Error in '" + e.Field + "' field. Details: " + e.Message);
23     }
```

[3] Az adott tranzakció során esetlegesen elvégzett adatmódosítások visszavonása.

```
24      return instruments;
25    }
26
27    private int GetCurrentUserId()
28    {
29      return this.random.Next(50);
30    }
31  }
```

4.17. kód. Facade réteg

4.2.2. Összegzés

Általánosan is igaz, hogy egy *module*, egy *library* az általa kezelt, és kifelé publikált hibákat egy logikus és jól felépített hierarchiába zárja. A gyakorlatban természetesen eszünkbe se jusson *DuplicateKey* és *UniqueConstraint* kivétel osztályokat gyártani. Ezeket biztosítják számunkra megfelelő keretrendszerek/osztálykönyvtárak. A hangsúly itt most azon van, hogy a *facade* rétegben ha elkezdjük kibontani a specifikus kivételeket, akkor ezzel nem csupán egy redundáns kódot kapunk, hanem a réteget felruházzuk olyan felelősséggel, melyhez semmilyen köze nincsen (Mit érdekli a *facade* réteget, hogy a dupla kulcsok hibát okoztak? Szerencsés esetben arról sem tud, hogy vannak kulcsok egyáltalán!).

Mindenképpen érdemes megemlíteni, hogy - néhány kivételes esettől eltekintve[4] - nem *type-safe* objektum-orientált megoldás az, hogy minden *Exception*-t elkapunk és lekezelünk. Ez esetben pl. a *NullArgumentException*-t is elkapjuk, és egy hibásan megírt "lekezelő" blokk nem pusztán a hibát fedheti el, hanem az alkalmazás minőségében is nagyon komoly csorbát ejthet.

[4] Távoli metódus hívásoknál, webservice végpontoknál, miután (!) minden ismert hibát lekezeltünk, és pl. a megfelelő hibává alakítottuk (pl. json text vagy xml soap message), biztonsági okokból egy *Exception* blokkal, és egy általános hibaüzenet gyártásával fogunk találkozni a gyakorlatban. E blokkok naplózását érdemes kiemelten figyelni és kezelni.

4.3. Eseménykezelés

Előfeltétel

osztályrelációk, öröklés, interface-ek, egységbezárás, delegate-ek, felelősségek kezelése

Az objektum-orientált programozás javarészt arról szól, hogy a világ modellezése során létrehozott típusok (osztályok) között kapcsolatokat építünk fel. E kapcsolatokat nevezzük osztályrelációknak, és az alábbi csoportosítás érvényes ezekre:

- **Asszociáció**: Egymástól független osztályok társítása (lehet kétirányú).
  - **Aggregáció**: Rész-egész egyirányú kapcsolat. Gyenge aggregáció esetén a rész életciklusa független leget az egésztől.
  - **Kompozíció**: Erős aggregáció, a tartalmazott nem vehető ki a tartalmazóból.
- **Öröklés**: Struktúra, viselkedés megosztása, hierarchia kialakítása.

Mind az aggregáció, mind a kompozíció asszociációnak számít, egyedül az öröklés az, ami egészen más mélységekben hozza kapcsolatba az osztályokat egymással. *Asszociáció* pl. egy tagfüggvény metódusának *String* típusú argumentuma, hiszen ez esetben az osztály és a *String* típus egy példánya között kapcsolat lép fel. Ugyancsak asszociáció az, ha egy metódus visszatérési típusa egy másik osztálynak a példánya (minden olyan metódus ide számít, ami nem **void**, értéktípusú, és ha nem önmaga típusát adja vissza, bár ha ez esetben egy másik példányról van szó, akkor ez is tekinthető asszociációnak). Asszociáció az is, ha egy alprogramon belül példányosítunk egy másik osztályt, meghívjuk annak metódusait, stb. Általánosan kijelenthető, hogy minden olyan interakció két osztály között asszociáció, mely során valamilyen - tipikusan gyenge - formában kapcsolatba kerülnek egymással.

Az *aggregáció* már sokkal erősebb, tartalmazási kapcsolat. Az egyik osztály tárol és kezel egy példányt egy másik osztály példányából. Pl. egy személy osztályban a nevét leíró *String* típusú *name* mező egy aggregációt hoz létre az osztály és a *String* között. Ehhez képest a *kompozíció* csak abban tér el, hogy logikailag a tartalmazott elem az alkalmazás üzleti szempontjai szerint önmagában nem létezhet, az mindig a tartalmazó példány életciklusával együtt mozog. Az aggregáció és a kompozíció szintaktikailag nem tér el egymástól.

Az *öröklés* nem csupán a csoportosítás szempontjából különálló. Szintaxisa, lehetőségei és az objektum-orientált világban folyamatosan fejlődő lehetőségei lévén egészen más dimenziókat nyit meg két osztály között. A reláció segítségével természetesen rendkívül nagy mennyiségű redundáns kódtól szabadulunk meg, azonban típusaink strukturáltságára nézve is jótékony hatású.

És a történet itt ér véget. Mármint az osztályok közötti kapcsolatteremtés története. Nem hiányzik a felsorolásból az eseménykezelés? Hiszen ez pontosan arról szól, hogy az egyik osztály értesíti a másik osztályt egy akció végrehajtásáról. Mi ez, ha nem kapcsolat a két osztály között! Természetesen az, azonban nem új elem. Az eseménykezelés nem más, mint egy egyszerű aggregáció. Apró nyelvi trükk az, ami megvalósítja a tipikusan öröklési és tartalmazási kapcsolatban sem álló osztályok között a kommunikációt. Ebből a szempontból vizsgálva az eseménykezelés egy gyenge kapcsolatnak tekinthető, legtöbbször lazán kapcsolt elemekből áll.

4.3.1. Feladatkezelő egyszerű monitorozással

Egy egyszerű feladatkezelő alkalmazás segítségével nézzük meg, hogy miként valósul meg az eseménykezelés az objektum-orientált nyelvekben. A *Task* osztály (lásd. 4.18. kódrészlet) egy folyamat végrehajtásának idejéig "él", és az egyes fázisok végrehajtása után mindig regisztrálja a százalékos állapotot. Az első részfeladat végrehajtása után CREATED állapotból STARTED állapotra vált (az állapotok listáját a 4.19. felsorolás típus tartalmazza), majd a munka legvégén STARTED-ből DONE-ba kerül. Ha valami hiba lép fel, akkor állapota FAILED-be vált át. Tegyük fel, hogy az osztály életciklusától és más osztályokkal való interakciójától függetlenül szeretnénk monitorozni azt, amikor a példány életében állapotváltozás lép fel!

```
1  public class Task
2  {
3    private TaskStatus status;
4    private int progress;
5    private StatusChangeEvent statusChange;
6
7    private TaskStatus Status
8    {
9      set
10     {
11       if (this.status != value)
12       {
13         this.status = value;
14         if (this.statusChange != null)
15         {
16           this.statusChange.OnChange(this.status, this.progress);
17         }
18       }
19     }
20   }
21
22   public Task()
23   {
24     this.status = TaskStatus.CREATED;
25     this.progress = 0;
26   }
27
28   public Task DoIt(int progress)
29   {
30     this.progress += progress;
31     this.Status = TaskStatus.STARTED;
32     if (this.progress >= 100)
33     {
34       this.Status = TaskStatus.DONE;
35     }
36     return this;
37   }
38
39   public void Error()
40   {
41     this.Status = TaskStatus.FAILED;
42   }
43
44   public void BindStatusChangeEvent(StatusChangeEvent statusChange)
45   {
46     this.statusChange = statusChange;
47   }
48
49   public void UnbindStatusChangeEvent()
50   {
51     this.statusChange = null;
52   }
53
54   public override string ToString()
55   {
56     return "Task: " + this.progress + " %";
57   }
58 }
```

4.18. kód. Feladat

```
1 public enum TaskStatus
2 {
3   CREATED,
4   STARTED,
5   DONE,
6   FAILED
7 }
```

4.19. kód. Állapotok

E feladatra ad megoldást az eseménykezelés, mely szintaktikailag nem más, mint egy interface aggregációja az osztályban! Ezen interface ráadásul tipikusan egy művelettel rendelkezik, mely az esemény hordozója lesz (lásd. 4.20. kódrészlet). Az *OnChange()* metódus meghívása, az aktuális állapottal és százalékos értékkel fog értesítést adni azon "külső" szemlélő számára, aki monitorozni szeretné a *Task* példány életútját. A *Task* osztály (lásd. 4.18. kódrészlet) eme interface típusból kezdetben nem referál példányra (lásd. 4.18. kódrészlet 5. sor), viszont lehetőséget biztosít arra, hogy valaki "feliratkozzon" ezen referenciának az értékadásán keresztül az állapotváltozásra. A feliratkozás *BindStatusChangeEvent()* metóduson keresztül (lásd. 44. sor) történhet meg, míg az esemény keletkezése a **private** *Status* tulajdonság **set** blokkjában (16. sor). Csak abban az eseten kell értesíteni az eseményről a megfigyelőt, ha az létezik. Ha senkit nem érdekel az adott *Task* állapotváltozása, az nem fogja megakasztani a feladat végrehajtását.

```
1 public interface StatusChangeEvent
2 {
3   void OnChange(TaskStatus status, int progress);
4 }
```

4.20. kód. Állapotváltozás eseménye

Szimuláljuk le egy feladat létrehozását és monitorozását. A monitorozónak legyen egy azonosítója. A 4.21. kódrészlet 2. sorában feliratunk egy *StatusChangeHandler* példányt a *Task* példány *StatusChangeEvent* típusú "event"-jére. Mivel utóbbi egy interface, a *StatusChangeHandler*-nek egy olyan osztálynak kell lennie, mely ezen *StatusChangeEvent*-et implementálja (lásd. 4.22. kódrészlet).

```
1 Task task = new Task();
2 task.BindStatusChangeEvent(new StatusChangeHandler("INFO"));
3 Console.WriteLine(task.DoIt(30));
4 Console.WriteLine(task.DoIt(20));
5 Console.WriteLine(task.DoIt(50));
```

4.21. kód. Tesztelés

```
1 public class StatusChangeHandler : StatusChangeEvent
2 {
3   private readonly String logger;
4
5   public StatusChangeHandler(String logger)
6   {
7     this.logger = logger;
8   }
9   public void OnChange(TaskStatus status, int progress)
10  {
```

```
11      Console.WriteLine("[" + logger + "] Change status to " + status + " (" + progress +
            "%)");
12    }
13  }
```

4.22. kód. Állapotváltozás eseménykezelője

Ha futtatjuk a szimulációt, az állapotváltozások folyamatában látni fogjuk az INFO monitorozó által megkapott állapotváltozásokat. Érdemes megfigyelni a felelősségek elosztását. A *Task* osztályra egyáltalán nem tartozik az eseménykezelő kódja (mely kiírja a monitorozó azonosítóját és az állapotváltozás adatait a képernyőre). Lehetne a monitorozó implementációja olyan, hogy egy e-mail üzenetet küld minden állapotváltozásról a cégvezetőnek. Mindehhez semmi köze nincsen magának a feladatnak! Utóbbinak a felelőssége csupán az állapotváltozás tényének megállapítása, és az esemény jelzése. Ugyancsak megjegyzendő az is, hogy a "kliens" kódnak (lásd. 4.21. kódrészlet) szintén nincs felelőssége az eseménykezelőben.

4.3.2. Feladatkezelő többszörös monitorozással

Az előzőleg ismertetett monitorozásnak van egy igen komoly korlátja: egy feladatot egy időben kizárólag egy megfigyelő tud felügyelni. Ez nem csupán eltér az elvárt viselkedéstől, de hibás megközelítés is, hiszen a megfigyelők között semmilyen reláció nem áll fenn, így egymásról sem tudnak (reláció kizárólag az eseményt tartalmazó, és az eseménykezelőt implementáló osztály között áll fenn)!

A megoldáshoz az interface (esemény) aggregációja helyett egy listányi interface-t fogunk aggregálni! Az adott listába a feladat karbantartja azokat a megfigyelőket, akik számára értesítést szükséges küldeni. A 4.23. kódrészletben az látszik, hogy két naplózó megfigyelőt, és egy e-mail küldő megfigyelőt iratunk fel ugyanannak a feladatnak az állapotváltozására.

```
1 Task task = new Task();
2 task.BindStatusChangeEvent(new StatusChangeHandler("INFO"));
3 task.BindStatusChangeEvent(new StatusChangeHandler("DEBUG"));
4 task.BindStatusChangeEvent(new StatusChangeEmailHandler("admin@qwaevisz.hu", "David"));
5 Console.WriteLine(task.DoIt(30));
6 Console.WriteLine(task.DoIt(20));
7 Console.WriteLine(task.DoIt(50));
```

4.23. kód. Tesztelés

Az e-mail üzenetet küldő eseménykezelőt a 4.24. kódrészlet mutatja be. A szimuláció végett e-mail üzenetet valójában nem küldünk, azonban a példa bemutatja, hogy különféle eseménykezelő implementációk lehetségesek, és mindez teljesen független a *Task* osztályától. Számára irreleváns az, hogy kiket és milyen formában érdekel az állapotváltozás.

```
1  public class StatusChangeEmailHandler : StatusChangeEvent
2  {
3    private readonly String email;
4    private readonly String name;
5
6    public StatusChangeEmailHandler(String email, String name)
7    {
8      this.email = email;
9      this.name = name;
10   }
11
12   public void OnChange(TaskStatus status, int progress)
```

```
13   {
14     Send(this.email, this.name, "Change status to " + status + " (" + progress + "%)");
15   }
16
17   private static void Send(String email, String name, String body)
18   {
19     Console.WriteLine("Sending email to " + name + " (" + email + "): " + body);
20   }
21 }
```

4.24. kód. Állapotváltozás e-mail küldő eseménykezelője

A *Task* osztályt néhány ponton szükséges módosítani a cél érdekében (lásd. 4.25. kódrészlet). A konstruktorban létrehozzuk a lista referenciáját (24. sor), a *BindStatusChangeEvent()* metódusban az eseménykezelőt hozzáadjuk (31. sor), míg az *UnbindStatusChangeEvent()* metódusban töröljük (36. sor) az interface-ek listájából. A korábbi **null** vizsgálatot lecseréljük a lista bejárására (13-16. sor), mely egy üres lista esetén egyetlen értesítést sem fog elküldeni.

```
1  public class Task
2  {
3    [..]
4    private List<StatusChangeEvent> statusChanges;
5
6    private TaskStatus Status
7    {
8      set
9      {
10       if (this.status != value)
11       {
12         this.status = value;
13         foreach (StatusChangeEvent statusChange in this.statusChanges)
14         {
15           statusChange.OnChange(this.status, this.progress);
16         }
17       }
18     }
19   }
20
21   public Task()
22   {
23     [..]
24     this.statusChanges = new List<StatusChangeEvent>();
25   }
26
27   [..]
28
29   public void BindStatusChangeEvent(StatusChangeEvent statusChange)
30   {
31     this.statusChanges.Add(statusChange);
32   }
33
34   public void UnbindStatusChangeEvent(StatusChangeEvent statusChange)
35   {
36     this.statusChanges.Remove(statusChange);
37   }
38
39   [..]
40 }
```

4.25. kód. Tesztelés

Az értesítések sorrendisége lényegtelen, azonban a választott implementációban legelőször az az eseménykezelő fog lefutni, aki az adott szálban legelőször iratkozott fel az eseményre.

4.3.3. Eseménykezelés szintaktikai cukorkával

Talán megfogalmazódott a kérdés többekben is: az eseménykezeléshez egy halom osztályt kell létrehozni. Igen. Erre talán ez a legjobb és legszerencsésebb válasz. A tiszta objektum-orientált alkalmazásban tengernyi osztály van, és lehetőség szerint minden osztálynak egyetlen kizárólagos felelőssége azonosítható. A gyakorlatban ez ritkán valósul meg, és a képet az is árnyalja, hogy a hibrid (nem tisztán objektum-orientált) C# nyelv létrehozott egy *szintaktikai cukorkát* az eseménykezelés megkönnyítésére.

A szintaktikai cukort az **event** kulcsszó alkalmazása során ízlelhetjük meg, ehhez azonban egy **delegate**-re van szükségünk. A *delegate* egy függvényreferencia típus. Minden olyan metódus, mely a *delegate* által leírt követelménynek megfelel, az adott referenciával hivatkozható (többalakúság értelmezése a függvényeken). Követelménynek a függvény argumentum típusai, azok sorrendisége és a visszatérési érték típusa számít. Ennek megfelelően a 4.26. kódrészletben bemutatott *SimplifiedStatusChangeHandler delegate*-nek pontosan azok a metódusok felelnek meg, melyeket korábban a *StatusChangeEvent* interface *OnChange()* implementációiként használtunk.

```
1 public delegate void SimplifiedStatusChangeHandler(TaskStatus status, int progress);
```

4.26. kód. Tesztelés

A *Task* osztály ismételt módosítása (lásd. 4.27. kódrészlet) nagyban leegyszerűsíti az osztály kódmennyiségét (a *bind* és *unbind* metódusok, valamint az inicializálás teljesen eltűnt). Ellenőrizni a feliratkozást az eseményre ezúttal is szükséges. Ha legalább egyvalaki érdeklődik, akkor az *event* kulcsszóval megjelölt *delegate* típusú függvényreferencia már nem *null* lesz. Az *event* kulcsszó a háttérben a *delegate*-ek láncolt listáját tárolja, és a felülírt += és -= operátorok segítségével kezeli elemeit. Tipikusan nyilvános mezőnek szokták felvenni.

```
1  public class Task
2  {
3    [..]
4    public event SimplifiedStatusChangeHandler statusChangeEvent;
5
6    private TaskStatus Status
7    {
8      set
9      {
10       if (this.status != value)
11       {
12         this.status = value;
13         if (this.statusChangeEvent != null)
14         {
15           this.statusChangeEvent(this.status, this.progress);
16         }
17       }
18     }
19   }
20
21   [..]
22 }
```

4.27. kód. Tesztelés

Az eseménykezelés tesztelése néhány szintaktikai elemben tér el a korábban már bemutatott példáktól (lásd. 4.28. kódrészlet). *Bind* metódus helyett a += operátort használjuk.

```
1  private static void Test()
2  {
3    Task task = new Task();
4    task.statusChangeEvent += new StatusChangeHandler("INFO").OnChange;
5    task.statusChangeEvent += new StatusChangeHandler("DEBUG").OnChange;
6    task.statusChangeEvent += new StatusChangeEmailHandler("admin@qwaevisz.hu",
         "David").OnChange;
7    task.statusChangeEvent += LocalStatusChangeEventHandler;
8    Console.WriteLine(task.DoIt(30));
9    Console.WriteLine(task.DoIt(20));
10   Console.WriteLine(task.DoIt(50));
11 }
12
13 private static void LocalStatusChangeEventHandler(TaskStatus status, int progress)
14 {
15   Console.WriteLine("Change status to " + status + " (" + progress + "%)");
16 }
```

4.28. kód. Tesztelés

A 4.28. kódrészlet felhívja a figyelmet egy nagyon gyakori, ám kissé zavaró felhasználási módra. Lehetőségünk van *Handler* osztály nélkül is eseménykezelőt készíteni. Ez egy csábító (újabb osztály elkészítését mellőzi), ám veszélyes cukorka. Így legtöbbször az eseménykezelő kódja egy olyan osztályba kerül (sokszor példányszintű metódusként), melyhez felelősség szerint nincs sok köze. Ha belegondolunk abba, hogy pl. az e-mail küldést megvalósító *handler* helyett ezt a megoldást választjuk, akkor a tartalmazó osztály még két új adattagot is aggregál az eseménykezelő kedvéért. Figyeljünk oda a felelősségekre ha tiszta objektum-orientált programot szeretnénk készíteni, és mindig legyünk kíváncsiak arra, hogy egy-egy szintaktikai cukorka mögött mi is van valójában.

5. fejezet

Programozási technikák

Ismeretszerzés

mágikus szám, extension method, attribútumok/annotációk, egység tesztelés, templatek/generikus típusok

A jegyzet e fejezetében nem egy meghatározott problémára keressük a megoldást, miközben az algoritmikus gondolkodásmódban egy-egy lépéssel előrébb jutunk, vagy egy-egy technikával gazdagodunk. E fejezet sokkal szárazabb lesz: olyan technikák bemutatásáról fog szólni, melyet bármely későbbi programunk fejlesztése során alkalmazhatunk, és talán érdemes is alkalmaznunk.

5.1. Felsorolás típusok

Előfeltétel

osztályszintű felelősség, végleges mezők, konstansok

A felsorolás típusok széles körben elterjedtek a legtöbb programozási nyelvben, hol jobban, hol kevésbé integrálódva az objektum-orientált szemléletbe. Arra a nagyon alapvető gondolatmenetre épülnek, hogy egy forráskódban minden név nélkül elhelyezett literál "mágikus számnak" tekinthető, mely "lóg a levegőben", és bár abban a pillanatban, amikor begépeljük, még ismerjük a jelentését, később visszatérve az adott kódrészletre, már gondolkodnunk kell rajta.

Egy *mágikus szám* feloldása legtöbbször nagyon egyszerű: létrehozunk egy **private** láthatóságú konstanst (**const** vagy **static readonly** kulcsszavak felhasználásával, mikor melyik előnyösebb/alkalmazható), és ezt írjuk a felhasználás helyére. Akkor is megtesszük mindezt, ha csupán egyetlen egy helyen fordul elő a konstans! A láthatóság nem véletlenül lett kiemelve: attól, hogy valami konstans, még legtöbb esetben nagyon jól meghatározható, hogy pontosan melyik osztály belügye. Semmi szükség arra, hogy elárasszuk nyilvános konstansokkal a forráskódot! A konstans elnevezése is lényeges pontja a módszernek: az üzleti elnevezést kell alkalmazni, és nem pedig az elnevezett literált! Tipikus rossz példa egy LETTER_X elnevezésű 'x' literál, mely a kilépésre szolgál, illetve egy 1000 ezredmásodperces időkorlát értékeként használt MSEC_1000 konstans. Utóbbira jó példa lehet a TIMEOUT, előbbire pedig az EXIT_KEY elnevezés.

Vannak azonban olyan szituációk, melyek esetén ezen konstansok között összefüggés van. Ilyenkor az *enum* típus alkalmazása kézenfekvő! Egy *enum* alá felsorakoztatva az értékeket egyet jelent azok egységbe zárásával. Csak logikailag összetartozó értékek kerülhetnek egy felsorolás típusba, és az *enum* neve értelemszerűen tükrözi ezt a kapcsolatot. Sok ellenpéldával lehet találkozni, de az *enum* elnevezése **egyesszámú** főnév közeli kifejezés legyen, ahogyan ezt egy osztály elnevezési szabályánál is alkalmazzuk (bár osztály nevében elképzelhető a többesszám, annak ellenére hogy inkább valamilyen struktúrával érdemes ezt a esetet is feloldani[1]).

5.1.1. Enum mint szintaktikai cukorka

A C# *enum* típusa ún. *valuetype*. Nem más, mint egy *szintaktikai cukorka*[2] az egész típusokra [3], kivéve a *char* típust. Az 5.1. kódrészlet a Petri hálózatok[5] éltípusainak felsorolását mutatja be, melyeket egy *enum* részeként deklaráltunk. Alapértelmezésben az *enum* értékek egy *int* értékének felelnek meg. Ha az értéket nem jelezzük, akkor 0-tól kezdődő sorfolytonos egész számoknak felelnek meg. Ahol jelezve van az összerendelés (a példában a legelső *NORMAL* értékhez megadott -1-es érték), ott ezt az értéket veszi fel a konstans, majd a soron következő egész szám lesz a következő (ha nincs jelölve újabb helyen az összerendelés).

[1] Ha van egy osztályunk, melynek könyvek halmazának tárolása és/vagy kezelése a feladata, akkor azt ne *Books*-nak, hanem inkább *BookCatalog*-nak, vagy *Library*-nek nevezzük el.
[2] syntactic sugar
[3] integral types

```
1 public enum EdgeType
2 {
3   NORMAL = -1,
4   INHIBITOR,
5   RESET
6 }
```

5.1. kód. Petri hálózat éltípusai

Az *integral type* megadása szintaktika szerint a típus neve mögött lehetséges, ahogy ezt az 5.2. kódrészlet bemutatja. Ezt a szintaktikát ne keverjük össze az öröklés, illetve az implementáció[4] jelzésével[5].

```
1 public enum UnsignedEdgeType : uint
2 {
3   NORMAL,
4   INHIBITOR,
5   RESET
6 }
```

5.2. kód. Nem-negatív érték megszorítás alkalmazása

Elsősorban teljesítmény szerepe van annak, hogy az *enumok* egyszerű számok a C# nyelvben. A fejezet további részeiben azt fogjuk megvizsgálni, hogy mit lehet, és mit nem lehet a felsorolás típusokkal tenni, és mit tegyünk abban az esetben, ha nem elégszünk meg azzal, amit a C# nyelv szintaktikailag biztosít számunkra e téren.

Az 5.3. kódrészlet végigveszi a leggyakoribb műveleteket, melyeket *enumokkal* végezhetünk. Az 1. sorban hivatkozott NORMAL értéket ha megjelenítjük, visszakapjuk a nevét, de bármikor számmá alakíthatjuk, ahogyan a 2. sorban a -1-es értéket kiírjuk. Az ellenkező irányú átalakításra mutat példát a 4. sor, ahol az 1-es értékből a RESET értéket "állítjuk elő", és a rákövetkező sorban ellenőrizhetjük is az *Enum* osztály[6] *IsDefined()* osztály szintű metódusával, hogy létezik ezen 1-es számhoz rendelt érték a nevezett *enumban* (a metódus *igaz* értéket fog visszaadni). A 7. és 8. sor pontosan arra a veszélyre hívja fel a figyelmet, hogy lehetőségünk van olyan *enum* értéket létrehozni, amely sose létezett, a C#-ban ennek ellenére mindez nem jelent még futás idejű hibát sem. Egyetlen lehetőségünk a védekezésre az *IsDefined()* metódus használata az összes olyan helyen, ahol pl. *enum* paramétert használunk (a gyakorlatban ritkán ellenőrizzük ily módon a bemeneteket).

```
1 EdgeType normalEdge = EdgeType.NORMAL;
2 Console.WriteLine(normalEdge + " --> " + (int)normalEdge);
3
4 EdgeType reset = (EdgeType)1;
5 Console.WriteLine(reset + " --> " + (int)reset + " (" + Enum.IsDefined(typeof(EdgeType),
      reset) + ")");
6
7 EdgeType unknown = (EdgeType)42;
8 Console.WriteLine(unknown + " --> " + (int)unknown + " (" +
      Enum.IsDefined(typeof(EdgeType), unknown) + ")");
9
```

[4] Ebben a szituációban értve alatta azt, hogy egy osztály *implementál* egy vagy több interface-t.

[5] Ahogyan az öröklést se keverjük össze az implementációval, függetlenül attól, hogy a C# nyelven mindezek azonosan vannak jelölve, ezzel rontva az olvashatóságot és az egyértelműséget.

[6] Még véletlenül se gondoljuk azt, hogy az *Enum* osztálynak hierarchikus kapcsolata van a *enum valuetype*-okkal. Azok, akik Java nyelv alaposabb ismerői, furcsa lehet mindez, mivel ott az *Enum* osztály szintén létezik, mint az összes *enum* közös őse (sőt legtöbbször közvetlen őse), amiből természetesen következik az is, hogy a Java nyelven az *enumok* nem *valuetype*-ok, vagy ahogy ott nevezzük ezeket: nem *primitív* típusok.

```
10 EdgeType zeroEdgeType = new EdgeType();
11 Console.WriteLine("'new' " + zeroEdgeType + " --> " + (int)zeroEdgeType);
12
13 EdgeType inhibitor = (EdgeType)Enum.Parse(typeof(EdgeType), "INHIBITOR");
14 Console.WriteLine(inhibitor + " --> " + (int)inhibitor);
15
16 String inhibitorStr = Enum.GetName(typeof(EdgeType), EdgeType.INHIBITOR);
17 Console.WriteLine(inhibitorStr);
18
19 EdgeType whatIsIt = EdgeType.NORMAL + 2;
20 Console.WriteLine(whatIsIt + " --> " + (int)inhibitor);
```

5.3. kód. Enum példányok felhasználása

A 10. és 11. sor egy másik veszélyes felhasználásra mutat példát: "elővehetjük" a zéró értéknek megfelelő felsorolt értéket a típus nevével jelzett "konstruktor" hívás segítségével (10. sor). Remélhetőleg mondani sem szükséges, itt semmilyen objektum-orientált fogalomból ismert konstruktorról nem beszélhetünk. Ezzel az utasítással a tiltó élet fogjuk visszakapni (INHIBITOR). Azon kívül, hogy a kód tökéletesen olvashatatlan lesz ennek alkalmazása mellett, mindez abban az esetben, ha a nullához nincs érték összerendelés, akkor is vissza ad egy olyan *EdgeType*-ot, mely nem is létezik (és nem lesz futás idejű hiba sehol sem). Ajánlásként fel szokták hozni, hogy ne hozzunk létre olyan *enum* típust, melyben nincsen *nulla* értékű elem [7].

Még továbbra is az 5.3. kódsorokat vizsgálva a 13. sor azt mutatja be, hogy karakterláncból hogyan állítjuk elő az *enum* értéket (pl. ha a felsorolás érték neve állományból, vagy egy SOAP üzenetből származik), illetve a 16. sorban ennek ellenkezője van soron: miként állítjuk elő a karakterláncot a felsorolt értékből. A 19. sor a furcsaságok egy következő szintjén hozza elő: képesek vagyunk összeadás és kivonás műveleteket végezni az *enum* értékeken! Teljesen világos: ha a Petri hálózat NORMAL éltípusához hozzáadunk kettőt, akkor RESET élet kapunk! Óvatosan bánjunk az efelé műveletekkel.

Gyakran lehet szükségünk arra, hogy az összes felsorolt értéket figyelembe vegyük, vagy egyszerűen listázzuk (pl. egy lenyíló listában szeretnénk megjeleníteni). Erre mutat példát az 5.4. kódrészlet, ahol ismét az *Enum* osztály megfelelő osztály szintű metódusait hívjuk segítségül. Lehetőségünk van a neveket, illetve magukat a felsorolt értékeket kinyernünk. Utóbbi gyakran hasznosabb, hiszen pl. egy lenyíló listában eltárolt hivatkozások átalakítás nélkül felhasználhatóak a későbbiekben, és minden nyelvi elem *String* reprezentációja egyébként is előállítható, és a legtöbb vezérlő megjelenítés során ezt kihasználja.

```
1  EdgeType[] edgeTypes = (EdgeType[])Enum.GetValues(typeof(EdgeType));
2  for (int i = 0; i < edgeTypes.Length; i++)
3  {
4    Console.WriteLine(edgeTypes[i] + " --> " + (int)edgeTypes[i]);
5  }
6
7  String[] enumStrs = Enum.GetNames(typeof(EdgeType));
8  for (int i = 0; i < edgeTypes.Length; i++)
9  {
10   Console.WriteLine(enumStrs[i]);
11 }
```

5.4. kód. Enum értékek és nevek listázása

[7] Figyeljünk arra is, hogy hibamentes kódot írjunk, sőt a mellettünk ülő másik szakember is hasonló hibamentes kódot írjon most is, és a következő 50 évben folyamatosan!

5.1.2. Kiegészítő metódusok alkalmazása

Miután megismerjük a felsorolás típus nyelvi lehetőségeit, talán joggal merül a kérdés: nem lehet mindezt még ennél is többre, avagy esetleg nem lehet biztonságosabban használni? A bemutatott példánál maradva, a Petri hálózat éltípusait mind-mind egyedi jelölésmód illeti meg, és gyakran szükség lehet arra is, hogy egy lokalizált felületen ne azt jelenítsük meg hogy INHIBITOR, hanem azt hogy "Tiltó él". Ezek a "mezők" mind-mind olyanok, melyek felelősség szempontjából az *enum* egyes értékeihez tartoznak. Milyen kár, hogy az egyes *enum* értékek nem egy valós osztály példányai, hiszen akkor el lehetne "őket" látni kiegészítő tulajdonságokkal, sőt akár műveletekkel is.

Van azért a C# nyelvben is némi lehetőségünk arra, hogy a kívánt célt részlegesen elérjük. Ehhez ún. "extension method"-ust szükséges készítenünk. Mindez bár az objektum-orinetált programozás "hőskorában" előforduló, ám ma már igen objektum-orientált idegen szintaktikával tud megvalósulni [8]. Egy "tagfüggvény" minden esetben valamely osztály részeként definiálandó. Ezt a felelősséget az oo világa nagyon komolyan veszi, és szintaktikailag legtöbb esetben az osztály blokkján belül definiáljuk az oda tartozó műveleteket. Azon típusok/osztályok esetén viszont, ahol vagy nincs erre szintaktikai lehetőségünk (pl. *enum*), vagy nem a mi kezünkben van az osztály kódja (pl. *String*), a "kiterjesztő metódusokat" hívhatjuk segítségül.

Az 5.5. kódrészlet bemutatja, hogy mi az az elvárt szintaktika, mely megfelelően tömör és egyértelmű ahhoz, hogy könnyen bele lehessen szeretni, ha egy *enum* értékhez tartozó kiegészítő információra van szükségünk. A felsorolt érték "példány szintű" metódusaként látszik a *GetLabel()* metódus, mely visszaadja az adott él típushoz tartozó lokalizált feliratot, illetve létezik egy *GetCustomOrdinal(int)* metódus is, mely egy bemenő paraméter függvényében egy módosított *ordinal* értéket szolgáltat.

```
1 SimpleEdgeType normalEdge = SimpleEdgeType.NORMAL;
2 Console.WriteLine("Label of " + normalEdge + " edge type: " + normalEdge.GetLabel());
3 Console.WriteLine("Custom ordinal of RESET: " +
      SimpleEdgeType.RESET.GetCustomOrdinal(5));
```

5.5. kód. Kiterjesztések használata

Implementáció szintjén az 5.6. kódrészlet tartalmazza az "extension method"-ok részleteit. Ezek a speciális metódusok "osztályszintűként" vannak definiálva, tipikusan (bár nem kötelezően) egy "statikus osztályban"[9]. A metódusok első paramétere a **this** kulcsszóval jelzi azt a típust, melyhez valójában tartoznak (a befoglaló osztály lényegtelen, a **static** kulcsszó valószínűsíthetően ezért fontos). Megjegyzendő még, hogy a példában létrehozott *EdgeTypeLabel* osztály helyett bármi más is elképzelhető, sőt szebb volna ezt az osztályt a *Singleton design pattern* (lásd. 7.1. fejezet) szerint elkészíteni.

```
1 public class EdgeTypeLabel
2 {
3   private Dictionary<SimpleEdgeType, String> map = new Dictionary<SimpleEdgeType,
        String>();
4
5   public EdgeTypeLabel()
6   {
7     this.map = new Dictionary<SimpleEdgeType, String>();
8     this.map.Add(SimpleEdgeType.NORMAL, "Normal el");
9     this.map.Add(SimpleEdgeType.INHIBITOR, "Tilto el");
```

[8] A C# hibrid nyelv, nem kizárólagosan objektum-orientált, így ezen nem kell meglepődnünk.

[9] Osztályszintű osztály létezik az objektum-orintált nyelvekben, de nem attól lesz egy osztály osztályszintű, mert minden metódusa osztályszintű (a C# viszont így értelmezi).

```
10      this.map.Add(SimpleEdgeType.RESET, "Reset el");
11    }
12
13    public String GetLabel(SimpleEdgeType type)
14    {
15      return this.map[type];
16    }
17  }
18
19  [..]
20
21  public static class EdgeTypeExtensions
22  {
23    public static String GetLabel(this EdgeType edgeType)
24    {
25      return new EdgeTypeLabel().GetLabel(edgeType);
26    }
27
28    public static int GetCustomOrdinal(this EdgeType edgeType, int param)
29    {
30      return ((int)edgeType * param);
31    }
32  }
```

5.6. kód. Kiterjesztések

5.1.3. Saját felsorolás típus definiálása

A kiterjesztő metódusok bár képesek arra, hogy példány szintű felelősséget adjanak egy *enum* értéknek, nem tudnak tényleges adatmezőt csatolni hozzájuk (ehhez külön típust kellett a példában is létrehoznunk *EdteTypeLabel* néven), és osztályszintű metódust sem tudnak definiálni (pl. egy *EdgeType.GetDefault()* metódus hasznos lenne, mely kirajzoláskor visszaadná az alapértelmezett él típust). Mit tehetünk akkor, ha minderről nem szeretnénk lemondani? Egyszerűen elfelejtjük, hogy létezik olyan *szintaktikai cukorka* a C# nyelvben, hogy *enum*, és létrehozunk egy típust, amin programozottan némi korlátozást vezetünk be. A cél az, hogy pontosan ugyanolyan egyszerűen, biztonságosan és azonos szintaktikával tudjunk egy felsorolás értéket használni, mint az *enum* esetében.

Ha alaposan áttanulmányozzuk az 5.7. kódrészletben bemutatott *EdgeType* osztályt, akkor a következő megállapításokat tehetjük.

- Az osztály egy olyan tovább nem származtatható (**sealed**) osztály, mely klónozás hatására kivételt dob (a *CloneNotSupportedException* saját kivétel osztály egy példányát).
- Konstruktora **private**, így csak és kizárólag saját belügye lehet új példányok létrehozása.
- Minden felsorolt érték egy nyilvános (**public**) konstans (**static readonly**) az osztályban, melyeknek automatikusan lekérdezhető neve (*Name* példány tulajdonság) illetve sorszáma/sorrendje (*GetOrdinal()* példány metódus).
- *ToString()* metódusa visszaadja a felsorolt érték karakterlánc reprezentációját (nevét).
- *Values* osztály szintű tulajdonságán keresztül címezhető bármely sorszámú eleme.
- Osztályszintű *Parse()* metódusa segítségével karakterláncból bármikor előállítható egy létező felsorolt érték referenciája.

```
1  public sealed class EdgeType : ICloneable
2  {
3    public static readonly EdgeType NORMAL = new EdgeType("NORMAL");
4    public static readonly EdgeType INHIBITOR = new EdgeType("INHIBITOR");
5    public static readonly EdgeType RESET = new EdgeType("RESET");
6
7    private static List<EdgeType> values;
8
9    private readonly String name;
10
11   public static EdgeType[] Values
12   {
13     get
14     {
15       return values.ToArray();
16     }
17   }
18
19   public String Name
20   {
21     get { return this.name; }
22   }
23
24   private EdgeType(String name)
25   {
26     this.name = name;
27     add(this);
28   }
29
30   private static void add(EdgeType item)
31   {
32     if (values == null)
33     {
34       values = new List<EdgeType>();
35     }
36     values.Add(item);
37   }
38
39   public int GetOrdinal()
40   {
41     return values.IndexOf(this);
42   }
43
44   public static EdgeType Parse(String name)
45   {
46     EdgeType ret = EdgeType.NORMAL;
47     EdgeType[] items = EdgeType.Values;
48     bool find = false;
49     int i = 0;
50     while ((i < items.Length) && (!find))
51     {
52       if (items[i].name.Equals(name))
53       {
54         ret = items[i];
55         find = true;
56       }
57       ++i;
58     }
59     return ret;
60   }
61
62   public override String ToString()
63   {
64     return this.name;
65   }
66
67   public Object Clone()
68   {
69     throw new CloneNotSupportedException();
70   }
71 }
```

5.7. kód. Felsorolás típus mint osztály

Mindezek pedig azt jelentik, hogy azon kívül, hogy az osztály referenciáit ugyanolyan kényelmesen tudjuk használni, mint a beépített felsorolás típust [10](lásd. 5.8. kódrészlet), egy olyan zárt halmazt hoztunk létre, mely kívülről nem bővíthető, biztonságos vezérlést ad számunkra.

```
1  EdgeType normal = EdgeType.NORMAL;
2  Console.WriteLine(normal + " --> " + normal.GetOrdinal());
3
4  EdgeType inhibitor = EdgeType.Values[1];
5  Console.WriteLine(inhibitor + " --> " + inhibitor.GetOrdinal());
6
7  EdgeType reset = EdgeType.Parse("RESET");
8  Console.WriteLine(reset + " --> " + reset.GetOrdinal());
9
10 foreach (EdgeType edgeType in EdgeType.Values)
11 {
12   Console.WriteLine(edgeType + " --> " + edgeType.GetOrdinal());
13 }
```

5.8. kód. Felsorolás "osztály" tesztelése

Mi az amit az így létrehozott *EdgeType* osztállyal meg tudunk tenni? Gyakorlatilag "bármit", amire szükségünk lehet:

- Bővíthetjük az osztályt példányszintű mezőkkel és metódusokkal, melyek segítségével kiegészítő tulajdonságokkal ruházhatjuk fel a típust (lásd. 5.9. kódrészlet, mely egy *felirattal* egészíti ki az osztály példányait).

```
1  public sealed class EdgeType : ICloneable
2  {
3    public static readonly EdgeType NORMAL = new EdgeType("NORMAL", "Normal el");
4    public static readonly EdgeType INHIBITOR = new EdgeType("INHIBITOR", "Tilto
         el");
5    public static readonly EdgeType RESET = new EdgeType("RESET", "Reset el");
6
7    private readonly String label;
8
9    [..]
10
11   public String Label
12   {
13     get { return this.label; }
14   }
15
16   private EdgeType(String name, String label)
17   {
18     [..]
19     this.label = label;
20   }
21   [..]
22 }
```

5.9. kód. Felirat hozzáadása

- Bővíthetjük példány és osztály szintű üzleti metódusokkal, melyek felelőssége egyértelműen a típushoz vagy példányaihoz kötődik (lásd. 5.10. kódrészlet, melyben egy osztályszintű metódus visszaadja az alapértelmezett él típus példányát).

```
1  public sealed class EdgeType : ICloneable
2  {
3    [..]
4    public static EdgeType GetDefault()
5    {
```

[10] természetesen nem azonos teljesítményben

```
6       return NORMAL;
7     }
8     [..]
9   }
```

5.10. kód. Alapértelmezett él típus

5.1.4. Attribútumok alkalmazása

A fejezet végén talán még megemlíthetőek a C# *attribútumok* (vagy más nyelvekben *annotációk*) használatának lehetősége, melyeket akár egy-egy *enum* értékhez is rendelhetünk, és ezáltal tudunk olyan "hatást" elérni, mintha a konstansokhoz tulajdonságokat definiálnánk (lásd. 5.11. kódrészlet). Az *attribútumok* használatával már elveszítjük az *enum valuetype* által nyújtott teljesítmény előnyöket, és messze nem olyan kényelmes a használatuk, mint a fejezetben bemutatott korábbi alternatíváké.

```
1  [AttributeUsage(AttributeTargets.Field)]
2  public sealed class EdgeTypePropAttribute : Attribute
3  {
4    private readonly string label;
5
6    public string Label
7    {
8      get { return this.label; }
9    }
10
11   public EdgeTypePropAttribute(String label)
12   {
13     this.label = label;
14   }
15 }
16 [..]
17 public enum EdgeType
18 {
19   [EdgeTypeProp("Normal el")]NORMAL,
20   [EdgeTypeProp("Tilto el")]INHIBITOR,
21   [EdgeTypeProp("Reset el")]RESET
22 }
23 [..]
24 public static EdgeTypePropAttribute GetAttribute(EdgeType edgeType)
25 {
26   return (EdgeTypePropAttribute)Attribute.GetCustomAttribute(ForValue(edgeType),
         typeof(EdgeTypePropAttribute));
27 }
28 public static MemberInfo ForValue(EdgeType edgeType)
29 {
30   return typeof(EdgeType).GetField(Enum.GetName(typeof(EdgeType), edgeType));
31 }
32 [..]
33 String label = GetAttribute(EdgeType.INHIBITOR).Label;
```

5.11. kód. Attribútumok használata

5.2. Egység tesztelés

Előfeltétel

osztályszintű felelősség, kivételkezelés

Egy fejlesztési feladat sokféleképpen "elkészülhet". Vannak természetesen elvárható minimális követelmények egy alkalmazással szemben, ilyenek pl. a szintaktikai helyesség, elindíthatóság, alkalmazás szerverhez készült komponensek esetén az adott konténerbe való hibanélküli telepítés [11] lehetősége. Ezek hiánya esetén értelmetlen bármi másról beszélnünk.

Ha a korábban felsorolt tételeknek megfelel már a programunk, eldönthetjük, hogy mikor fogjuk azt mondani, hogy "kész van"[12], vagyis mit tekintünk eme állapot "definíciójának"[13]. A teljesség igénye nélkül ide az alábbiak tartozhatnak (nem mindegyik, hiszen van amelyik részben fedi a másikat):

- **Review**: A fejlesztési feladatot átnézte és megjegyzéseivel látta el egy másik fejlesztő munkatárs. Az elfogadott megjegyzések kijavításra kerültek.
- **Validation**: A fejlesztési feladatot átnézte/ellenőrizte egy tesztelő munkatárs.
- **Unit test**: A fejlesztési feladat rendelkezik egység tesztekkel, melyek X%-ban lefedik a kódsorokat/vezérlési szerkezeteket.
- **Branch/Line coverage**: A fejlesztési feladat teszt lefedettsége X%, mely a kódsorokra és/vagy a futási ágakra vonatkozik.
- **Static production checks**: A termék kód[14] megfelel a közösen kialakított fejlesztési szabályoknak, és ez manuálisan vagy automatikusan ellenőrizve lett.
- **Static test checks**: A teszt kód megfelel a teszt kódra értelmezett közösen kialakított fejlesztési szabályoknak, és ez manuálisan vagy automatikusan ellenőrizve lett.
- stb.

A *review* egy nagyon jó módszer arra, hogy kiderüljön, az elkészült munka egy másik fejlesztő számára is könnyen és egyértelműen értelmezhető, a kód minősége rendben van és ha később az adott kódrészletet módosítani szükséges (igény változása vagy egy korábban nem várt jelenség által okozott hiba végett), akkor a módosításnak nincsen kiemelt kockázata. A kód "tisztaságára" (**clean code**[9]) elsősorban ez a módszer tud figyelni, és itt lehet látványos az is, ha egy része az alkalmazásnak részleges vagy teljes átírásra szorul (**refactor**).

[11] deploy, deployment
[12] Egy alkalmazás sosincs kész, legfeljebb abbahagyjuk a vele való foglalkozást.
[13] Definition of Done
[14] production code, a forráskód tesztek nélküli része

5.2.1. Felelősség

A fejezetben az egység tesztek (*unit test*) készítéséről és szükségességükről lesz szó, illetve a kapcsolódó lefedettség (*coverage*) témakörét is érinteni fogjuk. A *unit* teszt írása semmiben nem tér el szintaktikailag attól, ahogyan eddig algoritmusokat készítettünk egy adott cél megvalósítása érdekében. Általában (bár ez nem szükségszerű) ugyanazon a nyelven készülnek az egység tesztek, melyen a tesztelt elemek, osztályok is íródtak. A különbség abban áll, hogy mi a feladata az egyiknek, illetve a másiknak. A termék kód feladata az eredeti követelmények megvalósítása, míg az egység tesztek feladata a megvalósított vagy megvalósítandó termék kód eredeti követelménye szerinti validáció.

Az alkalmazásfejlesztés során nem elég az esetek "túlnyomó többségét" lefednünk, lekezelnünk (bár a gyakorlatban igen sokszor csak erre van idő). Hiába következik be egy esemény az adott alkalmazás életében nagyon ritkán, nekünk az esetek 100%-ára kell helyesen reagálnunk. Egy szükséges, de nem elégséges feltétel az, amikor egy alkalmazást saját magunk teszteljük, pl. futtatjuk és megnézzük hogy gyakori, vagy általános bemenetekre milyen eredményt ad. Az egység tesztek írásakor arra törekszünk, hogy minden lehetséges esetre készítsünk egy automatikusan futó és ellenőrizhető tesztet. Talán mondani sem szükséges: ezzel rengetek időt és energiát takarítunk meg a későbbiekben, illetve majd látni fogjuk, magabiztosan képesek leszünk az alkalmazásunkon bármilyen, akár átfogó átalakításokat is végezni.

Mielőtt azonban megismerkedünk e módszer alapjaival, egy fontos dolgot tisztáznunk kell: az egység tesztek megléte nem garantál számunkra semmit. Ha rossz egység teszteket készítünk, csak az időnket pazaroljuk vele. Rossz "termék kód" még eladható, mindenki tud erre példát hozni, az egység teszt önmagában viszont eladhatatlan. Ha csak azért készítünk egység teszteket, hogy "legyenek", azzal többet ártunk mint használunk. Jó egység tesztek írásához nem ugyanazok a fejlesztési módszerek és szabályok megismerése szükséges, amit a termék kód írásakor, annak algoritmusainak elkészítésekor már elkezdtünk megismerni. Vannak természetesen átfedések, de nem ugyanattól lesz az egyik és a másik "jó", avagy "tisza" kód.

A egység teszt projektben egység tesztek vannak, melyekben teszt metódusokat hozunk létre. Minden ilyen metódusnak csak és kizárólag kétféle kimenete lehet: az adott teszt elbukik (*failed*), vagy átmegy (*passed*) az aktuális követelményeken. Legelőször is érdemes felsorolni hogy mik a *unit* teszt írásának legfontosabb szabályai:

- Nagyon gyorsan végre kell hajtódnia egy egység tesztnek. Ez azt jelenti, hogy nem tehetünk bele blokkoló utasítást, nem várakozhatunk, nem indíthatunk el hosszú futás idejű folyamatot.
- Nem függhet az egység teszt futása semmilyen olyan folyamattól, melynek vezérlését az adott környezet nem tudja befolyásolni (pl. nem olvashatunk be állományt, mert nem tudjuk garantálni, hogy a beolvasott állomány biztosan elérhető lesz).
- Az egység tesztnek automatikusan kell tudnia futni. Nem állhat le manuális lépéssel, nem kérhet be adatot a felhasználótól.
- Egy bukó egység tesztnek egyértelműen ki kell tudnia jelölnie, hogy miért nem volt sikeres a futása, és hol van a hibásan működő komponens, programrész (ugyanez természetesen a sikeresen futó esetre is igaz, csak ott nem szoktuk keresni a "hiba okát"). Nagyon lényeges a tesztnek az elnevezése ebből a szempontból.

- Egyszerre lehetőség szerint csak egy dolgot teszteljünk, annyira aprólékosan, amennyire lehetőségünk van. A termék kódban megtalálható egyetlen metódushoz akár tucatnyi teszt kódot is készíthetünk [15].
- Az egység tesztnek teljesen izoláltnak kell lennie, kizárólag a tesztelt osztály kódjának szabad futnia a termékből (minden más osztály helyett "mock" vagy "fake" példányokat kell használni). Ha egy osztály tesztje elbukik, akkor egyrészt a programhibának ezen osztályban kell lennie (vagy a tesztjében), és nem szabad semmilyen másik osztály tesztjének elbuknia ugyanezen okból.
- Akármennyiszer futtatjuk le az egység tesztet, mindig ugyanazt az eredményt kell adnia (konzekvensen el kell buknia, vagy át kell mennie).
- Az egység tesztek egymástól sem függhetnek. Bármilyen sorrendben futtatjuk a rendelkezésre álló teszteket, azoknak mindig ugyanazt az eredményt kell adniuk.
- A redundáns teszt kód sokszor ugyanolyan hiba, mint a redundáns termék kód, azonban teszt kód esetén a redundancia hozzáadhat az olvashatósághoz, ezért megengedett lehet. A teszt kód algoritmusai legtöbbször faék egyszerűségűek. A jó teszt kódot ránézésére annak is tudnia kell értelmeznie, aki még kevésbé számít tapasztalt fejlesztőnek az adott területen.
- Ne akarjuk túlságosan "mélyen" letesztelni a konkrét implementációt, de ne is legyünk felületesek. Meg kell találni az egészséges középutat annak érdekében, hogy az egység teszt megvédje a kódot az új funkciók fejlesztése során szükségszerűen érkező programhibáktól, viszont lehetőséget biztosítson arra, hogy át tudjuk alakítani a termék kódot oly módon, hogy az átalakítás során az adott komponenshez készített egység tesztekhez nem nyúlunk hozzá (*overspecification*).

Akkor érjük el a célunkat, ha csak és kizárólag akkor lesznek bukó *unit* tesztjeink, ha az eredeti követelmények megváltoznak (és ezért pl. kódmódosítást hajtunk végre, mely értelemszerűen a teszt kód módosítását is érinteni fogja), illetve ha új, eddig ismeretlen programhiba lép fel. Tekintsünk úgy az egység tesztre, mint egy és szétválaszthatatlan kódrészére a lefejlesztett funkciónak. Ha egy módosítás öt osztályt érint a termékben, akkor számoljunk tíz osztály módosításának költségével[16].

Az egység teszt írása során úgy kell az implementációra, a tesztelt kódra tekintenünk, mint ha az egy fekete doboz lenne számunkra (és ez kiemelten azért nehéz, mert az egység tesztet az a fejlesztő írja, aki a termék nevezett részét is fejlesztette/fejleszti). A termék kód nyilvános, kifelé látható elemeit teszteljük, ne a vezérlési szerkezeteket és a **private/protected** metódusokat. Ez tudja garantálni azt, hogy a követelményeknek, és ne az implementációnak feleljen meg a teszt. Utóbbi megfontolásból létezik komoly szakirodalma annak, hogy az egység teszteket írjuk meg először, és csak később az implementációt. Ezt nevezzük a *Test Driven Development* (TDD) módszerének.

5.2.2. Egység tesztek készítése

Eddig is készítettünk "teszteket", csupán nem formalizáltuk azokat, illetve nem törekedtünk a teljes bementi halmaz lefedésére, megelégedtünk egy általános, avagy egy ti-

[15] Persze ez esetben már érdmes elgondolkodni azon, hogy a termék kód biztosan a legjobban van-e szervezve, hiszen gyanúsan sok a felelőssége.

[16] Költség alatt itt nem egy lineáris képletet értve. A fejlesztési folyamatok idő és erőforrás költsége nem azonos azzal, ahogyan a gyertyagyárban a futószalagon a paraffin tölcsérek készülnek.

pikus futási ág működésének validációjával. Mindennek elméleti alapjaival a 2. fejezetben kezdtük el foglalkozni, és fokozatosan építettük fel az eddig elkészített alkalmazásainkat oly módon, hogy azok könnyen készek legyenek egység tesztekkel való kiegészítésre. Amit mellőzni fogunk, az a *randomizáció* alkalmazása. Itt most nem szimulációt készítünk, hanem validációt.

A C# nyelvhez (is) számos külső *library* létezik egység tesztelésre. A jegyzetben a **Microsoft Unit Testing Frameworkkel** ismerkedünk meg alaposabban, mert ez része a *Visual Studionak*. E keretrendszer nem képes ún. "mock" vagy "fake" elemek létrehozására, ezért e téren "kézzel" fogunk lépéseket tenni, amikor szükség lesz ilyesmire[17].

Visual Studioban az egyes projekteket egy *Solution* alá lehet szervezni, ezáltal jelezni, hogy "van közük egymáshoz". Eleddig egy célfeladatra létrehozott *Solution* egyetlen projektet tartalmazott ("Visual C#/Windows/Console Application" sablonból kiindulva), ami alá a forrásainkat és erőforrásainkat szerveztük. Egy adott projekthez készíthetünk egy "kiegészítő projektet" ("Visual C#/Test/Unit Test Project" sablonból kiindulva), mely az egység teszteket tartalmazza a hivatkozott projektre nézve[18]. Az új projekt referenciáihoz hozzá kell adni a tesztelendő projektet[19]. Alapértelmezés szerint az egy *Solution* alatt megtalálható projektek nem látják egymást (ez így természetes).

Az egység teszteket tartalmazó projektben elvben bármilyen osztály és felelősség elfér, szintaktikailag nem vagyunk korlátozva. Belépési pontként szolgáló *Main()* metódust azonban nem kell hozzá készíteni, mert futtatni a teszteket a *Test Explorerből*[20], vagy az adott teszt osztály/teszt metódus helyi menüjének "Run Tests" parancsával fogjuk.

Egy metódus akkor lesz "egység teszt", ha rendelkezik a [TestMethod] attribútummal, nyilvános, nincs visszatérési értéke és paramétere, illetve olyan osztályban található, mely rendelkezik a [TestClass] attribútummal (lásd. 5.12. kódrészlet).

```
1  [TestClass]
2  public class SampleTest
3  {
4    [TestMethod]
5    public void Sample_test_method()
6    {
7      double i = Math.PI * Math.E * 4.55;
8      Assert.AreEqual(42, i, 3, "The number " + i + " is not close enough for the meaning
           of life!");
9    }
10 }
```

5.12. kód. Egység teszt minta

A teszt metódusokban általában az *Assert*, a *StringAssert* és a *CollectionAssert* osztályok statikus metódusait használjuk validációra, illetve néhány attribútum lesz még gyakori. Ezek mindegyike a "Microsoft.VisualStudio.TestTools.UnitTesting" névtér alatt található meg. Az osztálykönyvtár részletezése nem célja a jegyzetnek, példának okáért

[17] Számos *mock framework* létezik a C# nyelvhez, gyakran említik szakirodalomban a **Moq** névre hallgató keretrendszert.

[18] *Solution Exlorerben* a *Solution* helyi menüjéből válasszuk ki az "Add/New Project..." parancsot, majd a megjelenő "Add New Project" dialógus ablakban válasszuk ki a "Visual C#/Test/Unit Test Project" sablont.

[19] *Solution Explorerben* a kiválasztott *Unit Test Project* "References" részén helyi menüből válasszuk ki az "Add Reference..." parancsot, majd a megjelenő "Reference Manager" dialógus ablakban válasszuk ki a "Solution" alatt megtalálható tesztelendő "Console Application" sablon szerint létrehozott projektünket.

[20] A *Test Explorer* a tesztek futásakor automatikusan előugrik, de a "Test" főmenü "Windows/Test Explorer" menüpontjával előhívható.

az 5.12. kódrészlet 8. sorában használt *AreEqual()* metódusnak számos *overload* változata létezik, az aktuálisan használt metódus az első paraméterben megadott elvárt értéket (*expected*) hasonlítja a második paraméterként átadott aktuális értékhez (*actual*) a harmadik paraméterben definiált *delta* közelítéssel. Ha a számolt érték 39-nél kisebb, vagy 45-nél nagyobb, akkor a negyedik paraméterben megadott hibaüzenet megjelenítése mellett a teszt el fog bukni (a példában ez történik).

Az 5.12. kódrészletben bemutatott példa természetesen nem tesztel semmit, csak szintaktikát mutat. Egy egység teszt metódus egy termék kódon ellenőriz valamilyen működést, nem pedig helyben kiszámolt értéken. Az 5.13. kódrészlet egy egyszerű "termék" kódját mutatja, melyben két egész szám hányadosát számoljuk ki (van benne egy szándékos értelmetlenség, miszerint *hetet* sosem tud eredményül adni az algoritmus).

```
1  public class Calculator
2  {
3
4    public static int CalculateIntegerDivision(int divident, int divisore)
5    {
6      int result = divident / divisore;
7      if (result == 7)
8      {
9        result += 1;
10     }
11     return result;
12   }
13
14 }
```

5.13. kód. Egész számok osztása némi csavarral

A metódushoz készített egység teszt meghívja két olyan paraméterrel a függvényt, melynek eredménye fejben is könnyen kiszámolható (42 / 10 = 4), majd ellenőrzi, hogy a visszaadott érték megfelel ennek az elvárt értéknek (lásd. 5.14. kódrészlet).

```
1 [TestMethod]
2 public void Integer_division_shuold_be_working()
3 {
4   int result = Calculator.CalculateIntegerDivision(42, 10);
5   Assert.AreEqual(4, result);
6 }
```

5.14. kód. Általános teszt eset

Érdemes egy ilyen egyszerű példánál felhívni a figyelmet arra - az egyébként nem olyan ritka - jelenségre, amikor az elvárt értéket hasonló, vagy ugyanolyan algoritmussal számoljuk ki, mint amit egyébként az implementációban is használtunk. Erre mutat egy példát az 5.15. kódrészlet: fejben egyrészt legtöbben nem tudjuk kiszámolni, hogy a művelet eredménye 17343, másrészt ha eme számot írnánk a kódban az *AreEqual()* első argumentuma helyébe, kicsit "titkos üzenet" lenne, mit is ellenőriz a kód valójában. A példában nagyon feltűnő, hogy ugyanazt a műveletet végezzük el a tesztben és a termékben, azonban ez nem mindig lesz így. Nem szabad abba a hibába esni, hogy elkezdnünk "üzleti kódot algoritmizálni" a tesztben, függetlenül attól, hogy azért vannak olyan műveletek, melyekben bátran megbízhatunk egy teszt kódsoraiba írva is (pl. pont ilyen az egész osztás művelete).

```
1 [TestMethod]
2 public void Integer_division_shuold_be_working2()
3 {
4   int result = Calculator.CalculateIntegerDivision(75564242, 4357);
```

```
5   Assert.AreEqual(75564242 / 4357, result);
6 }
```

5.15. kód. Fejtörős teszt eset

Az 5.13. kódrészlet tesztelése azonban még nincsen készen, hiszen van olyan - tegyük fel hogy eredeti követelmények szerint tisztázott - követelmény, melyet még nem teszteltünk le: nevezett esetben az algoritmus *hetet* sosem tud visszaadni, ilyenkor egyel nagyobb szám lesz az eredmény. Ellenőrizni is tudjuk e teszttel még lefedetlen részeket, ha az adott projekthez/osztályhoz készítünk egy *Coverage* riportot[21]. A kód lefedettségének ellenőrzése során az ellenőrzött kódrészek háttérszíne megváltozik a *Visual Studioban*. A *zöld* háttérrel rendelkező részeket érintik a tesztek, míg a *piros* háttérrel rendelkezőeket nem. Fontos tudni, hogy attól, hogy egy osztály 100%-ban lefedett, nem lesz garantálva a minőség és a hibabiztosság. Nem jelenti azt, hogy a tesztek, melyek "végigfutnak" rajta, valóban hasznosak is, ellenőrzik-e megfelelően, illetve a készült egység tesztek megfelelnek-e a *unit* tesztek elméleti szabályainak. Kezeljük óvatosan az így kapott százalékos értékeket, és ne az alapján minősítsük a kódot!

A lefedetlen részekhez könnyen készülhet teszteset, ahogyan ez pl. az 5.16. kódrészletben látható, és ezzel előáll a 100%-os *coverage* érték, mindamellett, hogy még mindig van olyan szituáció, melyre az adott kódot nem ellenőriztük le!

```
1 [TestMethod]
2 public void Integer_division_shuold_be_working_in_the_special_case()
3 {
4   int result = Calculator.CalculateIntegerDivision(42, 6);
5   Assert.AreEqual(8, result);
6 }
```

5.16. kód. A "csavar" tesztesete

Mi történik akkor, ha nullával szeretnénk elosztani egy egész számot? Mivel az osztás műveletet most nem mi készítettük el (bár ez ugye fekete doboz a számunkra, mikor az egység tesztet készítjük), ezért egy *DivideByZeroException* példány dobását várjuk el. Ne csak elvárjuk, teszteljük is le! Az 5.17. kódrészletben bemutatott új teszteset akkor lesz "Passed", ha futása során az [ExpectedException] attribútumban megadott kivétel dobódni fog. A 6. sorban meghívott *Assert.Fail()* utasításnak helyes működés esetén nem szabad lefutnia [22]!

```
1 [TestMethod]
2 [ExpectedException(typeof(DivideByZeroException))]
3 public void Divide_by_zero_should_failed()
4 {
5   int result = Calculator.CalculateIntegerDivision(42, 0);
6   Assert.Fail("Divide by zero didn't cause failure!");
7 }
```

5.17. kód. Nullával való osztás tesztesete

A bemutatott egység tesztek nagyon egyszerűek, és bizonyosan elsőre nem látszik a hasznuk, azonban a későbbiekben összetettebb problémák esetén is alkalmazni fogjuk a

[21] A "Test" főmenü "Analyze Code Coverage/All Tests" parancsát futtatva a megjelenő *Code Coverage Results* ablakban keressük ki a *Calculator* osztályt, és nézzük meg a táblázatban leírt százalékos eredményeket.

[22] Bár értelmetlen, de lehet "Code Coverage" riportot készíteni a teszt projekt osztályaira is. Ez esetben látni fogjuk, hogy a nevezett teszt metódusban az *Assert.Fail()* sor mindig piros háttérrel fog látszódni, ha az üzleti kód helyesen dobta az elvárt kivételt.

technikát. A *unit* tesztek meglétének egyik, ha nem a legnagyobb előnye a következő: ha egy osztálykönyvtárhoz elég nagy mennyiségben és minőségben készülnek egység tesztek, akkor az osztálykönyvtár bármely komponensét nagyon magas biztonsággal át lehet írni (**refactoring**) annak érdekében, hogy pl. átláthatóbb, tisztább legyen, vagy egyszerűen lecserélhetjük egy már régóta használt - nem általunk írt - alkomponenst egy másikra, vagy ugyanannak egy magasabb verziójára. Természetesen ezt olyan környezetben is megtehetjük és megtesszük, ahol nincsenek egység tesztek, azonban ott számolni kell az ezernyi új programhiba miatt fellépő minőség és hírnév romlással.

5.3. Generikusság

Előfeltétel

programozási tételek, tömbök, overloading, osztályszintű metódusok, öröklés, többalakúság

A generikus típusok megjelenése a C++ template technikával kezdődött, majd jelent meg a Java illetve a C# nyelvben. A generikus lista és szótár elemek gyors elterjedése egy ismert, ám kevésbé reklámozott szituációt hozott előtérbe: számos fejlesztő használ generikus típusokat, hiszen ezek megkönnyítik, sőt sokszor javítják a kód karbantarthatóságát, azonban sokan közülük nem ismerik mindennek a hátterét. Egy nyelv lehetőségeinek bővülésekor általában átlátjuk, hogy az adott szintaktikai cukorka nélkül milyen szintaxist kellene használnunk ugyanazon cél elérésének érdekében. Egy *foreach* ciklusnál általában tudjuk, hogy mögötte egy *iterator design patternre*[6] épülő megoldás található, és az *indexer* tulajdonság használatakor is az érdeklődők hamar rátalálnak és megértik a hátteret. A generikus típusoknál is bizonyosan sokan belekezdenek a háttér megismerésébe, ám egy részük elveszik a részletekben, és feladja. Ellenben felhasználni a nyelv részeként létező generikus típusokat nem fogja mellőzni! Miért áll elő ez a furcsa, kissé szakmaiatlan helyzet? Leginkább azért, mert a generikusság során nem a nem-generikus típusok szintjén futó osztályokat írunk, hanem utasításokat adunk a futtató környezetnek arra, hogy adott szabály szerint nem-generikus típusokat hozzon létre helyettünk. Ezt a több szinten zajló gondolkodásmódot nem mindenki látja át első körben[23].

5.3.1. Generikusság a Java és a C# nyelvekben

A generikus típusok által állított kihívásokhoz mindezek mellé hozzájön a köztes kódra forduló nyelvek esetén a generikusság értelmezésének színtere. A Java 1.5 és a C# 2.0 nyelvek vezették be a generikus típusokat, és pl. egy generikus lista adatszerkezetének felhasználása a két nyelven nagyrészt azonos szintaxissal valósul meg [24], a mögöttes háttér azonban eltérő!

A Java generikusság úgy lett tervezve, hogy egy generikus Java kód gond nélkül értelmezhető legyen egy Java 1.5 előtti JDK-n is, vagyis a *byte kód* szintjén a generikusság már nem jelenik meg (a fejezet végén mindez érthetőbb lesz), a Java *generics* ún. "compile-time feature", egy *List<String>* típust a JVM gyakorlatilag *List*-nek látja. Mindennek számos előnye mellett (komolyabb megszorítási lehetőségek, típus biztosabb kód, castolási lehetőség nem generikus és a generikus világ között, stb.) természetesen hátránya is van (kevésbé rugalmas, teljesítménye kevésbé optimalizálható).

A C# során a .NET framework tud a generikus típusokról, és értelemszerűen ebből az is következik, hogy a korábbi nem generikus elemek és a generikus társaik egyidőben léteznek a *managed kód* szintjén, vagyis ezek egymástól különböző osztályok (komolyabb castolási képességek a két világ között ezáltal megszűnnek). Ugyanazon generikus típus,

[23] Pedig ennek a gondolkodásmódnak a fejlesztése kiemelten fontos a mai trendek szerint. Egy-egy fejlesztési környezetben sokszor 4-5 szintű fordítás történik, mire pl. egy java virtuális gép megkapja azt a byte kódot, amit értelmezni tud. A JVM szempontjából teljesen lényegtelen, hogy a Java kódot (melyet le kell fordítani) ember írta, vagy egy gép generálta le olyan szabályok alapján, melyet egy másik szintaxis szerint vezérelt egy ember.

[24] Apróbb különbségek vannak generikus metódusok, osztályok definiálásakor, de a szintaxis könnyen felismerhető és átjárható a két nyelv között.

ha csak a generikus paraméterében különbözik egymástól, közös JIT kóddal rendelkezik (mely hatékony), ugyanez a Java világában nem elképzelhető (minden új generikus paraméter felmerülésekor egy új típus/metódus keletkezik a byte kód szintjén). A C#-ban van lehetőségünk *value type*-okat is alkalmazni generikus paraméterként, a Java-ban a primitívek estén ezt nem oldották meg (a C# az érték típusokra külön JIT kódot tart fenn, vagyis ezek között nem lesz kód megosztás). A C# *generics* "runtime feature".

Mindkét nyelv fejlődik a generikusság kezelésének terén (pl. a Java reflection API tartalmaz metódusokat, mely a generikus típus paraméterek lekérdezhetőségét támogatja *runtime*), azonban az alapok különbözősége miatt egy-egy speciálisabb esetben várhatóan még sokáig el fognak térni. A szerző véleménye az, hogy a generikus és a nem generikus világ teljes szétválasztása sok esetben hátrány, és bár a Java világa e téren látszólag kevesebbet tud, könnyebb integrálni, és elfedni az implementáció szintjén. Ugyanis egy dolog nagyon lényeges: a generikusság nem azt jelenti, hogy mostantól a kliens kódnak, vagy egy konkrét üzleti célt kiszolgáló *interface*-nek generikusnak kell lennie! A generikusság nagyon nagy rugalmasságot tesz lehetővé, azonban ez nem cél a legtöbb üzleti követelmény számára.

5.3.2. A tömbök és a generikusság

Annyira kézenfekvő a tömb típus létezése egy magas szintű nyelvben (hiszen a legalapvetőbb *programozási tételek*[1] is sorozatokra épülnek, mivel a legtöbb számítástechnikai probléma a sorozatok feldolgozásán alapszik, lévén ez az, amiben a gépek jobbak az embereknél), hogy sokszor nem vesszük észre, hogy a tömb egy nyelvi szinten definiált, alapvetően generikus típus (miközben a generikusság a magas szintű nyelvekben sokkal később jelent meg). A tömb háttere természetesen teljesen más (rendkívül hatékony és gyors, "összehasonlíthatatlan" a teljesítménye a generikus típusokkal), azonban felhasználás során valami hasonlót viszünk végbe: ha nem tudnánk definiálni a tömb alapelemének típusát, akkor minden tömb csak *System.Object* referenciákat tartalmazhatna, és a legalapvetőbb műveletek során is át kellene alakítanunk az elemeket használat előtt. Mindez a *type-safe* kódolásnak a tökéletes ellentéte lenne, ami egy karbantartható és tiszta kód[9] előállítását nem tenné lehetővé. A nem generikus dinamikus lista típusok (pl. *ArrayList*) pontosan ezt nyújtják (lásd. 4.1.4. fejezet).

5.3.3. Tömb elemek megjelenítése

Képzeljünk el egy szituációt, melyben a szakmabeli - de nem fejlesztő - munkatárs az alábbi igényt fogalmazza meg egy fejlesztő felé: "Szükségünk volna szám-név párokra, melyeket az alkalmazás egy meghatározott pontján szeretnénk kinyomtatni.". A fejlesztő megrágja a feladatot, majd jelzi hogy itten készülni fog egy entitás, mely egy számot és egy nevet egységbe zár (lásd. 5.18. kódrészlet[25]), majd készül egy üzleti *PrintArray* osztály, melynek felelőssége lesz egy sorozatnyi szám-név pár "kinyomtatása" (lásd. 5.19. kódrészlet).

```
1 public class TestEntity
2 {
3   private int id;
4   private String name;
```

[25] Ilyesmi célra a C# *Tuple* típusa legtöbbször elegendő volna, de ez most elrontaná a szituációs játékot (Tuple<int, String> pair = new Tuple<int, String>(42, "Hello");).

```
5
6   public TestEntity(int id, String name) {
7     this.id = id;
8     this.name = name;
9   }
10
11  public override String ToString() {
12    return this.Name + " (" + this.Id + ")";
13  }
14 }
```

5.18. kód. Entitások

```
1 public class PrintArray
2 {
3   public static void Print(TestEntity[] input) {
4     foreach (TestEntity element in input) {
5       Console.WriteLine(element);
6     }
7   }
8 }
```

5.19. kód. Szám-név párok listázása

Az igénylő elfogadja az elkészült implementációt, és megelégedéssel tekint az 5.20. kódrészletben bemutatott *PrintArray.Print()* metódus felhasználására. Tegyük fel, hogy pontosan erre volt szüksége, erre gondolt amikor megfogalmazta igényét a fejlesztő felé.

```
1 TestEntity[] testEntities = new TestEntity[3];
2 testEntities[0] = new TestEntity(10,"alma");
3 testEntities[1] = new TestEntity(20,"korte");
4 testEntities[2] = new TestEntity(30,"szilva");
5
6 PrintArray.Print(testEntities);
```

5.20. kód. Szám-név párok létrehozása és kinyomtatása

Kis idő elteltével a kolléga ismét megkeresi a fejlesztőt, hogy új igények merültek fel, és van olyan, mikoron az azonosító nem egész szám, hanem valós, illetve a név csupán egy karakter, sőt az is előfordul, hogy mindkét mező karakterlánc. El is készülnek az új entitások (lásd. 5.21. kódrészlet), és jön a fejlesztőtől a viszont kérdés: "Sajnos a kinyomtatás műveletét is ki kell egészíteni a változás végett, és ez bizony az eredeti nyomtatás fejlesztési költségének akár többszöröse is lehet".

```
1 public class DemoEntity
2 {
3   private double id;
4   private char name;
5
6   public DemoEntity(double id, char name) {
7     this.id = id;
8     this.name = name;
9   }
10
11  public override String ToString() {
12    return this.Name + " (" + this.Id + ")";
13  }
14 }
15
16 public class DummyEntity
17 {
18   private String id;
19   private String name;
20
21   public DummyEntity(String id, String name) {
22     this.id = id;
```

```
23      this.name = name;
24    }
25
26    public override String ToString() {
27      return this.Name + " (" + this.Id + ")";
28    }
29  }
```

5.21. kód. Entitások

Miért mondta mindezt a fejlesztő? És miért érte az igénylőt a plusz fejlesztési költség váratlanul? Az első kérdésre a válasz egyszerű, a fejlesztő nem ismeri a generikus típusokat, a második kérdésre a választ azonban nem a kódban kell keresnünk: az igénylő nem kért új üzleti funkciót, a nyomtatás képességét pedig végképp nem kívánta módosítani. Részéről jogos a "felháborodás", miért kell a nyomtatást ezúttal újraimplementálni? Ha megnézzük az 5.22. kódrészletet, látványossá válik, hogy a *DemoEntity* és a *DummyEntity* létrehozása új elemek a kódban (ennek költségével számol az igénylő), azonban a nyomtatást "elvárja", hogy ugyanúgy működjön (lásd. 17. és 18. sor).

```
1  TestEntity[] testEntities = new TestEntity[3];
2  testEntities[0] = new TestEntity(10,"alma");
3  testEntities[1] = new TestEntity(20,"korte");
4  testEntities[2] = new TestEntity(30,"szilva");
5
6  DemoEntity[] demoEntities = new DemoEntity[3];
7  demoEntities[0] = new DemoEntity(10.0,'a');
8  demoEntities[1] = new DemoEntity(20.0,'k');
9  demoEntities[2] = new DemoEntity(30.0,'s');
10
11 DummyEntity[] dummyEntities = new DummyEntity[3];
12 dummyEntities[0] = new DummyEntity("10","alma");
13 dummyEntities[1] = new DummyEntity("20","korte");
14 dummyEntities[2] = new DummyEntity("30","szilva");
15
16 PrintArray.Print(testEntities);
17 PrintArray.Print(demoEntities);
18 PrintArray.Print(dummyEntities);
```

5.22. kód. Tömbök listázása overload metódusokkal

A fejlesztő a cél érdekében igen redundáns kódot készít, ahogy ezt az 5.23. kódrészlet mutatja. *Overload* metódusok segítségével eléri ugyan, hogy az igénylő számára megfelelő "kliens kód" legyen leszállítva, azonban a kód redundáns, és fejlesztése plusz, nem várt költségekkel járt.

```
1  public class PrintArray
2  {
3    [..]
4
5    public static void Print(DemoEntity[] input) {
6      foreach (DemoEntity element in input) {
7        Console.WriteLine(element);
8      }
9    }
10
11   public static void Print(DummyEntity[] input) {
12     foreach (DummyEntity element in input) {
13       Console.WriteLine(element);
14     }
15   }
16 }
```

5.23. kód. Tömbök listázása

A redundáns kód nem pusztán valamiféle ízlés kérdésköre. Az objektum-orientáltság tűzzel vassal üldözi, és a legtöbb (ha nem szinte az összes) szintaktikai cukorka illetve nyelvi elem azért létezik, hogy redundáns kódot sehova se kelljen készíteni. Ugyanis a redundáns kóddal a "legjobb" úton haladunk a karbantarthatatlan, javíthatatlan kód felé. Ha egy hiba javítása egy redundáns kódrészlet nem minden előfordulásában lesz korrigálva, akkor ezt követően olyan borzasztó nehezen megtalálható, illetve javítható hibák lesznek a rendszerben, mely a minőséget és a végfelhasználó megelégedettségét alapvetően fogja meghatározni[26].

A megoldást a *Print* metódus generikus implementációja szállítja (lásd. 5.24. kódrészlet). Pontosan azokat a részeket "cseréljük" ki egy generikus típus paraméterre (a példában 'E'-re), ami a korábbi redundáns implementáció "különbözősége" volt. Vagyis elérjük a redundancia mentes kódot az adott szituációban.

```
1 public class GenericPrintArray
2 {
3   public static void Print<E>(E[] input) {
4     foreach ( E element in input ) {
5       Console.WriteLine(element);
6     }
7   }
8 }
```

5.24. kód. Tömbök listázása generikus metódussal

Mindez nem pusztán azt jelenti, hogy ezt követően egy negyedik és ötödik entitás bevezetésével nem kell a nyomtatás műveletére plusz erőforrást biztosítani, hanem a fejlesztés módszertanában kicsit közelebb került egymáshoz az "igény" és az "implementáció". Utóbbi minél szorosabb összekötése vezethet egyszer majd oda, hogy egy forráskódban csak az üzleti szabályokra kelljen koncentrálni[27].

```
1 public static void GenericPrintArrayTest() {
2   [..]
3
4   GenericPrintArray.Print(testEntities);
5   GenericPrintArray.Print(demoEntities);
6   GenericPrintArray.Print(dummyEntities);
7 }
```

5.25. kód. Tömbök generikus listázása

Vessünk egy pillantást az 5.25. kódrészletre. Miben különbözik egymástól a generikus és a nem generikus megoldás "kliens kódja"? Ne keresgéljük a különbségeket, ugyanis nincsenek! Attól, hogy az implementáció mélyén a megoldás során a generikusság lett alkalmazva annak érdekében, hogy redundancia mentes, általános megoldást kapjunk, nem jelenti azt, hogy az "igénylő" hirtelen máshogy várja el a művelet meghívását! Érdemes a generikus típusok alkalmazása során ezt észben tartani.

[26] Gondoljunk bele, mi az idegesítőbb: a programban találunk egy hibát, melyet a következő verzióban javítanak, és ennek örülünk. Vagy a programban találunk egy hibát, mely a következő verzióban már néha működik, de még évekkel később is előjön, és eszünkbe juttatja a jelenséget. A szerző biztos benne, hogy találkozott már az olvasó olyan levelezőprogrammal, vagy szövegszerkesztővel, mely évtizedes fejlesztés után is tartalmaz olyan hibajegyet, mely néha előjön, olykor pedig nem.

[27] Mindezt ekkor feltehetően már nem fejlesztő mérnökök fogják végezni, hanem azon milliók, akiknek már ma is igényük volna "fejleszteni", de megtanulni nincs idejük. A fejlesztő mérnökök továbbra is a saját szakterületükön maradva azon programokat/keretrendszereket fogják írni, melyek mindezt lehetővé teszik.

5.3.4. Generikus osztályok

Készítsünk egy kutyakozmetikát modellező osztályt, melyben sorban ülnek a kutyák, és amikor a "fodrász" néven szólítja valamelyik háziállatot, az belül a kés alá, és meghatározott cm-el kisebb lesz a szőre. Ugyanebben az alkalmazásban készítsünk egy kisállat-kereskedést is, mely kizárólag kutyákkal vagy macskákkal foglalkozik. Minden háziállatnak lehessen beállítani a "tulajdonosát", amikor valaki hazaviszi (megvásárolja).

A két problématér természetesen eltér egymástól, azonban van egy közös viselkedésük, melyet örökléssel fogunk megvalósítani: mind a kisállat-kereskedés, mind pedig a kutyakozmetika egy sornyi kisállatot tart karban. Természetesen utóbbi kizárólag kutyákkal foglalkozik, és itt jön a képbe egy generikus ősosztály.

Hozzuk létre a kisállatokat reprezentáló osztályokat. Minden *macska* (lásd. 5.26. kódrészlet) és *kutya* (lásd. 5.27. kódrészlet) *kisállat* (lásd. 5.28. kódrészlet), ahogyan ezt az öröklési láncuk is mutatja. Mindenkinek van egy neve illetve egy gazdija, a kutyák esetén ezeken kívül még a szőrük nagyságát is karbantartjuk, hogy a fodrász majd tudjon igazítani rajta (lásd. 5.27. kódrészlet *CutHair()* metódus).

```
1  public class Cat : Pet
2  {
3    public Cat(String name) : base(name) { }
4
5    public override string ToString()
6    {
7      return "Cat" + base.ToString();
8    }
9  }
```

5.26. kód. Macska

```
1  public class Dog : Pet
2  {
3    private int lengthOfHairs;
4
5    public Dog(String name)
6      : base(name)
7    {
8      this.lengthOfHairs = 10;
9    }
10
11   public void CutHair(int length)
12   {
13     this.lengthOfHairs -= length;
14   }
15
16   public override string ToString()
17   {
18     return "Dog" + base.ToString() + " length of hairs: " + this.lengthOfHairs;
19   }
20 }
```

5.27. kód. Kutya

```
1  public class Pet
2  {
3    private readonly String name;
4    private String owner;
5
6    public String Name
7    {
8      get { return this.name; }
9    }
10
```

```
11    public String Owner
12    {
13      get { return this.owner; }
14      set { this.owner = value; }
15    }
16
17    public Pet(String name)
18    {
19      this.name = name;
20    }
21
22    public override String ToString()
23    {
24      return " " + this.name + (this.owner != null ? " (owner: " + this.owner + ")" : "");
25    }
26  }
```

5.28. kód. Kisállat

Egy generikus ősosztályt fogunk definiálni a kisállat-kereskedés és a kutyakozmetika számára. Ezt mutatja be a *PetHolder* **abstract** osztály (lásd. 5.29. kódrészlet). Az osztálynak van egy generikus T paramétere, melyet egy megszorítással is ellátunk: kisállatnak kell lennie (**where** T: Pet). Az osztály lehetőséget biztosít kisállatok tárolására és név alapján történő keresésére (a *Find()* metódusban a T generikus típusnak azért van *Name* tulajdonsága, mert a fordító is tisztában van vele, hogy T legalább "kisállat", és minden kisállatnak van neve).

```
1  public abstract class PetHolder<T> where T : Pet
2  {
3    private List<T> animals;
4
5    public PetHolder()
6    {
7      this.animals = new List<T>();
8    }
9
10   public void Add(T animal)
11   {
12     this.animals.Add(animal);
13   }
14
15   public T Find(String name)
16   {
17     T result = null;
18     foreach (T animal in this.animals)
19     {
20       if (animal.Name.Equals(name))
21       {
22         result = animal;
23         break;
24       }
25     }
26     return result;
27   }
28 }
```

5.29. kód. Kisállat kezelő

Először a kisállat-kereskedés implementációját készítsük el. A *PetShop* egy olyan leszármazott *PetHolder*, mely a generikus T paramétert továbbviszi (lásd. 5.30. kódrészlet)! A *Buy()* metódusa a paraméterben kapott kisállatot kikeresi a "raktárból", majd hozzárendeli az új gazdiját.

```
1  public class PetShop<T> : PetHolder<T> where T : Pet
2  {
3    public void Buy(String name, String owner)
4    {
```

```
5       this.Find(name).Owner = owner;
6     }
7   }
```

5.30. kód. Kisállat-kereskedés

Mindez ebben a formában meg tudott volna valósulni akkor is, ha az ősosztály nem generikus, és egy *List<Pet>* mezőt tartalmaz. Azonban követelmény volt, hogy egy kisállat kereskedés kizárólag kutyákkal vagy macskákkal foglalkozik, és itt már ki tudjuk használni a *type-safe Add()* metódust! A macskákkal foglalkozó bolt példányosítására egy példát az 5.31. kódrészlet mutat be.

```
1 public static void TestShop()
2 {
3   PetShop<Cat> shop = new PetShop<Cat>();
4   shop.Add(new Cat("Cirmos"));
5   shop.Add(new Cat("Filemon"));
6   shop.Buy("Filemon", "Gyongyver");
7   Console.WriteLine(shop.Find("Filemon"));
8 }
```

5.31. kód. Kismacska-kereskedés létrehozása

A kutyakozmetika implementációja során kicsit másként járunk el (lásd. 5.32. kódrészlet). A *DogCosmetics* egy olyan nem generikus osztály, melynek őse egy generikus osztály, meghatározott T paraméterrel (mely természetesen a *Dog*). Felolvasva ezen kódsorokat: a kutyakozmetika egy sornyi kutya kezelést megvalósító ősosztályból származik (és nem egy sornyi kisállat kezelését megvalósító osztályból, mely egy nem generikus *List<Pet>* mezővel rendelkező osztály lehetne).

```
1 public class DogCosmetics : PetHolder<Dog>
2 {
3   public void Haircut(String dogName, int length)
4   {
5     this.Find(dogName).CutHair(length);
6   }
7 }
```

5.32. kód. Kutyakozmetika

A hajvágás során (lásd. 5.32. kódrészlet *Haircut()* metódus), kikeressük a várakozó kutyák közül a megadott nevűt, majd elvégezzük a szőrzet adott méret szerinti eltávolítását. *CutHair()* metódusa csak a kutyáknak van, a *Haircut()* metóduson belül azonban semmilyen átalakítást nem kellett elvégeznünk, hiszen a szalonban kizárólag kutyák várakozhatnak.

```
1 public static void TestCosmetics()
2 {
3   DogCosmetics cosmetics = new DogCosmetics();
4   cosmetics.Add(new Dog("Bodri"));
5   cosmetics.Add(new Dog("Puli"));
6   cosmetics.Haircut("Puli", 5);
7   Console.WriteLine(cosmetics.Find("Puli"));
8 }
```

5.33. kód. Kutyakozmetika tesztelése

Azzal, hogy a generikus típust elfedtük, a kutyakozmetika osztály példányosítása és felhasználása során teljesen transzparens lett az, hogy a háttérben generikus típusok vannak (lásd. 5.33. kódrészlet). Ez sok szempontból előnyös, és egyben bemutat egy példát arra is, hogyan lehet relációba hozni a generikus és a nem generikus típusokat.

6. fejezet

Játékok

Ismeretszerzés

mock/fake objektum, entitások azonosítása, egy felelősség elve, varargs, indexer, feladat kontextusa, követelmények pontosítása, tiszta futási ágak

A felnőttkor egyik legnagyobb lehetséges tévedése, hogy a játék csupán a kisgyerekek számára fontos. A játék mindannyiunk mindennapjának részét képzi, sokszor apró lépések megtételekor megszerzett apró örömök azok, melyek biztatnak minket a következő lépés megtételére. Ahogy a játék egyidős lehet az emberi civilizációval, úgy a számítástechnikában is kezdetektől jelen van, és mi is megpróbáljuk az ez által szerzett örömöket ismeretszerzésünk növelésében manifesztálni. Meglepő módon, ha a kiskamasz szobája kitakarításáért kapna 30 XP-t, legközelebb nem engedné a testvérének hogy felvegye a ledobott morzsát a szőnyegről...

Van abban valami érdekes, ha az ember a saját maga által készített játékkal játszik. Egészen máshogy figyeli a szabályokat, a lehetséges kombinációkat, a mesterséges játékos lépéseit. Azon gondolkodik, hogy tudná még élvezetesebbé, még jobbá tenni a játékélményt. Mindezt tökéletesen lehet függetleníteni a grafikától, sőt akár a megjelenítéstől is. Ebben a fejezetben egyszerű, néhol ismerős játékok fejlesztésébe és szimulálásába vágjuk a fejszénket, és talán átérzünk valamit abból, amit egy *Amiga* vagy *Commodore* játék fejlesztői a mai napig éreznek, ha előveszik egyik-másik 25 éves játékukat (pedig közülük nem kevesen a mai napig a játékfejlesztésben dolgoznak).

6.1. Kockajáték

Előfeltétel

egység tesztelés, kivételkezelés, öröklés, Random osztály

Készítsünk egy egyszerű kockajátékot, ahol a játékos általa választott tét letétele után két hat oldalú szabályos kockával dob. Ha a dobott számok megegyeznek, a tétje duplájáát nyeri. Ennél is jobban jár, ha mindkét szám a kockán a hatos lesz, ilyenkor háromszoros tétet visz haza. Ha a dobott értékek nem azonosak, de egymást követő számok, akkor csak a tétet bukja, minden más esetben azonban ennek kétszeresét veszíti. Kivételkezeléssel valósítsuk meg azt az esetet, mikor a játékosnak elfogy a pénze (tétet még tudott tenni, de nem volt szerencséje, és veszített). Készítsünk szimulációt, melyben 1000 egységnyi vagyonnal beül egy játékos játszani, és 30-100 közötti tétekkel játszik folyamatosan. Számoljuk meg, hány partit képes végigjátszani a pénzével.

Azt már az elején fontos közölni: ebben a játékban a bank van nyertes helyzetben. Ez egyrészt a szomorú valóság, másrészt remek abból a szempontból, hogy a szimulációnk csak elméleti síkokon lesz "végtelen ciklus". A szimuláció mellett a játékhoz egység teszteket is fogunk készíteni, melyekben megkíséreljük a véletlent kiküszöbölni az egyenletből!

6.1.1. Entitások azonosítása

A feladatkiírásból elég hamar kiolvasható az a három alap entitás, mely köré az alkalmazást építeni fogjuk: *játékos*, *dobókocka* és maga a *játék*. A *játékos* "játszik és pénzt nyer/veszít", a *dobókocka* "dob", a *játék* pedig "összefog és vezérel". Ezek a főnév-cselekvés párok kijelölik számunkra a felelősségeket, és egyben az osztályokat is. Ha TDD módszerét alkalmaznánk, akkor nem ezen entitások végiggondolásával indulna a feladat elkészítése, hanem megpróbálnánk felírni a legelső és legegyszerűbb tesztesetet, melynek a követelményét a leendő (akkor még nem létező) alkalmazásunknak teljesítenie kell. Pl. ha valaki beül játszani 0 egység pénzzel, akkor nem tud tétet tenni, vagy ha a játékos 100-as tétet tesz, és két hatost dob, akkor 200-al több pénze lesz, mint mielőtt beült játszani. Ezeket az eleinte egyszerű, majd később összetettebb eseteket megvalósítva a végén "kialakul" valahogyan az alkalmazás egésze is. Nem biztos, hogy az így kialakított kép azonos lesz azzal, amit "hagyományos" módszer szerint implementálunk, azonban várhatóan a kettő egymásra "refaktorálható", ha ez az összkép végén valamelyik oldalon szükségesnek érződik.

6.1.2. Játékos

Visszatérve az azonosított entitásokra, a legegyszerűbb talán a *játékos*, akit a vagyona "azonosít", és ennek gyarapodása illetve szivárgása jelöli ki a leendő osztály műveleteit. Mindezt a 6.1. kódrészlet mutatja be.

```
1 public class Player
2 {
3   private int money;
```

```
4
5   public int Money
6   {
7     get { return this.money; }
8   }
9
10  public Player(int money)
11  {
12    this.money = money;
13  }
14
15  public void Pay(int bet)
16  {
17    this.Lose(bet);
18  }
19
20  public void Win(int amount)
21  {
22    this.money += amount;
23  }
24
25  public void Lose(int amount)
26  {
27    this.money -= amount;
28    if (this.money < 0)
29    {
30      throw new NotEnoughMoneyException();
31    }
32  }
33
34 }
```

6.1. kód. Játékos

Tétet tenni és veszíteni valójában ugyanazt az állapotváltozást okozza: a téttel, vagy a vesztett mennyiséggel csökken a játékos vagyona. A kettő állapotváltozás a kód olvashatósága végett két különböző néven elért azonos funkciót takar, a *Pay()* nyilvános művelete a háttérben a *Lose()* metódust hívja meg. Ha a delikvens pénze elfogy, az alkalmazás egy *NotEnoughMoneyException* kivételt hoz létre és dob el, ahogyan ezt a feladat kiírása definiálta (a kivétel osztályát lásd. a 6.2. kódrészletben).

```
1 public class NotEnoughMoneyException : ApplicationException
2 {
3 }
```

6.2. kód. Kivétel

6.1.3. Dobókocka

Hasonlóan egyszerű felelőssége van a *dobókockát* reprezentáló osztálynak is: egy véletlenszám generátor segítségével szimulálnia szükséges kérésre egy hat oldalú kockával való dobás eredményét. Erre talán pazarlás is egy külön osztály létrehozása, azonban itt most célunk van mindezzel: mivel *mocking framework* megismerését nem tűzzük ki célul magunk elé, viszont tesztelni szeretnénk a játék viselkedését (melynek alapja lesz a véletlen dobás eredménye), szükségünk van (sajnos) a termék kódban olyan módosításra, mely a tesztelhetőséget biztosítja. A "cél szentesíti az eszközt", azonban ha ilyet teszünk mindig legyen a szemünk előtt: a termék kód az, amelyet szállítunk, amely a megrendelőt érdekli, melyért felelősséget kell vállalnunk. Minden olyan része, mely nem ezt a célt szolgálja, gyakorlatilag "halott kód"[1], és használata kockázatos és kerülendő. A véletlen

[1] Halott kód (dead code) a forráskód azon része, melyre nincsen sehol sem hivatkozás, sehol sem használjuk. Ha valamit csak és kizárólag az egység tesztekből használunk, avagy ezek miatt létezik

generátort, vagyis a *Random* osztály példányát kell kijátszanunk, ehhez a következő technikát fogjuk választani: becsomagoljuk a *Random* osztályt egy saját típusba (ezt hívjuk *Wrapper* osztálynak), és azokat a metódusokat, melyeket az eredeti *Random* osztályból használni szeretnénk, kiemeljük majd delegáljuk ezen új csomagoló osztályban. A "trükk" ott lesz, hogy e kiemelt metódusokat *virtuálissá* tesszük, és lehetőséget biztosítunk majd arra, hogy a tesztből egy "fake" leszármazottban ezen *virtuális* metódusok viselkedését megváltoztassuk[2].

A 6.3. kódrészlet a *Random* osztály csomagolóosztályát mutatja be, melyet a *dobókocka* osztályában használunk fel (lásd. 6.4. kódrészlet). A *Wrapper* osztály 10. sorában megtalálható **virtual** kulcsszó a lényeges, illetve természetesen a *Next()* metódus hívásának delegálása (egy kicsit a *Next()* felelősségébe is belenyúltunk).

```
1  public class RandomWrapper
2  {
3    private readonly Random random;
4
5    public RandomWrapper(Random random)
6    {
7      this.random = random;
8    }
9
10   public virtual int Next(int maxValue)
11   {
12     return this.random.Next(maxValue) + 1;
13   }
14
15 }
```

6.3. kód. Véletlen becsomagolása

A *dobókockának* felelőssége az, hogy *Roll()* metódusa hívásakor visszaadjon egy 1 és 6 közötti egész számot (lásd. 6.4. kódrészlet). Ehhez csupán a véletlen szám generátor példányára van szüksége (melyet a *RandomWrapper* példányon keresztül kap meg).

```
1  public class Dice
2  {
3    private const int NUMBER_OF_SIDES = 6;
4
5    private readonly RandomWrapper randomWrapper;
6
7    public Dice(RandomWrapper randomWrapper)
8    {
9      this.randomWrapper = randomWrapper;
10   }
11
12   public int Roll()
13   {
14     return this.randomWrapper.Next(NUMBER_OF_SIDES);
15   }
16
17 }
```

6.4. kód. Kocka

a megvalósítás ebben a formájában, akkor a termék szempontjából ezek is *dead code*-ok!

[2] A C# nyelvben megkülönböztetjük a *nem-virtuális* és a *virtuális* metódusokat. A nem általunk írt osztályok *nem-virtuális* metódusait nem tudjuk felülírni egy leszármazottban. Pl. Java-ban mindez egy kicsit egyszerűbb volna, hiszen ott minden metódus *virtuális*. Nem lenne szükség *Wrapper* osztályra, amennyiben a módosítandó osztály leszármazása engedélyezett.

6.1.4. Játék

A *játék* elindításához egy *játékos* és egy *dobókocka* szükséges, melyeket "belügy" lévén a konstruktorban hozunk létre (lásd 6.5. kódrészlet). A *Play()* metódusban történik a követelményekben megfogalmazott igények kielégítése. A feladat algoritmikusan nagyon egyszerű: a játékos tétet tesz (ha nincs elég pénze ehhez, akkor itt már dobódik a megfelelő kivétel), majd két kockával dob (pontosabban "ugyanazzal" kétszer egymás után), végül az algoritmus sorban ellenőrzi, hogy a dobott számok alapján nyeremény áll a házhoz, vagy viszik a festményeket.

```
1  public class Game
2  {
3    private const int SPECIAL_ROLL_VALUE = 6;
4
5    private readonly Player player;
6    private readonly Dice dice;
7
8    public Player Player
9    {
10     get { return this.player; }
11   }
12
13   public Game(RandomWrapper random, int money)
14   {
15     this.player = new Player(money);
16     this.dice = new Dice(random);
17   }
18
19   public void Play(int bet)
20   {
21     this.player.Pay(bet);
22     int firstRoll = this.dice.Roll();
23     int secondRoll = this.dice.Roll();
24     if (firstRoll == secondRoll)
25     {
26       if (firstRoll == SPECIAL_ROLL_VALUE)
27       {
28         this.player.Win(bet * 3);
29       }
30       else
31       {
32         this.player.Win(bet * 2);
33       }
34     }
35     else if (Math.Abs(firstRoll - secondRoll) > 1)
36     {
37       this.player.Lose(bet);
38     }
39   }
40
41 }
```

6.5. kód. Játék

6.1.5. Szimuláció

A játék szimulációja során "tesztelgethetjük" hogy 1000 egység pénzzel mennyi partit vagyunk képesek "túlélni" (lásd. 6.6. kódrészlet). Lehet játszani azzal, hogy esetleg stratégiákat alakítunk ki, és ezt építjük be a szimulációba. Persze ebben az egyszerű játékban túl sok esélyünk nincs ezzel kísérletezni, de a szituáció egy összetettebb játék esetén is hasonló volna.

```
1 Random random = new Random();
2 Game game = new Game(new RandomWrapper(random), 1000);
```

```
3  int numberOfParties = 0;
4  try
5  {
6    while (true)
7    {
8      numberOfParties++;
9      game.Play(random.Next(70) + 30);
10     Console.WriteLine("Money: " + game.Player.Money);
11   }
12 }
13 catch (NotEnoughMoneyException e)
14 {
15   Console.WriteLine("Player loses all of his money within " + numberOfParties + "
         parties.");
16 }
```

6.6. kód. Szimuláció

6.1.6. Egység tesztek készítése

Az egész játék azért szerepel a jegyzetben hogy egység tesztjeit el tudjuk készíteni. Erre könnyű lehetett rájönni, hiszen a feladat egyszerűbb mint az 1. fejezetben elkészített *programozási tételek*. A 6.7. kódrészletben olvasható teszt metódus leírja a legáltalánosabb (és leggyakoribb) tesztesetet, miszerint a dobott két szám ha túl messze van egymástól, akkor a játékos a tétje kétszeresét veszíti el[3]. Az adott egység teszt bár sokszor fog helyesen lefutni, néha azért elbukik, ugyanis a véletlent nem tudjuk befolyásolni a teszt futása során. Itt jönne megoldásként valamely "mocking" framework használta: minden olyan osztály példányát, mely nem a tesztelendő osztályhoz tartozik, *mockolni/fake*-elni szükséges annak érdekében, hogy a *unit* teszt független legyen minden külső tényezőtől[4]!

```
1 [TestMethod]
2 public void Player_loses_the_bet_twice_when_the_rolled_numbers_are_far_away()
3 {
4   RandomWrapper random = new RandomWrapper(new Random());
5   Game game = new Game(random, 1000);
6   game.Play(250);
7   Assert.AreEqual(500, game.Player.Money);
8 }
```

6.7. kód. Leggyakoribb esemény tesztesete

Ezen a ponton nyer szerepet a *RandomWrapper* osztályunk, melyet az egység teszt projektben (!) leszármaztatunk, és létrehozzuk ennek "alternatív valóságát", kihasználva a

[3] A teszt metódusok elnevezése egy érdekes pontja az egység tesztek készítésének: a metódus neve lesz az, amit egy automatikusan készülő riportban, vagy a *Visual Studio* egyik ablakának felsorolásában látni fogunk először, ha az adott teszt elbukik. Nagyon fontos tehát hogy jól nevezzük el! A "Test1", "ShouldWorkWell" és társai teljesen használhatatlanok, és gyakori hiba az is, hogy a tesztelendő metódus neve alapján nevezik el az alprogramot (pl. "Play1", "Play2", stb. mivel ugyanarra a metódusra gyakran több teszteset készül). A teszt metódust soha senki nem fogja programból meghívni, és a *unit* tesztek sem hívják egymást, ezért bátran adjunk hosszú neveket. Egy módszertan szerint teljes mondatokat használjunk a teszt metódusok neveiként (az írásjeleket természetesen mellőzve). Utóbbi módszert alkalmazza ez a jegyzet is.

[4] Ez a szabály alapérvényűnek tekinthető, de azért előfordulhatnak kivételek. Ha egy osztály létrehoz egy másik osztályt, és csak és kizárólag ő használja ezt belső működése során, akkor a "fekete doboz" elve szerint erről a tesztet készítő nem is tudhat, ezáltal "mockolni" sem tudja. A példában ilyen osztály lehet pl. a *Player*, vagy egyébként a *Dice* is. Ettől függetlenül ezen osztályokat külön is érdemes lehet letesztelni. Ha egy osztályt nem *mockolunk* ki, és a valós példányt engedjük futni a tesztek során, akkor a benne lévő programhibák nem pusztán ezen osztály egység tesztjeit fogják elbuktatni, hanem minden olyan osztályét is, melyek *mock* nélkül alkalmazták ezt (és ezzel hamis képet adunk az alkalmazásban lévő hibák előfordulásáról, illetőleg elfedhetjük a hiba eredeti okát).

virtuális metódus hívás által nyújtott nyelvi lehetőségeket. A 6.8. kódrészlet egy lehetséges *fake* implementációt mutat be. A "hamis véletlenszámot" előállító leszármazott osztálynak előre megadjuk (*varargs* konstruktor paraméter segítségével) azokat a számokat, melyeket "véletlenül" pont ebben a sorrendben fog visszaadni, ha meghívjuk a *Next()* metódusát.

```
1  public class FakeRandomWrapper : RandomWrapper
2  {
3
4    private readonly int[] rolls;
5    private int index;
6
7    public FakeRandomWrapper(params int[] rolls)
8      : base(null)
9    {
10     this.rolls = rolls;
11     this.index = 0;
12   }
13
14   public override int Next(int maxValue)
15   {
16     return this.rolls[this.index++];
17   }
18
19 }
```

6.8. kód. Hamis véletlenszám generátor

A korábban megírt teszt metódust átírjuk úgy, hogy a *RandomWrapper* példányt lecseréljük *FakeRandomWrapper* példányra. Az egység tesztben így ténylegesen meg tudjuk adni azt a két számot, melyek egymástól specifikáció szerint "messze" vannak, és biztosítani tudjuk azt, hogy a játékos az adott teszt futása során mindig elveszítse a tétje kétszeresét, amennyiben az alkalmazásunk helyesen működik (lásd. 6.9. kódrészlet)!

```
1 [TestMethod]
2 public void Player_loses_the_bet_twice_when_the_rolled_numbers_are_far_away()
3 {
4   FakeRandomWrapper random = new FakeRandomWrapper(1, 3);
5   Game game = new Game(random, 1000);
6   game.Play(250);
7   Assert.AreEqual(500, game.Player.Money);
8 }
```

6.9. kód. Leggyakoribb teszteset javított változata

További teszteseteket készíthetünk ugyanehhez a *Play()* metódushoz:

- A két dobott szám közel van egymáshoz, de nem azonosak
- A két dobott szám azonos, de nem speciális
- A két dobott szám azonos és speciális

Az elkészült tesztmetódusokat mutatja be a 6.10. kódrészlet. Az összességében négy metódus sorban kijelöli az eredeti algoritmusban található elágazások által kialakított futási ágakat. Ez azért fontos, mert ha bármelyik ágban egy módosítás során olyan esemény történik, mely során az adott ág futása nem a tesztben leírt eredményt hozza, akkor lesz olyan teszteset, melynek a bukása ezt gyakorlatilag azonnal jelezni fogja.

```
1 private const int SPECIAL_ROLL_VALUE = 6;
2
3 [TestMethod]
4 public void Player_loses_the_bet_when_the_rolled_numbers_are_not_far_away_and_not_equal()
```

```
5  {
6    FakeRandomWrapper random = new FakeRandomWrapper(2, 3);
7    Game game = new Game(random, 1000);
8    game.Play(300);
9    Assert.AreEqual(700, game.Player.Money);
10 }
11
12 [TestMethod]
13 public void Player_wins_money_when_the_rolled_numbers_are_equals_but_not_special()
14 {
15   FakeRandomWrapper random = new FakeRandomWrapper(4, 4);
16   Game game = new Game(random, 1000);
17   game.Play(300);
18   Assert.AreEqual(1300, game.Player.Money);
19 }
20
21 [TestMethod]
22 public void Player_wins_double_money_when_the_rolled_numbers_are_equals_and_special()
23 {
24   FakeRandomWrapper random = new FakeRandomWrapper(SPECIAL_ROLL_VALUE,
         SPECIAL_ROLL_VALUE);
25   Game game = new Game(random, 1000);
26   game.Play(300);
27   Assert.AreEqual(1600, game.Player.Money);
28 }
```

6.10. kód. További teszt esetek

Bár a *Player* osztályra tartozik a kivétel dobása, ha elfogadjuk azt, hogy a *játékos* osztály példányosítása a *játék* osztály belügye, akkor elfér egy teszt metódus a *Game* osztály tesztjei között is, ami ezen kivétel kiváltását várja el (lásd. 6.11. kódrészlet).

```
1  [TestMethod]
2  [ExpectedException(typeof(NotEnoughMoneyException))]
3  public void Throw_NotEnoughMoneyException_when_player_loses_all_of_his_money()
4  {
5    FakeRandomWrapper random = new FakeRandomWrapper(1, 6, 2, 5);
6    Game game = new Game(random, 1000);
7    game.Play(600);
8    Assert.AreEqual(400, game.Player.Money);
9    game.Play(600);
10   Assert.Fail();
11 }
```

6.11. kód. Egység tesztek

Az egység tesztek írása van, hogy nagyobb falat, mint magának az alkalmazásnak az elkészítése. Értelemszerűen mérlegelni kell, hogy mindez kinek és mikor éri meg a ráfordítást. Azzal azonban könnyű egyetérteni, hogy hiba nélküli program nem létezik, és a hibajavítás költsége akkor a legalacsonyabb, ha azt a kódhoz legközelebb találják meg. Ha nincs egy adott kódrészletre egység teszt, de a hibát felfedezi egy tesztelő munkatárs, akkor ennek költsége már egy másik ember munkájában mérhető. Ha valamilyen oknál fogva ő sem veszi észre, de az egyik ügyfélnél előáll egy hiba, melynek a javítása során feltűnik, hogy az adott kódrészlet lefedetlen volt, akkor itt már tucatnyi ember munkaköltségével számolhatunk egy nagyobb vállalati felépítés esetén.

6.2. Kártyajáték

Előfeltétel

programozási tételek, osztályrelációk, egységbezárás, virtuális ToString() metódus

Készítsünk egy "fejszámolós" kártyajátékot, melyet a 32 lapos magyar kártyával N játékos (N<=4) tud játszani. Minden játék a kártyapakli megkeverésével indul, majd az osztó a játékosoknak sorban egymás után K darab lapot kioszt a megkevert pakli tetejéről. Minden lapnak a színe (1-4) és rangja (7-8-9-10-15-20-30-50) szerint van egy számolt értéke (a két *ordinal* szorzata). A játékosok összesítik a kezükben lévő lapok értékeit. Az lesz a győztes, akinek a kezében a lapok összege a legnagyobb! Szimuláljuk le a játék folyamatát, és mutassuk meg a számolt részeredményeket.

Bár feltehetően túl magas élvezeti faktorral nem rendelkezik ez a kártyajáték, mégis remekül alkalmas arra, hogy az objektum-orientált tervezésnek, gondolkodásnak mintájául szolgáljon. Vegyük sorba, melyek azok az adatok/állapotok, melyek a feladat szövegéből kiolvashatóak, illetve milyen "akciók", műveletek fogalmazhatóak meg ezekkel kapcsolatban (lásd. 6.1. táblázat)!

Adat/állapot	Művelet/felelősség
Kártya színének neve/értéke	Kártya értékének meghatározása
Kártya rangjának neve/értéke	Kártyapakli megkeverése
Pakli kártyalapjai	Kártyalapok kiosztása a játékosoknak
Pakli	Lapok visszaadása a pakliba
Játékos neve	Kézben lévő lapok összértékének kiszámítása
Játékos kézben lévő K lapja	Győztes játékos kiválasztása
Játék N résztvevője	Pakli legfelső lapjának visszaadása
Kártya színe	Játékos csatlakozása
Kártya rangja	Új játék indítása
...	Játékos kártyaszerzése
	Lapok eldobása a kézből
	...

6.1. táblázat. Kártyajáték tervezése

6.2.1. Feladat kontextusa

A felsorolást lehet még bővíteni, kiegészíteni. Az objektum-orientált tervezésben a kontextus kiválasztása nagyon lényeges pontja a munka elkezdésének. Bármilyen mélységig elmehetünk, amikor egy adott entitás modellezése során az adatokat elkezdjük összegyűjteni. Vegyünk példának egy rally autóversenyt, ahol az autó mint versenyző entitás bizonyára meg fog jelenni. Az autónak várhatóan modellezni fogjuk a kerekei fizikai adatait, légellenállási együtthatóit, különféle töréstesztekben mért arányszámait (stb.), miközben tároljuk a színét, matricáinak a fajtáját és helyét (stb.). Mindezekről mi döntünk azon a ponton, amikor a követelményekből kihámozzuk a valós világban létező versenyautó

számítógépben modellezett párját. Nem valószínű, hogy az adott esetben tárolni fogjuk a csomagtér-világítás izzójának típusát, vagy a motorhoz szükséges olaj paramétereit. Ellenben a motornak néhány fizikai jellemezője megjelenhet a modellben. Ha egy másik feladatban, pl. egy autószerviz működését szeretnénk számítógépen modellezni, az autó entitás szintén fel fog merülni, viszont egészen más mélységben fogjuk leírni. Lesznek közös adattagok (pl. motor teljesítménye), de valószínűleg a légellenállási együtthatókra nem lesz szükségünk, miközben akár az utolsó csavar típusát is leírhatjuk, ha a szerviz foglalkozik annak cseréjével.

Nem tudunk valamilyen gyűjtő helyről egy általános "autó" entitást előkeresni, és használni azt az alkalmazásunkban. Minden feladat más és más, és ezért előfordul, hogy az egyik programban az autó entitás a "legkisebb" egység (értve ez alatt hogy az autók leginkább aggregálva vannak más nagyobb entitásokban), míg a másikban a "legnagyobb" (az autó aggregál számos további adatot). Az sem egyértelmű, hogy "ugyanazon" entitásokat tartalmazó osztályrelációt azonosan értelmezünk. Az autóversenyben az autóhoz tartozó gumiabroncsokat bátran nevezhetjük kompozíciónak (az autó entitás tartalmaz egy halmaznyi kerék entitást), miközben ugyanezen entitások a szervizben "csupán" aggregációnak számítanak, hiszen bármikor előfordulhat hogy a szerviz kerekeket rendel be annak érdekében, hogy a kerékcserére érkező ügyfeleket kiszolgálja[5].

Az objektum-orientált programozás egyik előnye pontosan az a szabadság, melyet a kontextus választása során alkalmazhatunk. A legtöbb számítógépes modellezés a valóságra épül, az ott valahogyan működő dolgokat szeretnénk leírni egy formalizált nyelven. Ha az adott módszertan ránk erőltet egy zárt halmaznyi "típust", akkor a modellezés során durván egyszerűsíteni fogunk. Az objektum-orientált programozásban mi magunk határozzuk meg a típusokat, amik kapcsán elvben nem korlátoz minket senki. "Tökéletes" típus természetesen nem létezik, a gyakorlatban sokszor ellentmondásra is futhatunk (ellentétes igényeknek szeretnénk ugyanazt az entitást megfeleltetni), ellenben érdemes törekednünk arra, hogy az aktuális követelményeknek és a valós világban létező entitásnak a lehető legjobban és egyszerre tudjunk megfelelni.

6.2.2. Entitások csoportosítása

Visszatérve a kártyajátékra: miután összegyűjtöttük az általunk fontosnak vélt állapotokat és felelősségeket, ideje ezeket csoportosítani: melyek azok, melyek összetartoznak, és ezen tartozásnak tudunk-e nevet adni? Ha tudunk, már nyert helyzetünk van: ebből lesznek az entitásaink. Előfordulhat, hogy ezen lépés során lesznek "levegőben lógó" állapotok. Ezekről döntenünk kell: ha nem szükségesek feltétlen a feladathoz, kihúzhatjuk őket, ellenkező esetben helyet szoríthatunk nekik egy másik entitásban. Utóbbi esetben mindig ellenőrizzük, hogy az a név, melyet az így egy csoportba zárt adatoknak és műveleteknek adtunk, megfelel-e a valóságnak. Furcsa dolog ez, de sokkal fontosabb egy karbantartható kód esetén a helyesen megválasztott elnevezések megléte, mint a "hibamentes" algoritmus. Ugyanis utóbbit az előbbi megléte esetén kockázatmentesen tudjuk javítani, míg ellenkező esetben elképzelésünk sincs arra nézve, mekkora kockázat lesz ha beesik egy korábban nem várt hibajelenség.

A 6.2. táblázat az adatok egy lehetséges csoportosítását mutatja. Ennek előállítása nem mindig kézenfekvő. Egy lehetséges módszer a következő: kiválasztjuk az összetartozó

[5] Az aggregáció egy kicsit gyengébb kapcsolat, mint a kompozíció. A példában részt vevő kerék entitás szétválaszthatatlan részét képezi egy versenyautónak. Viszonylag kevés rally autót látni versenyen kerekek nélkül. Ellenben egy szervizben a kerekek önállóan is léteznek, nem szükséges hozzájuk egy autó entitás.

Adat/állapot	Művelet/felelősség
Kártya rang	
Kártya rangjának neve/értéke	
Kártya színe	
Kártya színének neve/értéke	
Kártya	
Kártya színe Kártya rangja	Kártya értékének meghatározása
Pakli	
Pakli kártyalapjai	Kártyapakli megkeverése Pakli legfelső lapjának visszaadása
Játékos	
Játékos neve Játékos kézben lévő K lapja	Kézben lévő lapok összértékének kiszámítása Játékos kártyaszerzése Lapok eldobása a kézből
Játék	
N játékos Pakli	Kártyalapok kiosztása a játékosoknak Lapok visszaadása a pakliba Játékos csatlakozása Új játék indítása Győztes játékos kiválasztása

6.2. táblázat. Csoportosítás

állapotokat, hiszen azok az állapotok, melyek egymástól azonos "szinten" függnek (vagyis a kiválasztott egy csoportba tartozó állapotok között nincsen két olyan, mely egymástól jobban függ mint a többiektől), több mint valószínű, hogy egy entitásba fog tartozni. Ezt követően elkezdjük az állapotokat és a műveleteket összekötözgetni a szerint, hogy az adott művelet elvégzéséhez melyik adattagra van szükségünk. Pl.:

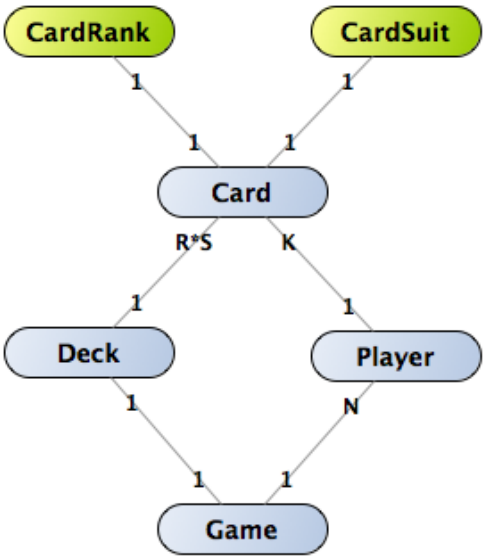
- A kártya értékének meghatározásához szükségünk van a kártya színére és rangjára.
- A kártyapakli megkeverése pakli nélkül problémás.
- A pakli legfelső lapja is a pakli kártyalapjai közül kerül ki.
- A kézben lévő lapok értékének összesítéséhez szükségünk van ezen játékosnál lévő lapokra.
- Kártyalapokat kiosztani játékosoknak tudunk, mint ahogy a győztes is e szereplők közül fog kikerülni.
- stb.

Lesznek műveletek, melyek egyértelműen egy csoportba tartozó adatokon hajtódnak végre (pl. a kártya értékének meghatározása). Ezzel a kapcsolattal már létrejött egy önálló entitás: hiszen fel tudunk sorolni összetartozó adato(ka)t, és rajta/rajtuk végzett művelet(ek)et. Lesznek olyan műveletek, melyekhez nem tartozik állapot (pl. Új játék indítása). Itt elképzelhető, hogy maga a művelet mögött nincs direkt állapotvezérlés, de az is lehetséges, hogy ezen a szinten ez még nem látszik. A másik oldal is előfordulhat: árván maradnak állapotok, nem lesznek hozzájuk csatolt műveletek (pl. kártya rangjának

neve/értéke). Itt megtehetjük, hogy dobjuk a struktúrát, és integráljuk egy már meglévőbe (a példában a kártyába helyezhetnénk a kártya rangját, mint egy egész szám, mely azt (elsősorban az értékét) szimbolizálja), vagy elgondolkodunk azon, hogy van-e olyan "csonka" típus, ami egy ilyen csökkentett felelősségnek meg tud felelni (pl. felsorolás típus vagy struktúra).

6.2.3. Osztályrelációk összeírása

Ha elkészült az entitások azonosítása, akkor a soron következő lépés lehet (nem szükségszerűen) az osztályrelációk és számosságuk előzetes felírása (lásd. 6.1. ábra). Ehhez a 3.1. fejezetben bemutatott ábrázolásmódot, vagy bármely saját jelölésrendszert használhatunk, de természetesen ha erre igény van, formalizálhatjuk ezt szabványos módon is (pl. UML diagram segítségével). Egy kártyalap egyetlen rangot és színt zár egységbe, a pakli viszont 32 kártyalapot aggregál magába (pontosabban R darab rang és S darab szín szorzatának megfelelő mennyiségű lapot). Egy játékos K kártyalapot birtokol, míg a játékot egy paklival N darab játékos játssza.



6.1. ábra. Entitások aggregációs kapcsolata

A 6.1. ábrával illetve a 6.2. táblázattal a kártyajátékot gyakorlatilag túlterveztük. Mindenképpen megjegyzendő, hogy ez ritkán vezet hatékony eredményre a gyakorlatban, azonban az objektum-orientált modellezés elsajátításának jó eszköze lehet. A feladat implementációja innentől nem más, mint a nyelv szintaktikai szabályainak betartása mellett néhány egyszerű *programozási tétel*[1] megvalósítása. Mik is ezek az utóbbiak?

- A kártyapakli megkeverése egyfajta "rendezési eljárás", bár elsősorban a **csere** segédtételét fogjuk alkalmazni benne[6].
- A játékos a kezében lévő lapokat az **összegzés** tétele segítségével fogja megszámolni.
- A győztes meghatározásához a **maximum kiválasztás** tételét fogjuk implementálni.

[6] Kártyapakli keverésére más, a valósághoz sokkal közelebb álló megoldás is elképzelhető.

Ha a feladat ezen a ponton kiosztásra kerülne seregnyi képzett *kódoló* szakembernek, egy közel egységes képű megoldás születne[7] (a megoldások "tisztasága" között azért bőven lenne eltérés).

6.2.4. Kártya színe és rangja

A "csekély" felelősséggel rendelkező, a *kártya színét* és *rangját* reprezentáló típusokat *enumként* valósítsuk meg. Ezzel képesek vagyunk az egyes "konstansoknak" nevet adni, és egyben az összetartozó elemeket csoportosítani. Az 5.1. fejezetben volt szó e típus előnyeiről és korlátairól, melyek alapján most elégséges megoldást jelentenek számunkra, ha az *ordinal* értékeket sorban beállítjuk a követelményekben meghatározottak szerint (lásd. 6.12. kódrészlet).

```
1  public enum CardSuit : short
2  {
3    Acorns = 1,
4    Bells = 2,
5    Leaves = 3,
6    Hearts = 4
7  }
8
9  public enum CardRank : short
10 {
11   r7 = 7,
12   r8 = 8,
13   r9 = 9,
14   r10 = 10,
15   Under = 15,
16   Over = 20,
17   King = 30,
18   Ace = 50
19 }
```

6.12. kód. Kártya színe és száma/rangja

6.2.5. Kártya

A *kártya* osztály példányaiban egy kártya adatait fogjuk össze: a színét és a rangját. Érdemes egy percet a 6.13. kódrészletben bemutatott definíció mezőnevein megállni. A kártya rangjának típus neve *CardRank*, mivel önmagában a *Rank* nem volna egyértelmű az esetek döntő többségében. Egy kártya osztályon belül azonban a mező, melynek típusa *CardRank* már ne vegyen fel pl. *cardRank* változónevet, mivel minden, ami egy *Card* osztály belsejében van, a kártyára vonatkozik. A "card" folytonos ismételgetése nem pusztán a redundáns kód miatt zavaró, hanem azért is, mert felhívja a figyelmet arra, ha egy adott változót prefixelni kell - jelen esetben - a *card* szóval, akkor az feltehetően azért van, mert létezik pl. egy másik típusú "rank" is az adott osztály mezői között. Utóbbi pedig a legjobb jele annak, hogy az osztályban egy másik osztály felelőssége bújt el[8]!

[7] Szándékos a *kódoló* szakma megnevezése a fejlesztő mérnök helyett. Utóbbi a modellezésben és a tervezésben szerves részt vállal.

[8] Nézzünk erre egy egyszerű példát: egy személyt szeretnénk modellezni, leírjuk nevét, szemszínét, életkorát és hogy szemüveges-e. Létrehozunk ennek egy *Person* típust. Felmerül, hogy szeretnénk tárolni a hajszínét és frizurájának "típusát" (göndör, egyenes, stb.). Első lépésre talán legtöbben felvennének a már létező *Person* osztályba két új mezőt, *hairColor* és *hairType* néven. A prefix szükséges, hiszen egy személy osztályban a "szín" vagy a "típus" egyébként nem volna egyértelmű. A két új mezőről "messziről látszik", hogy egymáshoz több közül van, mint pl. bármelyiknek a

A kártya értékét a *GetValue()* metódusban számoljuk ki, mivel senki másra nem tartozik az, hogy miként számolódik ki egy kártya értéke, csak és kizárólag a kártyára. Bár maga a művelet nagyon egyszerű, nem szabad abba a hibába esni, hogy ezt a műveletet kiszervezzük később máshova. Nézünk majd erre egy rossz példát a későbbiekben. Ne készítsük el az *accessor* elemeket (tulajdonságok **get** blokkja vagy *getter* metódusok) előre, hiszen elképzelhető hogy nem lesz rájuk szükség. Egészen addig maradjon egy kártyalap belügye az ő rangja, amíg arra valamelyik másik osztály nem lesz kíváncsi (pl. egy olyan játékban, ahol a "tökök" joker szerepben vannak, a kártya színének önálló lekérdezése szükségessé válhat, jelen esetben azonban erre nem lesz igény, sőt akár később hibákat is véthetünk ha "túlkódoljuk" magunkat, és ezzel osztályunk felelősségét).

```
1  public class Card
2  {
3    private readonly CardSuit suit;
4    private readonly CardRank rank;
5
6    public Card(CardSuit suit, CardRank rank)
7    {
8      this.suit = suit;
9      this.rank = rank;
10   }
11
12   public int GetValue()
13   {
14     return (int)this.suit * (int)this.rank;
15   }
16
17   public override string ToString()
18   {
19     return ("Card: " + this.suit + ":" + this.rank + " value: " + this.GetValue());
20   }
21 }
```

6.13. kód. Kártya

Ne menjünk addig tovább a következő osztály készítésére, amíg nem győződtünk meg arról, hogy az elkészült osztályunk az elvárásoknak megfelelően működik. Ennek egyik egyszerű ám nagyszerű módja, mellyel az osztály lényegi felelősségét nem befolyásoljuk, hogy elkészítjük az adott osztály *String* reprezentációját (felülírva a *ToString()* metódust), illetve természetesen egy apró kódrészletben készítünk az osztályból egy példányt, és meghívjuk annak üzleti metódusait (lásd. 6.14. kódrészlet). Mindez megelőzheti az osztály implementációját is, a lényeg, hogy amit ebben a kódrészletben látunk, az tiszta[9] és egyértelmű legyen. Úgy tudjuk "felolvasni" a forráskódot, mintha az egy apró története lenne egy könyvnek.

```
1 private static void TestCard(Random rand)
2 {
3   Card card = new Card(CardSuit.Acorns, CardRank.r8);
4   Console.WriteLine(card);
5 }
```

6.14. kód. Kártya tesztelése

személy nevéhez, ezért itt egy másik osztály felelősségét rejtettük el a személyen belül. Kézenfekvő a megoldás: létrehozunk egy *Hair* típust, melyben a *color* és a *type* nevű mezőnevek immáron egyértelműek, és a *haj* példányát aggregáljuk a *személy* osztályban.

6.2.6. Pakli

A következő osztály, melyhez minden aggregáció már beköthető, ezáltal szintaktikailag helyesen, fordítási hibától mentesen el tudjuk készíteni, a *Deck* vagy a *Person* osztály lehet. Válasszuk az előbbit (lásd. 6.15. kódrészlet). A *Card* osztály példányai teljesen *immutable*-ök voltak (minden állapotuk végleges, **readonly**), a *Deck* esetében a legfelső lap indexe miatt ezt nem tudjuk elérni, azonban minden esetben törekedni érdemes arra, hogy a lehető legtöbb állapot rendelkezzen C# szintaktika szerint a **readonly** módosítóval [9]. Pontosan tudjuk, hogy a pakliban 32 különböző kártyalapnak kell lennie. Ez 32 kissé unalmas programsorban bárki el tudja készíteni, azonban aki algoritmust szeretne írni, az ennek előállításához két ciklust fog készíteni (imperatív gondolkodásmód szerint, lévén ennek remek funkcionális megoldása is létezik, de utóbbival a jegyzet nem foglalkozik). A kártyalapok számát is dinamikusan képezhetjük, hiszen a színek és a rangok darabszámából kijelöli a kombinációjuk számosságát is.

Egy pillanatra elgondolkodhatunk azon is, hogy a kártyalapok listáját milyen adatszerkezetben valósítsuk meg. Tudjuk előre, hogy 32 kártyalapunk lesz, se több, se kevesebb, ezért a fix elemszámú tömb kézenfekvő megoldás lehet[10], azonban mindig figyelembe kell venni a felhasználás módját is. A választott implementációban a kártyalapok referenciái fizikailag sosem fognak kikerülni a paklból (pedig a valósághoz közelebb volna az a megoldás, ahol a játékos kezében lévő lapok referenciái csak és kizárólag ott lennének elérhetőek, a pakliban nem). Azok a lapok, melyek már nincsenek a pakliban, a pakli "legfelső lapjának indexe alatt" lesznek megtalálhatóak, illetve természetesen a kiosztott lapok referenciái átkerülnek a játékosokhoz is. A lapok megkeverése egy "helyben rendezéshez" hasonló művelet, melyet a direkt címzésű tömbök remekül támogatnak. Maradjunk tehát ezeknél.

A pakli megkeverése (lásd. 6.15. kódrészlet 30. sor) során a *csere* segédtételét fogjuk alkalmazni. Kiválasztunk két véletlenszerű elemet a sorozatban, majd azokat megcseréljük egymással. Minél több cserét hajtunk végre, annál jobban megkeverednek a kártyalapok a pakliban. Mi történik akkor, ha ugyanaz az indexű elem lesz "kisorolva" cserére? Önmagával megcserélni egy elemet látszólag nincs értelme, mégsincs egy elágazással ez az eset vizsgálva. Ennek oka a következő: ha egy elágazást elhelyezünk a ciklusmagban, akkor ez borítékolhatóan minden ciklusban ki fog értékelődni. Ha nem tesszük ki, akkor elképzelhető, hogy az esetek néhány töredék százalékában 3 értékadás műveletet feleslegesen fogunk elvégezni a csere során. Az adott esetben az utóbbi kevésbé fájdalmas megoldásnak tűnik[11].

A pakli eleddig nem említett üzleti metódusa a legfelső kártyalap visszaadása (lásd. 6.15. kódrészlet 46. sor). Ha "elveszünk" egy kártyalapot a kupac tetejéről, akkor ezt követően már másik lap lesz ott. Ezt biztosítja az 52. sorban a *topCardIndex* állapot növelése. A megoldásban van egy egyszerűsítés, mely egy felkiáltójellel meg is lett jelölve. Ha elfogynak a kártyalapok, illemes volna valamilyen kivételt dobni, hogy jelezzük a ritka eseményt. E helyett most egyszerűen "előveszünk egy új paklit" (vagyis "újrakeverjük" a meglévőt).

[9] Ez a módszer a párhuzamos programozás és úgy általában a *thread-safe* alkalmazásfejlesztésben fogja leginkább meghálálni magát, azonban a kód tisztaságán is sokat tud javítani, mivel egyértelműen kijelöli az állapotváltozások határait.

[10] Természetesen az sem utolsó, hogy a könyv épít egy olyan lépésről lépésre épített tudáshalmazra, mely kapcsán a dinamikus listák és általában véve a generikus típusok még nem kerültek előtérbe.

[11] A példa tanulsága, hogy egy ciklusmagban elhelyezett utasítás súlya a ciklusmagok futásának számával arányos!

```
1  public class Deck
2  {
3    private const int NUMBER_OF_SWAPS = 100;
4
5    private readonly Card[] cards;
6    private readonly Random rand;
7    private int topCardIndex;
8
9    public Deck(Random rand)
10   {
11     this.rand = rand;
12     CardSuit[] suits = (CardSuit[])Enum.GetValues(typeof(CardSuit));
13     CardRank[] ranks = (CardRank[])Enum.GetValues(typeof(CardRank));
14     this.cards = new Card[suits.Length * ranks.Length];
15     for (int i = 0; i < suits.Length; i++)
16     {
17       for (int k = 0; k < ranks.Length; k++)
18       {
19         this.cards[(i * 8) + k] = new Card(suits[i], ranks[k]);
20       }
21     }
22     this.topCardIndex = 0;
23   }
24
25   public void Rotate()
26   {
27     this.Rotate(NUMBER_OF_SWAPS);
28   }
29
30   public void Rotate(int time)
31   {
32     for (int i = 0; i < time; i++)
33     {
34       this.SwapCards(this.rand.Next(this.cards.Length),
             this.rand.Next(this.cards.Length));
35     }
36     this.topCardIndex = 0;
37   }
38
39   private void SwapCards(int indexA, int indexB)
40   {
41     Card tmp = this.cards[indexA];
42     this.cards[indexA] = this.cards[indexB];
43     this.cards[indexB] = tmp;
44   }
45
46   public Card GetTopCard()
47   {
48     if (this.topCardIndex >= this.cards.Length)
49     {
50       this.Rotate(NUMBER_OF_SWAPS); // !
51     }
52     return this.cards[this.topCardIndex++];
53   }
54
55   public override string ToString()
56   {
57     StringBuilder builder = new StringBuilder();
58     builder.AppendLine("Cards:");
59     for (int i = 0; i < this.cards.Length; i++)
60     {
61       builder.Append("[" + (i + 1) + "] ");
62       builder.Append(this.cards[i].ToString());
63       if (this.topCardIndex == i)
64       {
65         builder.Append(" <-- top");
66       }
67       builder.AppendLine();
68     }
69     return builder.ToString();
70   }
71
72 }
```

6.15. kód. Pakli

Mielőtt a következő osztály implementációjára áttérünk, bizonyosodjunk meg arról, hogy a *Deck* úgy és abban a formában készült el, ahogyan azt elterveztük. Hozzuk lére a paklit, majd jelenítsük meg keverés előtti állapotában. Látnunk kell a 32 különböző kártyalapot. Keverést követően tudjuk ellenőrizni, hogy tényleg megvalósul-e a véletlen kártyasorrend, és kikérve pl. a pakli három legfelső lapját, tényleg azokat kapjuk-e, melyekre számítottunk (lásd. 6.16. kódrészlet).

```
1  private static void TestDeck()
2  {
3    Deck deck = new Deck(new Random());
4    Console.WriteLine(deck);
5    deck.Rotate();
6    Console.WriteLine(deck);
7    Console.WriteLine(deck.GetTopCard());
8    Console.WriteLine(deck.GetTopCard());
9    Console.WriteLine(deck.GetTopCard());
10 }
```

6.16. kód. Pakli tesztelése

6.2.7. Játékos

A soron következő osztály a *játékos* modellje. Az osztály hasonló állapotokat birtokol, mint a kártyapakli, hiszen felfoghatjuk a játékos kezében lévő lapokat egy "mini" paklinak is[12]. A játékos az *AddCard()* metódus iteratív hívásával lapokat kap az osztás során, a *DropCards()* végrehajtásakor pedig "visszateszi" azokat a pakliba. Utóbbi a választott implementációban nagyon egyszerű, mivel fizikailag a pakliban is rendelkezésre állnak azoknak a kártyáknak a referenciái, melyek a játékos kezében voltak, egyszerűen "eldobhatjuk" ezen referenciákat (ha a *cardIndex*-et, mely a kezében aktuálisan lévő lapok számát jelöli beállítjuk nullára, a következő kártyaszerzése sorban felül fogja írni a korábban kezébe kapott referenciákat).

A *GetCardsValue()* metódus (lásd. 6.17. kódrészlet 29. sora) az *összegzés tételének* tankönyvi esete. Lényeges pontja az algoritmusnak egy kártya értékének elkérése a *Card* osztály *GetValue()* metódusának hívásával.

```
1  public class Player
2  {
3    public const int NUMBER_OF_PLAYER_CARDS = 3;
4
5    private readonly String name;
6    private readonly Card[] cards;
7    private int cardIndex;
8
9    public Player(String name)
10   {
11     this.name = name;
12     this.cards = new Card[NUMBER_OF_PLAYER_CARDS];
13     this.cardIndex = 0;
14   }
15
16   public void AddCard(Card card)
17   {
```

[12] Az implementáció során az öröklés témakörét a virtuális *ToString()* metódus használatán kívül nem cél a feladat kapcsán előtérbe hozni.

```
18      if (this.cardIndex < NUMBER_OF_PLAYER_CARDS)
19      {
20        this.cards[this.cardIndex++] = card;
21      }
22    }
23
24    public void DropCards()
25    {
26      this.cardIndex = 0;
27    }
28
29    public int GetCardsValue()
30    {
31      int ret = 0;
32      for (int i = 0; i < this.cardIndex; i++)
33      {
34        if (this.cards[i] != null)
35        {
36          ret += cards[i].GetValue();
37        }
38      }
39      return ret;
40    }
41
42    public override string ToString()
43    {
44      StringBuilder builder = new StringBuilder();
45      builder.AppendLine("Player " + this.name);
46      for (int i = 0; i < this.cardIndex; i++)
47      {
48        if (this.cards[i] != null)
49        {
50          builder.Append("(" + (i + 1) + ") ");
51          builder.AppendLine(this.cards[i].ToString());
52        }
53      }
54      builder.AppendLine("SumValues: " + this.GetCardsValue());
55      return builder.ToString();
56    }
57  }
```

6.17. kód. Játékos

Vizsgáljuk meg egy pillanatra a 6.18. kódrészletet. Ugyanazt az összegzés műveletét mutatja be, viszont a kártya színét és rangját kikérve számolja ki a lap értékét. Látszólag az algoritmus helyes, azonban olyan műveletet végez ez a *Player* osztály, mely a *Card* osztály egyértelmű felelőssége. Játszunk el a szituációval, miszerint változnak a követelmények, és mostantól a kártyalap értéke a rangjának és színének összege lesz, és nem a szorzata. Átírjuk az implementációt a *Card* osztályban (ahova tartozik, és ahol keressük a módosítás elvégzése után a kapcsolódó kódrészleteket), viszont elfelejtjük átírni a *játékos* osztályban. A program futni fog, hiba nem lesz, de lesznek olyan esetek, amikor a program "furcsán" fog viselkedni. A felvázolt hibás implementáció egyáltalán nem ritka a szoftverfejlesztésben. Talán emlékszünk még arra, hogy a *Card* osztálynál a *rang* és a *szín* nyilvános tulajdonságait nem készítettük el előre. Ez a technika támogat minket abban, hogy az implementációt ne vigyük félre, ne helyezzünk át felelősséget azáltal, hogy egy példányon keresztül a példány több állapotát kikérve valahol máshol végzünk el egy műveletet, mely valójában a két állapotra tartozik.

```
1  public class Card
2  {
3    [..]
4
5    public CardSuit Suit
6    {
7      get { return this.suit; }
8    }
9
```

```
10    public CardRank Rank
11    {
12      get { return this.rank; }
13    }
14
15    [..]
16  }
17
18  public class Player
19  {
20      [..]
21
22    public int GetCardsValue()
23    {
24      int ret = 0;
25      for (int i = 0; i < this.cardIndex; i++)
26      {
27        if (this.cards[i] != null)
28        {
29          ret += (int)this.cards[i].Rank * (int)this.cards[i].Suit;
30        }
31      }
32      return ret;
33    }
34
35    [..]
36  }
```

6.18. kód. Hibás összegzés

Adjunk esélyt *Terence Hillnek* némi kártyalap birtoklásra, és ellenőrizzük le, hogy az *összegzés tétele*, melyet elkészítettünk, helyesen működik. Ehhez - ahogyan ez a *Card* és a *Deck* esetén is volt - írjuk felül az osztály ősétől örökölt virtuális *ToString()* metódust (lásd. 6.17. kódrészlet 42. sora).

```
1  private static void TestPlayer()
2  {
3    Card card = new Card(CardSuit.Acorns, CardRank.r8);
4
5    Player player = new Player("Terence Hill");
6    player.AddCard(new Card(CardSuit.Acorns, CardRank.r8));
7    player.AddCard(new Card(CardSuit.Bells, CardRank.r9));
8    player.AddCard(new Card(CardSuit.Acorns, CardRank.Ace));
9
10   Console.WriteLine(player);
11 }
```

6.19. kód. Játékos tesztelése

6.2.8. Játék

Az előzetes tervek alapján egy osztály maradt a végére, melynek aggregatív kapcsolatait immáron képesek vagyunk elkészíteni: a *játék* modellje. A *Game* osztály célja egységbe zárni és vezérelni a felmerült entitásokat. "Kívülről", vagyis a példány oldaláról nézve (lásd. 6.20. kódrészlet) a játékot létrehozzuk, magadva a játékosok számát, majd sorban ezt követően "beültetjük" őket az *AddPlayer()* metódus ismételt meghívásával. Új játékot a *Play()* segítségével indíthatunk, mely visszaadja számunkra a nyertes játékos referenciáját[13].

[13] Egyszer egy barátom azt mondta nekem, az informatikában az a szép, hogy egy pókerasztalhoz ültethetjük Bud Spencert és Anakin Skywalkert, de még akár utóbbi önmagát is legyőzheti a távoli jövőben egy messzi messzi galaxisban, egy pár soros C# kód szimulálása során.

```
1  private static void TestGame()
2  {
3    Game game = new Game(new Random(), 4);
4    game.AddPlayer("Terence Hill");
5    game.AddPlayer(new Player("Bud Spencer"));
6    game.AddPlayer("Darth Vader");
7    game.AddPlayer("Anakin Skywalker");
8
9    Console.WriteLine("------- WINNER -------");
10   Console.WriteLine(game.Play());
11   Console.WriteLine("------- GAME -------");
12   Console.WriteLine(game);
13 }
```

6.20. kód. Játék tesztelése

A játék lényegi része elsősorban a *Play()* metódus implementálása (lásd. 6.21. kódrészlet 27. sor), mely három részfeladatra bontható:

1. Kártyapakli megkeverése.
2. Lapok kiosztása a játékosoknak.
3. Győztes játékos kiválasztása.

A legelső részfeladat megvalósítása a legegyszerűbb: delegáljuk a feladatot a kompozícióban lévő kártyapaklira (lásd. 6.21. kódrészlet 29. sor). A kiosztás már egy érdekesebb feladat. A feladatkiírás alapján sorban egymás után (a beülés sorrenjében) minden játékos egyesével kap lapot a kártyapakli tetejéről. Ez leírva jól hangzik, arra kell törekedni, hogy a forráskódban megtalálható implementáció ugyanilyen egyszerűen "felolvasható" legyen. Ezt nagyban segíti egyrészt a *Deck* osztály *GetTopCard()*, illetve a *Player* osztály *AddCard()* metódusa.

A győztes kiválasztása egy *maximum kiválasztás tétele* a játékosok kezükben lévő lapjaik összesített értékére nézve (lásd. 6.21. kódrészlet 45. sor). Amire itt érdemes figyelni, az a *GetCardsValue()* metódus eredményének lokális változóban való tárolása (54. sor). Mivel ez számunkra "fekete doboz", elvben nem kell tudnunk arról, hogy a metódus mögötti implementáció a *Player* osztályban egy *összegzés tételét* takar, azonban ne csukjuk be a szemünket sem. Sokat nyerhetünk a lokális változóban való eltárolással, mert így pontosan minden játékos esetén egyszer fogjuk a összegzést elvégezni (az 57. sorban nem számoljuk ki ismét, lokális maximum megtalálásakor).

```
1  public class Game
2  {
3    private readonly Player[] players;
4    private readonly Deck deck;
5    private int numberOfPlayers;
6
7    public Game(Random rand, int maxNumberOfPlayers)
8    {
9      this.players = new Player[maxNumberOfPlayers];
10     this.deck = new Deck(rand);
11     this.numberOfPlayers = 0;
12   }
13
14   public void AddPlayer(Player player)
15   {
16     if (this.numberOfPlayers < this.players.Length)
17     {
18       this.players[this.numberOfPlayers++] = player;
19     }
20   }
```

```
21
22    public void AddPlayer(String name)
23    {
24      this.AddPlayer(new Player(name));
25    }
26
27    public Player Play()
28    {
29      this.deck.Rotate();
30      this.Division();
31      return this.GetWinner();
32    }
33
34    private void Division()
35    {
36      for (int i = 0; i < this.numberOfPlayers; i++)
37      {
38        for (int k = 0; k < Player.NUMBER_OF_PLAYER_CARDS; k++)
39        {
40          this.players[i].AddCard(this.deck.GetTopCard());
41        }
42      }
43    }
44
45    private Player GetWinner()
46    {
47      Player winner = null;
48      if (this.numberOfPlayers > 0)
49      {
50        int maxValue = this.players[0].GetCardsValue();
51        int maxPosition = 0;
52        for (int i = 1; i < this.numberOfPlayers; i++)
53        {
54          int currentValue = this.players[i].GetCardsValue();
55          if (currentValue > maxValue)
56          {
57            maxValue = currentValue;
58            maxPosition = i;
59          }
60        }
61        winner = this.players[maxPosition];
62      }
63      return winner;
64    }
65
66    public override string ToString()
67    {
68      StringBuilder builder = new StringBuilder();
69      builder.AppendLine("Game:");
70      for (int i = 0; i < this.numberOfPlayers; i++)
71      {
72        builder.Append("[" + (i + 1) + "] ");
73        builder.AppendLine(this.players[i].ToString());
74      }
75      builder.Append(this.deck.ToString());
76      return builder.ToString();
77    }
78  }
```

6.21. kód. Játék

6.2.9. Varargs

A fejezet végén vizsgáljunk meg néhány *szintaktikai cukorkát*. A játék példányosításakor meg kellett adnunk a játékosok maximális számát, majd sorban meghívni az *AddPlayer()* metódust. Utóbbi már önmagában is kicsit redundáns (bár ez a szimuláció adatrögzítése, valós körülmények között nem lenne redundáns), azonban talán nagyobb korlát az, hogy a hívások száma és a konstruktor paraméter között függés áll fenn (és ezen nem segít, hogy a hibaesetet kezeljük valamilyen módon). A játék első indítása előtt

szükséges a játékosok meghatározása, így konstruktor paraméter is lehetne az adatszerkezet. Azonban egy tömböt átadni túl nagy felelősség, ahogy erről már korábban volt szó. Egy apró szintaktikai cukorka segítségével viszont elegáns módon látszólag megoldhatjuk a problémát: *varargs*-ot használunk, ahogy ez a 6.22. kódrészlet 4. sorában látható.

```
1 private static void TestGame()
2 {
3   Console.WriteLine("# Test Game class");
4   Game game = new Game(new Random(), "Terence Hill", "Bud Spencer", "Darth Vader",
        "Anakin Skywalker");
5   Console.WriteLine("------- WINNER -------");
6   Console.WriteLine(game.Play());
7   Console.WriteLine("------- GAME -------");
8   Console.WriteLine(game);
9 }
```

6.22. kód. Játék létrehozása elegánsabb módon

A megoldás (lásd. 6.23. kódrészlet) implementálása során figyeljünk arra, hogy a létrehozott új konstruktor és a régi (ha megmarad) ne tartalmazzon redundáns részeket. Ez a példa implementációban egy harmadik, **private** láthatóságú konstruktor létrehozásával lett kiküszöbölve.

```
1 public class Game
2 {
3   [..]
4
5   public Game(Random rand, int maxNumberOfPlayers)
6     : this(rand)
7   {
8     this.players = new Player[maxNumberOfPlayers];
9   }
10
11  public Game(Random rand, params String[] playerNames)
12    : this(rand)
13  {
14    this.players = new Player[playerNames.Length];
15    foreach (String playerName in playerNames)
16    {
17      this.AddPlayer(playerName);
18    }
19  }
20
21  private Game(Random rand)
22  {
23    this.deck = new Deck(rand);
24    this.numberOfPlayers = 0;
25  }
26
27  [..]
28 }
```

6.23. kód. Konstruktor varargs paraméterrel

Sajnos azonban a megoldás bár elegánsnak tűnik, a tömb átadásakor fellépő veszélyek ellen nem véd (lásd. 6.24. kódrészlet).

```
1 String[] names = new String[4];
2 names[0] = "Terence Hill";
3 names[3] = "Anakin Skywalker";
4 Game game = new Game(rand, names);
```

6.24. kód. Játék létrehozásának veszélyei

6.2.10. Indexer

Van még egy témakör, melyet a feladat kapcsán megvizsgálhatunk. Létezik a C# nyelvben egy speciális tulajdonság, az *indexer*. Minden osztálynak lehet egy olyan felelőssége, mely kapcsán "ő" indexelhetővé válik. A szintaxis azonos lesz a tömbök indexelésével, mely egyrészt előnyös (ismerős szintaxis egy logikailag azonos helyen), másrészről - mint majd látni fogjuk - veszélyeket is jelenthet.

A játékos kezében lévő lapokat tegyük elérhetővé kifelé úgy, hogy az osztály példányát "indexeljük". Ezt mutatja be a 6.25. kódrészlet 5. sora. A *GetCard()* illetve a *SetCard()* metódus azt mutatja be, hogy miként tudnánk ezt a szintaktikai cukorkát kiváltani két metódussal.

```
1  public class Player
2  {
3    [..]
4
5    public Card this[int index]
6    {
7      get { return this.cards[index]; }
8      set { this.cards[index] = value; }
9    }
10
11   [..]
12
13   public Card GetCard(int index)
14   {
15     return this.cards[index];
16   }
17
18   public void SetCard(int index, Card value)
19   {
20     this.cards[index] = value;
21   }
22
23   [..]
24 }
```

6.25. kód. Kézben lévő kártyák "indexelése"

Az így létrehozott *indexert* felhasználhatjuk a *Game* osztályban a lapok kiosztásakor, ahogy ez a 6.26. kódrészlet 19. sorában látható. Látszólag tömör és szép megoldás, de azért gondolkodjunk el egy percet mit is látunk az adott sorban:

- A this.players[i] megcímez egy játékost a nevezett tömbben.
- A játékos [k]. eleme azonban nem egy tömb, ezért ez nem egy közvetlen ugrás egy elemre (a jelen példában egyébként erre vezet vissza).

Ugyanaz a szintaxis, e miatt a mögöttes tényleges jelentés a leírt kódból nem olvasható ki. Ezen az tud segíteni (és annak kell segítenie), ha az *indexer* tulajdonságot csak és kizárólag olyan szituációkban alkalmazzuk, ahol a jelentése az adott típusra nézve egyértelmű (pl. listák, szótárak esetén kézenfekvő és kényelmes, de majd később listákat kezelő vezérlők kapcsán is találkozni fogunk vele).

```
1  public class Game
2  {
3    [..]
4
5    public Player this[int index]
6    {
7      get { return this.players[index]; }
```

```
8       set { this.players[index] = value; }
9     }
10
11    [..]
12
13    private void Division()
14    {
15      for (int i = 0; i < this.numberOfPlayers; i++)
16      {
17        for (int k = 0; k < Player.NUMBER_OF_PLAYER_CARDS; k++)
18        {
19          this.players[i][k] = this.deck.GetTopCard();
20        }
21      }
22    }
23
24    [..]
25  }
```

6.26. kód. Játékosok "indexelése"

A 6.27. kódrészlet csak egy apró kiegészítést ad hozzá az eddigiekhez. A játékban is felvehetünk egy *indexert* a játékosok címzésére nézve. Ennek segítségével el tudjuk érni valamely játékos kezében lévő kártyalapjait[14].

```
1  private static void TestPlayer()
2  {
3    Player player = new Player("Teszt Elek");
4    Card card = new Card(CardSuit.Acorns, CardRank.r8);
5
6    // without indexer
7    player.SetCard(0, card);
8    // with indexer
9    player[0] = card;
10   }
11
12   private static void TestGame()
13   {
14   Game game = new Game(new Random(), 4);
15   [..]
16   Console.WriteLine("------- Third player's second card -------");
17   Console.WriteLine(game[2][1]);
18 }
```

6.27. kód. Indexerek tesztelése

6.2.11. Összegzés

A magyar kártyajáték implementálása során egy alaposan megtervezett objektumorientált programot készítettünk el. Minden olyan ponton, ahol egy építőkövet elkészítettünk, szimuláltuk az addig elkészült rész működését. Ez kiemelten fontos abból a szempontból, hogy az erre épülő további komponensek így nagyobb biztonsággal bízhatnak meg ezekben aggregáció során (az egység tesztek írása még további biztonságot adna e téren). Érdemes a programban a felelősségek szervezését is áttekinteni, akár az olyan egyszerű esetben, mint a megjelenítés: minden osztály rendelkezik *ToString()* metódussal. A játék megjelenítése során megjelenítjük a játékosokat, azok összesített pontjait, kezében

[14] Érdekes gondolkodásmódot fed el az *indexer* szintaxisának nem ismerése. Azok között, akik elkezdik használni a dinamikus lista és szótár adatszerkezeteket, vagy akár valamely listás GUI vezérlőt, nem mindannyian észlelik az ott alkalmazott szintaxis "furcsaságát". Ez arra utal(hat), hogy az osztálykönyvtár felhasználása során annak implementációjával nem, csak és kizárólag a használával foglalkozunk.

lévő lapjait, magát a játék során megkevert paklit, és annak legfelső lapját. Ehhez képest egy kártya megjelenítése pusztán annak rangjának és színének összeillesztése, mégis az alapját képezi mindennek. A *Game* osztály *ToString()* metódusának kódbonyolultsága nem nagyobb, mint a *Player* vagy a *Deck* osztály *ToString()* metódusáé, pedig kimenetben lényegesen több információt tartalmaz. Ha helyesen vagy megfelelően tervezzük meg osztályainkat, a bejövő új igények implementálása nem fogja szükségszerűen a meglévő kód bonyolultságát növelni. Hogy miért fontos mindez? Ha nem tudjuk a kód bonyolultságát kordában tartani, akkor egy ponton eléri azt a szintet, amit emberi (átlagos) szellemi képesség már nem tud átlátni. Ez az a pont, amikor egy alkalmazás *legacy*-vá válik. Benne új funkciók nem implementálhatóak, és a hibák javításával általában újabb hibák keletkeznek.

6.3. Ki nevet a végén?

Előfeltétel

programozási tételek, Random osztály, referencia típusú felsorolás típus, tömbök, generikus listák, overloading, osztályszintű metódusok, öröklés, többalakúság, több dimenziós tömb literálok, StringBuilder, Comparer

Készítsük el a "Ki nevet a végén?" népszerű klasszikus társasjáték szimulációját!

Rövid, tömör és egyértelmű követelményleírás. A játékot sokat ismerik, kérdésnek helye sincsen. Kis tervezés után már neki is eshetünk az implementációnak. Szükségünk lesz egy dobókockára, lesz egy pálya, játékosok sorban lépegetnek. Az majd érdekes lehet, hogy leüthetik-e egymást a figurák, illetve ha egy játékosnak több figurája van "mozgásban", melyikkel lépjünk előre, mi alapján döntsük el? Kilépni az adott játékos start mezőjére csak hatos dobással lehet? Ilyenkor újat dobhatunk, vagy sem? Itt is le lehet ütni ellenséges játékos bábuját? Megannyi kérdés felmerülhet, talán mégsem annyira egyértelmű a követelményleírás!

A valós életben nagyon gyakori, hogy megfogalmazásra kerül egy igény, mely egy adott absztrakciós szinten elegendő lehet, az implementációhoz közel azonban mindez édes kevés. Megtehetjük, hogy elkezdünk feltételezésekbe bocsátkozni, azonban a "megrendelőnél" megvannak a válaszok a kérdéseinkre, csupán számára ezek "egyértelműek", és utólag ha nem egyre gondoltunk, ő kérni fogja az implementáció megváltoztatását. Mi tehát a megoldás?

Ha a feladat leírása egy mérnök szemével nézve hagy kérdéseket maga után, akkor ezeket a kérdéseket fel kell tenni, és a megfelelő válaszok figyelembevételével lehet csak a tervezéssel tovább menni. Előfordulhat ilyenkor, hogy a megrendelő az egyébként "egyértelmű" szabályok megfogalmazásában kihívások elé érkezik. Ilyenkor általában nem érdemes ezen sokat rugózni, be kell vonni egy "szakértőt", aki segít a specifikáció megfogalmazásában. A köztes szereplő általában az üzleti hátteret és az implementáció részleteit sem ismeri alaposan, azonban az általa megfogalmazott követelményekhez tud mindkét fel ragaszkodni.

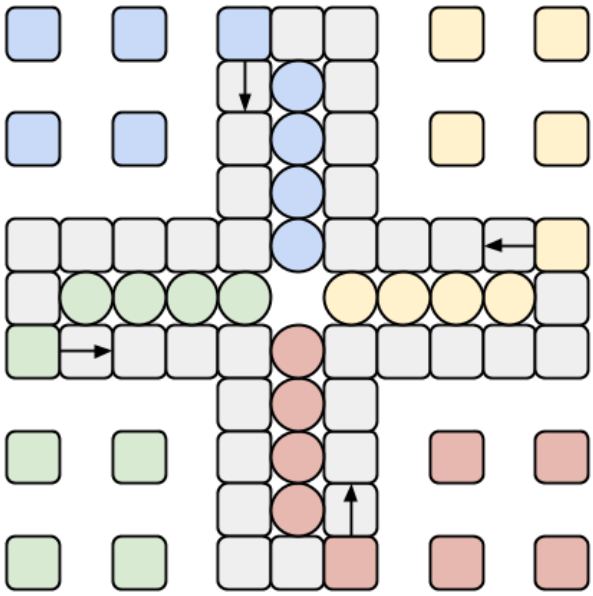
Vegyük sorra kérdéseinket, és ez alapján készítsünk részletes követelmény leírást!

- A játékot mennyien játszhatják, 2, 3 és 4 fő?
- Minden játékosnak 4 bábúja van? A pálya a klasszikus 40 mezős játéktábla? Jó volna egy ábra az egyértelműsítés kedvéért!
- Le tudják ütni egymást a bábuk? A játékosok szimulációja során előnyben legyen részesítve az, ha ütés lehetősége fennáll?
- A start mezőre csak 6-os dobással lehet kiállni? Le lehet ütni játékost a start mezőn? A saját start mezőn a bábu védett helyen van?
- A célba pontos dobással lehet bejutni, vagy elég ha körbeér a bábu a pályán?
- Ha a játékos 6-ost dob, ismét dobhat? Vagy mindez csak a start mezőre kiállás után van így, vagy ekkor sem?

- A játékosok az órajárás szerint vagy ellentétes irányban kövessék egymást? Ki kezdi a játékot?

Számos kérdés, melyek közül bár bizonyára néhányat ha megválaszolunk a megrendelő helyet, fel sem tűnne számára az, hogy ez felmerült mint eldöntendő tényállás. Érdemes figyelni arra egy valós szituációban, hogy ne tegyük fel egyszerre az összes kérdésünket. Prioritizálni szükséges, mivel egy-egy kérdésre adott válasz várhatóan fel fog vetni újabb kérdéseket (vagy még bizonytalanabbak leszünk a válaszban), illetve azokra a kérdésekre, melyekre a megrendelő nem fog tudni azonnal válaszolni, előszeretettel átugorja, felülemelkedve a kérdés tényén is. Utóbbi esetre a legjobb módszer, ha ezen kritikus kérdések "külön szálon" kerülnek egyeztetésre, így kisebb az esélye az elkeveredésnek. Nézzük hát a módosított specifikációt!

Készítsük el a "Ki nevet a végén?" népszerű klasszikus társasjáték 4 szereplős szimulációját (a tábla kinézetét lásd. 6.2. ábra)! Minden játékos 4 bábuval rendelkezik, sorban egymást követve az óramutató járásával ellentétesen követik egymást, kivéve ha valaki hatost dob, mert ilyenkor ismét dobhat. Mindaddig ő következik, amíg hatost dob. Utóbbi alkalmas arra is, hogy kilépjen a start mezőre. Egy játékosnak több bábuja is mozgásban lehet, mindig a célhoz legközelebb álló fog lépni, kivéve ha van lehetőség hatos dobásnál a start mezőre lépni (van még bázison lévő játékos, és nincs saját bábu a bázison). Egy mezőn csak egy figura szerepelhet. Ellenséges bábut bárhol le lehet ütni a pályán, de nem cél a szimulációban a szándékos ütés keresése. A célba nem szükséges a pontos dobás. Ha egy bábu már több mint 40 mezőt lépett, automatikusan a célba jut. Az lesz a győztes, akinek a leghamarabb jut a célba mind a négy figurája. Készítsünk dobogós eredménylistát a szimuláció során. A már célba ért játékost kihagyják a körökből, egészen addig, amíg a harmadik helyezett helye is egyértelmű lesz. Ekkor ér véget a szimuláció (az utolsó játékosnak nem kell köröznie egyedül a pályán).



6.2. ábra. Ki nevet a végén?

Bár nem rövid és tömör az új specifikáció, azonban lényegesen egyértelműbb. Ha mi magunk döntjük el, hogy pl. az ütést előnyben kell részesíteni egy általános lépéssel

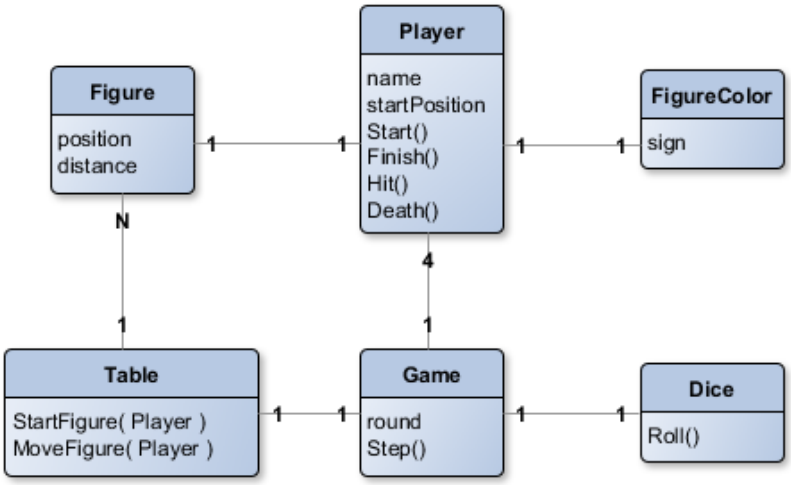
szemben, esetleg olyan komplexitást viszünk a rendszerbe, mely nem csupán indokolatlan, de plusz hibalehetőségeket is magában foglal. Az sem elhanyagolható, hogy a kérdéseinkre válaszul kaptunk egy ábrát, mely egyértelműsíti a pálya mezőinek számát és a start és cél mezők elhelyezkedését.

6.3.1. Implementációs ötletek számba vétele

A *Ludo* az a játék, mely tervezésekor talán először érezhetjük azt, hogy ezt bizony számos irányból meg lehet közelíteni, és ennek megfelelően implementálni. Van négy játékosunk, mindenkinek négy bábuja, melyek közül a 40 elemes táblán akár az összes, akár egy sem jelenik meg. A pályát modellezhetjük egy 40 elem hosszú tömbbel, melyben az elemek a figurák, és ahol épp nincs bábu, ott a tömb definiálatlan. Azon is érdemes lehet elgondolkodni, hogy nincsenek kitüntetett figurák, lehetne a játékosok referenciáit is mozgatni a pályán (ugyanazt a referenciát több helyen egyszerre). Egy időpillanatban azonban ritkán vannak sokan a pályán (maxium 16-an), ezért feleslegesnek tűnik egy fix 40 elemű tömb allokálása a feladatra. Gondolkodjunk figurák láncolt listájában, melyek jellemzője aktuális pozíciójuk, és leütésük vagy célba érkezésük esetén egyszerűen kiesnek a láncból.

Talán nincs olyan hogy "tökéletes" irány, mindegyik megoldásnak lehetnek előnyei és hátrányai is. Ha a megoldás betartja az objektum-orintált programozás alapvető elveit, redundancia mentes és tiszta (olvasható, karbantartható és továbbfejleszthető[9]), rendelkezik a követelmények ellenőrzésére képes egység tesztekkel, akkor a választott implementáció lehet másodlagos.

A 6.3. ábra egy előzetes terv osztályvázlatait tartalmazza. A játék osztály összefogja a dobókockát, a négy játékost és a táblát. Utóbbi egy halmaznyi figurát zár egységbe, akiknek pozíciójuk és megtett útjuk mellett természetesen jellemzőjük az is, melyik játékoshoz tartoznak. Minden játékosnak pedig van egy egyedi színe, mely most az egyszínű karaktergrafikus ábrázolásmód végett egy egyedi jel lesz.



6.3. ábra. Előzetes elképzelés

A választott implementációban a játék létrehozása egyben a négy játékos, a dobókocka és a tábla létrehozását is jelenteni fogja, azonban egyetlen figura sem fog még létezni. Elméletben legfeljebb 16 bábu létezhet egy időben, a gyakorlatban azonban ennél sokkal

kevesebbel kell foglalkoznunk. Számolni fogjuk az ütések és az "elhalálozások" számát is, hogy a végén a dobogós helyek kiosztásakor egy apró statisztikát tudjunk a lefutott szimuláció mellé adni. Bár sok véletlen tényezőtől függ a szimuláció, de azt várhatjuk, hogy a legeredményesebb játékos sok ellenséget ütött le, és kevésszer volt célpont (mindamellett hogy észben tartjuk azt is, hogy elképzelhető olyan forgatókönyv, melyben a többi játékost egymást felváltva lecsapva lesz befutó a nevető harmadik).

6.3.2. Dobókocka

A *dobókocka* osztály implementációja már ismerős lehet a 6.1. fejezetből. Egy egység tesztelésre kevésbé felkészített implementációt mutat be a 6.28. kódrészlet.

```
1  public class Dice
2  {
3    private const int MAX_VALUE = 6;
4
5    private readonly Random random;
6
7    public Dice(Random random)
8    {
9      this.random = random;
10   }
11
12   public int Roll()
13   {
14     return this.random.Next(Dice.MAX_VALUE) + 1;
15   }
16 }
```

6.28. kód. Dobókocka

6.3.3. Bábuszín

Szintén egy már korábban felmerült fejezetre utalunk vissza a *figurák színének* implementációjakor. Az 5.1. fejezetben bemutatásra került a felsorolás típusoknak egy referencia típust létrehozó variációja. Mivel a szín egy határozottan "fix" elemszámú konstans a társasjátékban, és minden ilyen színhez szeretnénk a karaktergrafikus ábrázolásmód szerint egy egyedi karakter jelet társítani, a 6.29. osztály kódja egy elegáns megoldás lehet.

```
1  public class FigureColor
2  {
3    public static readonly FigureColor Blue = new FigureColor('O');
4    public static readonly FigureColor Red = new FigureColor('X');
5    public static readonly FigureColor Green = new FigureColor('@');
6    public static readonly FigureColor Yellow = new FigureColor('#');
7
8    private static List<FigureColor> values;
9
10   private readonly char sign;
11
12   public static FigureColor[] Values
13   {
14     get
15     {
16       return values.ToArray();
17     }
18   }
19
20   public char Sign
21   {
22     get { return this.sign; }
23   }
```

```
24
25    private FigureColor(char sign)
26    {
27      this.sign = sign;
28      Add(this);
29    }
30
31    private static void Add(FigureColor item)
32    {
33      if (values == null)
34      {
35        values = new List<FigureColor>();
36      }
37      values.Add(item);
38    }
39
40    public override String ToString()
41    {
42      return "" + this.sign;
43    }
44  }
```

6.29. kód. Bábuszín

A *Ki nevet a végén?* megjelenítése során a 3.2. fejezethez hasonlóan, sorról sorra fogjuk kirajzolni a pályát és rajta a bábukat. Nem fogjuk alkalmazni a kurzor pozíció direkt beállítását, és a elő illetve háttérszínnel sem fogunk "játszani". Utóbbi bár rontja egy kicsit a vizuális élményt, a két megszorítás együttesen lehetővé teszi számunkra hogy a megvalósítandó osztályoknak *String* reprezentációt készítsünk, mely egy önálló felelősség, és jól illeszkedik az objektum-orientáltság által előírt felelősség témakörébe[15]. Bár "sorról sorra" megjeleníteni a társasjáték tábláját nem tűnik első ránézésre a legjobb ötletnek, egy olyan megoldást fogunk áttekinteni, ahol a megjelenítés algoritmusa a lehető legkevésbé sem a vizuális eredményt helyezi majd előtérbe (mert ellenkező esetben átláthatatlan kusza kód lenne az eredmény).

6.3.4. Játékos

Minden *játékosnak* lesz egy neve, illetve egy bábuszíne. Ezek bármelyike egyértelműen azonosítja. Valamilyen önkényes alapon a táblát körül fogják ülni, és mindenki ezáltal meghatározza, hogy kinek melyik lesz a "start" mezője. A játéktáblán összesen 40 mező van, a start mezők egymástól 10 lépés távolságra találhatóak. Minden mezőnek lesz egy sorszáma (0 és 39 között), mely a "start" mezőknek is azonosítható értéket ad. Utóbbit szintén eltároljuk a játékos osztályában, mint rá jellemző végleges mezőt (lásd. 6.30. kódrészlet).

Minden játékosnak két fontos állapota lesz: mennyi bábuja áll várakozva az indulásra, és mennyien vannak már a célban. Ezek a számok "virtuális figurákat" fognak jelképezni, mivel a játéktáblán nem szerepelnek[16], a darabszám elegendő a modellezésükre.

[15] Az osztálynak az a tipikusan **void** visszatérési értékkel rendelkező metódusa, mely a *Console.WriteLine()* osztályszintű metódust tömegesen hívja, kicsit kívülálló az osztály üzleti viselkedést leíró metódusai között. Egy *String* visszatérési értékkel rendelkező "átalakító" metódus viszont egy jól definiálható műveletet ad az osztályhoz: létrehozza annak karakterlánc alakját.

[16] Az "igazi" *Ki nevet a végén?* társasjáték tábláján természetesen szerepelnek ezek a mezők is, így a tábla valójában 40+16+16 mezőből áll. A megjelenítés során mi sem fogunk a plusz 32 mezőről megfeledkezni, azonban a szabályrendszerünk - mely a követelményben megfogalmazásra került - csak a 40 tábla mezőre vonatkozik, így ezen osztály felelősségének határa nyilvánvalóan erre fog építeni.

```
1  public class Player
2  {
3    private const int NUMBER_OF_FIGURES = 4;
4
5    private readonly String name;
6    private readonly FigureColor color;
7    private readonly int startPosition;
8
9    private int numOfFiguresAtStart;
10   private int numOfFiguresAtFinish;
11   private int numOfHits;
12   private int numOfDeaths;
13   private int winningRound;
14
15   public String Name
16   {
17     get { return this.name; }
18   }
19
20   public char Sign
21   {
22     get { return this.color.Sign; }
23   }
24
25   public int StartPosition
26   {
27     get { return this.startPosition; }
28   }
29
30   public int WinningRound
31   {
32     get { return this.winningRound; }
33   }
34
35   public Player(String name, FigureColor color, int startPosition)
36   {
37     this.name = name;
38     this.color = color;
39     this.startPosition = startPosition;
40
41     this.numOfFiguresAtStart = NUMBER_OF_FIGURES;
42     this.numOfFiguresAtFinish = 0;
43
44     this.numOfHits = 0;
45     this.numOfDeaths = 0;
46     this.winningRound = -1;
47   }
48
49   public Figure Start()
50   {
51     if (this.numOfFiguresAtStart > 0)
52     {
53       this.numOfFiguresAtStart--;
54     }
55     return new Figure(this);
56   }
57
58   public bool HasFigureAtStart()
59   {
60     return this.numOfFiguresAtStart > 0;
61   }
62
63   public void Finish()
64   {
65     if (this.numOfFiguresAtFinish < NUMBER_OF_FIGURES)
66     {
67       this.numOfFiguresAtFinish++;
68     }
69   }
70
71   public void Hit()
72   {
73     this.numOfHits++;
74   }
75
```

```
76    public void Death()
77    {
78      if (this.numOfFiguresAtStart < NUMBER_OF_FIGURES)
79      {
80        this.numOfFiguresAtStart++;
81        this.numOfDeaths++;
82      }
83    }
84
85    public bool IsFinish()
86    {
87      return this.numOfFiguresAtFinish == NUMBER_OF_FIGURES;
88    }
89
90    public void End(int round)
91    {
92      this.winningRound = round;
93    }
94
95  }
```

6.30. kód. Játékos

A szimuláció kedvéért néhány statisztikai adatot is gyűjt a játékos osztálya. Számolni fogjuk a játék során az ütések és az "elhalálozások" számát is, illetve ha valamely játékos minden figurája beérkezett a célba, a játék aktuális körének sorszámát is "elmentjük". A játék végén a dobogó megjelenítésekor így látszódni fog, ki mennyi "idő" alatt fejezte be a küldetést.

Az osztályt nyilvános metódusok segítségével fogjuk a játékon keresztül vezérelni. Ha hatost dob valaki a saját körében, és van még várakozó figurája (*HasFigureAtStart()* metódus), akkor útnak indíthatunk egy új bábut (*Start()* tagfüggvény). Utóbbi metódus olvasata: a játékos osztálya lesz a bábuk osztályának *factory*-ja! A játékos számára befejeződött a játék, ha az *IsFinish()* metódus igazat ad vissza (már négy bábuja a célban található). Az összes többi, név szerint a *Finish()*, *Hit()*, *Death()* és *End()* metódusok a megfelelő események hatásainak a monitorozására szolgálnak. Implementációjuk egyértelmű és egyszerű.

6.3.5. Bábu

A figurák példányai lesznek azok az entitások, amelyek "mozognak" a játéktáblán. Mindegyik pontosan egy játékkoshoz tartozik (aki létrehozza "őt"), illetve állapota az aktuális pozíciója, és a már megtett útja hossza lesz (lásd. 6.31. kódrészlet). Utóbbi azért fontos, mert minden bábu a tulajdonosa "start" mezőjén "születik" meg, ezért abból tudjuk majd legkönnyebben eldönteni, hogy az adott bábu "körbeért", ha már megtett útja nagyobb mint a pálya mezőinek száma (lásd. 36. sor, *IsHome()* metódus).

```
1  public class Figure
2  {
3    private readonly Player player;
4    private int position;
5    private int distance;
6
7    public Player Player
8    {
9      get { return this.player; }
10   }
11
12   public int Position
13   {
14     get { return this.position; }
15   }
```

```
16
17    public int Distance
18    {
19      get { return this.distance; }
20    }
21
22    public char Sign
23    {
24      get { return this.player.Sign; }
25    }
26
27    public Figure(Player player)
28    {
29      this.player = player;
30      this.position = player.StartPosition;
31      this.distance = 0;
32    }
33
34    public void SetPosition(int newPosition, int diceValue)
35    {
36      this.position = newPosition;
37      this.distance += diceValue;
38    }
39
40    public bool IsHome(int diceValue)
41    {
42      return this.distance + diceValue > Table.MAP_SIZE;
43    }
44
45  }
```

6.31. kód. Bábu

6.3.6. Tábla

A *játéktábla* tartalmazza a bábukat, kezdetben a játék szabályainak értelmében egyet sem (a játékosok összes bábúja várakozik az elindulásra). A tábla mezőinek számából egy egyszerű művelettel szolgáltatni tudja a játékosok "start" mezőinek távolságát (40 / 10 = 4), bár megjegyzendő, hogy ez konstansként is elfogadható művelet volna akár egy másik osztály felelősségeként (lásd. 6.32. kódrészlet).

```
1  public class Table
2  {
3    public const int MAP_SIZE = 40;
4
5    private readonly List<Figure> figures;
6    private readonly int playersDistance;
7
8    public int PlayersDistance
9    {
10     get { return this.playersDistance; }
11   }
12
13   public Table()
14   {
15     this.figures = new List<Figure>();
16     this.playersDistance = MAP_SIZE / Game.NUMBER_OF_PLAYERS;
17   }
18
19 }
```

6.32. kód. Játéktábla kerete

Az osztály legfontosabb feladatai az alábbiak:

1. Eldönteni, hogy egy játékos útnak indíthat-e új figurát

- Ha a játékos "start" mezőjén nem szerepel másik bábu, vagy
- A "start" mezőn ellenséges bábu szerepel (ez esetben leüthető, vagyis az útnak indítás lehetséges).

2. Elindítani a játékos bábuját
   - A játékost megkérni egy új bábu létrehozására
   - Kezelni szükséges a leütés esetét a játékos "start" mezőjén
     - Ha az adott mezőn van ellenséges figura, akkor azt el kell távolítani a tábláról, illetve
     - a támadó játékos leütési statisztikáját növelni szükséges, és
     - az ellenséges játékos "elhalálozási" eseményét ki szükséges váltani.
3. Mozgatni a játékos bábuját
   - Ki kell választani a játékos pályán lévő figurái közül azt, amelyikre prioritási sorrendben igaz, hogy
     - a dobott érték segítségével bejut a célba, vagy
     - a lehető legközelebb van a céltól, és a dobott értéknek megfelelő mezőre léphet
       - A mezőn nem szerepel másik bábu, vagy
       - az ott szereplő bábu ellenséges, és leütés mellett a mozgatás az adott mezőre lehetséges.

Kezdjük az implementációt a legelső felelősséggel: ha egy játékos hatost dob, és lehetséges a "start" mezőre való kilépés, akkor mindenképpen ezt a lépést kell a játékosnak választania a követelményrendszerben leírtak alapján. E speciális eset vizsgálatára készül el a tábla osztály *IsStartPossible(Player)* metódusa (lásd. 6.33. kódrészlet). A már játéktáblán lévő figurák közül ki kell választani azt, amelyik a paraméterben kapott (aktuális) játékos "start" mezőjén található éppen. Nem biztos, hogy van ott figura, ez esetben a kilépés természetesen lehetséges, azonban akkor is megtehetjük ezt, ha az ott lévő bábú ellenségnek minősül.

```
1  public class Table
2  {
3    [..]
4
5    public bool IsStartPossible(Player player)
6    {
7      Figure figure = this.FindFigure(player.StartPosition);
8      return this.IsFreePosition(player, figure);
9    }
10
11   private Figure FindFigure(int position)
12   {
13     Figure figure = null;
14     foreach (Figure current in this.figures)
15     {
16       if (current.Position == position)
17       {
18         figure = current;
19         break;
20       }
21     }
22     return figure;
23   }
24   private bool IsFreePosition(Player player, Figure figure)
25   {
```

```
26      return figure == null || figure.Player != player;
27    }
28
29  }
```

6.33. kód. Lehetséges az elindítás?

Ha lehetséges a kilépés, akkor ezt végrehajtja a tábla a nyilvános *StartFigure(Player)* metódus segítségével. Ha van a "start" mezőn ellenség, akkor azt le kell ütni, el kell távolítani a játéktábla által kezelt bábuk közül, és frissíteni szükséges a megfelelő statisztikákat és állapotokat (lásd. 6.34. kódrészlet). A korábban megvalósított *FindFigure() lineáris keresés*[1] remekül újra felhasználható ezen a ponton[17].

```
1  public class Table
2  {
3    [..]
4
5    public void StartFigure(Player player)
6    {
7      Figure enemy = this.FindFigure(player.StartPosition);
8      this.figures.Add(player.Start());
9      this.HandleHit(player, enemy);
10   }
11
12   private void HandleHit(Player player, Figure enemy)
13   {
14     if (enemy != null)
15     {
16       player.Hit();
17       enemy.Player.Death();
18       this.figures.Remove(enemy);
19     }
20   }
21
22 }
```

6.34. kód. Figura elindítása

A utolsó megvalósítandó üzleti művelet a játéktáblán a figurák mozgatása. A *MoveFigure()* metódus argumentumként megkapja az aktuális játékost, illetve az általa dobott értéket. Ha a versenyző képes a lépésre, akkor a lehetséges lépései közül a követelményrendszerben leírt prioritási sorrend szerint kell választania! Ennek egyik kulcseleme a céltól való távolság. Szükségünk van a játéktáblán az aktuális játékos összes bábujának *kiválogatására*[1] (lásd. 6.36. kódrészlet *Assortment()* metódusa), majd ezen bábuk saját megtett útjuk alapján történő rendezésére! Utóbbi egy érdekes implementációs kérdés, lévén megtehetjük, hogy írunk egy rendezési algoritmust e tulajdonságra nézve, ám ennél sokkal elegánsabb, ha készítünk egy célnak megfelelő *Comparer* osztályt (lásd. 6.35. kódrészlet).

```
1  public class FigureDistanceComparer : IComparer<Figure>
2  {
3    public int Compare(Figure figure1, Figure figure2)
4    {
```

[17] A végrehajtás egy szálon mindig úgy fog történni, hogy a *StartFigure(Player)* metódust csak akkor fogja meghívni a játékot kezelő osztály, ha előtte az *IsStartFigure(Player)* metódus igazat adott vissza. A két metódus egybezárható, esetleg a felesleges második lineáris keresés megspórolható az *IsStartFigure(Player)* által dobott kivétel segítségével. Bármely választott megoldás hatékonyabb volna, azonban a kód olvashatósága romlana. A jegyzetben ez egy olyan szituáció, ahol a kód olvashatósága előnyt élvez a teljesítmény felett. Ritka és legtöbbször kikerülhető szituáció ez, azonban felhívja a figyelmet arra, hogy érdemes-e vajon néha a hatékonyságot beáldozni az olvashatóság oltárán?

```
5       return figure1.Position - figure2.Position;
6     }
7   }
```

6.35. kód. Megtett út alapján csökkenő rendezéshez

A *MoveFigure()* metódus (lásd. 6.36. kódrészlet) implementációja a kiválogatás és rendezést követően egy "történet elmesélése" kell hogy legyen, ha egy szakmabeli rátekint. A kapott szabályrendszerből következik néhány futási ág, mely nagyon könnyen vezethet kusza, ún. *spagetti kódhoz*. Ha képesek vagyunk az imperatív nyelv eszközeivel olyan közel deklaratív folyamatot leírni, ami azonosítható a szabályrendszerrel, akkor úgy vélem jó úton haladunk a karbantartható és továbbfejleszthető kód írása felé.

```
1   public class Table
2   {
3     [..]
4
5     public void MoveFigure(Player player, int diceValue)
6     {
7       List<Figure> playerFigures = this.Assortment(player);
8       playerFigures.Sort(new FigureDistanceComparer());
9       foreach (Figure figure in playerFigures)
10      {
11        if (figure.IsHome(diceValue))
12        {
13          figure.Player.Finish();
14          this.figures.Remove(figure);
15          break;
16        }
17        else
18        {
19          int position = this.CalculateRealPosition(figure.Position + diceValue);
20          Figure enemy = this.FindFigure(position);
21          if (this.IsFreePosition(player, enemy))
22          {
23            figure.SetPosition(position, diceValue);
24            this.HandleHit(player, enemy);
25            break;
26          }
27        }
28      }
29    }
30
31    private List<Figure> Assortment(Player player)
32    {
33      List<Figure> figures = new List<Figure>();
34      foreach (Figure current in this.figures)
35      {
36        if (current.Player.Equals(player))
37        {
38          figures.Add(current);
39        }
40      }
41      return figures;
42    }
43
44    private int CalculateRealPosition(int position)
45    {
46      return position >= MAP_SIZE ? position - MAP_SIZE : position;
47    }
48
49  }
```

6.36. kód. Kivétel

Próbáljuk meg felolvasni magunknak a saját implementációnkat. Ha egy adott sor értelmezése nem automatikus, akkor ott még csiszolni szükséges valamin. Jó módszer lehet az összes feltételnek valamilyen "név" adása azáltal, hogy a feltételt kiszervezzük

egy **private** metódusba. Figyeljünk arra is, hogy a feltétel vizsgálat ez esetben ne negált legyen[18]! Mindig az elnevezéseken van a hangsúly. A *FindFigure()* visszatérési értéke "enemy" legyen, hiszen ha van ellenség, akkor azt le kell ütni. Minden sokkal logikusabban hangzik, mintha ha mindezt 'a'-nak, vagy általánosan 'figure'-nak nevezzük el. A szerző bízik abban, hogy az olvashatóság javítása a jegyzet ezen részén már nem jelenti senki számára azt, hogy *inline* kommenteket tűzdel a sorok közé, mögé.

6.3.7. Játék

A *játék* osztályt a szimuláció során a következőképpen fogjuk felhasználni: a *Step()* metódusát addig hívjuk, amíg az *IsFinish()* metódusa nem lesz igaz. (Érdekességként megjegyezhető, hogy ez egyfajta implementációja az *iterator design pattern*-nek[6]). A játék akkor fog véget érni, ha a dobogó mindhárom foka már betöltött (lásd. 6.37. kódrészlet), így hát a *Step()* metódus implementációja lesz az érdekesebb.

```
1  public class Game
2  {
3
4    public const int NUMBER_OF_PLAYERS = 4;
5    private const int START_DICE_VALUE = 6;
6
7    private readonly Dice dice;
8    private readonly Player[] players;
9    private readonly Table table;
10   private int playerIndex;
11   private int currentPlayerIndex;
12   private readonly Player[] palpitating;
13   private int palpitatingIndex;
14   private int round;
15
16   private Player CurrentPlayer
17   {
18     get
19     {
20       return this.players[this.currentPlayerIndex];
21     }
22   }
23
24   public Game(Random random)
25   {
26     this.dice = new Dice(random);
27     this.players = new Player[NUMBER_OF_PLAYERS];
28     this.table = new Table();
29     this.playerIndex = 0;
30     this.currentPlayerIndex = 0;
31     this.palpitating = new Player[3];
32     this.palpitatingIndex = 0;
33     this.round = 1;
34   }
35
36   public void AddPlayers(String first, String second, String third, String forth)
37   {
38     this.AddPlayer(first);
39     this.AddPlayer(second);
40     this.AddPlayer(third);
41     this.AddPlayer(forth);
42   }
43
44   private void AddPlayer(String name)
45   {
46     if (this.playerIndex < this.players.Length)
```

[18] Ha van egy olyan kódrészletünk, miszerint valamilyen érték ha nem páros, akkor végre kell hajtanunk néhány utasítást, akkor az "if (value % 2!= 0) ..." átírása ne az legyen, hogy "if (!IsEven(value)) ...", hanem válasszuk az "if (IsOdd(value)) ..." megoldást, vagyis törekedjünk arra, hogy feleslegesen ne használjuk a '!' operátort.

```
47      {
48        this.players[this.playerIndex] = new Player(name,
              FigureColor.Values[this.playerIndex], this.table.PlayersDistance *
              this.playerIndex);
49        this.playerIndex++;
50      }
51    }
52
53    public bool IsFinish()
54    {
55      return this.palpitatingIndex == 3;
56    }
57
58  }
```

6.37. kód. Játék osztály

Ahogyan a táblán a figurák mozgatása során megfogalmaztuk a követelményrendszerből kihámozott futási ágakat, tegyük ezt meg a lépés implementációja előtt is. Mit is kell tennünk, mikor egy lépést végre kell hajtania a játéknak?

1. Ellenőrizzük, hogy az aktuális játékos még játékban van-e. Ha már nincs (vagyis fenn áll a dobogó egyik fokán), akkor a következő játékosra lépünk.
2. Dobunk a dobókockával.
3. Ha lehetséges új bábu elindítása, akkor ezt megtesszük.
4. Ha lehetséges a mozgatás, akkor meglépjük a lehető legjobb variációt (korábban implementált *MoveFigure()* metódus felelőssége).
5. Ha a mozgatás után a játékos minden figurája a célba ért, akkor felállítjuk a játékost a dobogó következő fokára.
6. Ha a dobott érték hatos volt, akkor a játékos ismét dobhat, ellenkező esetben a következő versenyző kerül sorra.

Ha meg tudjuk fogalmazni az elvárt futási ágakat (egy nagyon magas szinten definiált pszeudó kódot írunk valójában), akkor az implementáció sem fog problémát okozni, nem utolsó sorban minden elágazásnak, ciklusmagnak fogunk tudni egy releváns nevet adni, ezáltal növelve a kód olvashatóságát (lásd. 6.38. kódrészlet).

```
1  public class Game
2  {
3    [..]
4
5    public String Step()
6    {
7      StringBuilder info = new StringBuilder(30);
8      Player player = this.CurrentPlayer;
9      info.Append("R-").Append(this.round).Append(" ").Append(player);
10     if (!player.IsFinish())
11     {
12       int diceValue = this.dice.Roll();
13       info.Append(" Dice: ").Append(diceValue);
14       if (this.IsStartFigurePossible(player, diceValue))
15       {
16         this.table.StartFigure(player);
17         info.Append(" START ");
18       }
19       else
20       {
21         this.table.MoveFigure(player, diceValue);
```

```
22          info.Append(" STEP ");
23          if (player.IsFinish())
24          {
25            this.AddItemToPalpitating(player);
26          }
27        }
28        if (diceValue != START_DICE_VALUE)
29        {
30          this.NextPlayer();
31        }
32      }
33      else
34      {
35        info.Append(" SKIP");
36        this.NextPlayer();
37      }
38      return info.ToString();
39    }
40
41    private bool IsStartFigurePossible(Player player, int diceValue)
42    {
43      return diceValue == START_DICE_VALUE && player.HasFiguresAtStart() &&
            this.table.IsStartPossible(player);
44    }
45
46    private void AddItemToPalpitating(Player player)
47    {
48      if (this.palpitatingIndex < this.palpitating.Length)
49      {
50        this.palpitating[this.palpitatingIndex++] = player;
51        player.End(this.round);
52      }
53    }
54
55    private void NextPlayer()
56    {
57      if (this.currentPlayerIndex < this.players.Length - 1)
58      {
59        this.currentPlayerIndex++;
60      }
61      else
62      {
63        this.currentPlayerIndex = 0;
64        this.round++;
65      }
66    }
67
68  }
```

6.38. kód. Lépés

6.3.8. Megjelenítés

Sorfolytonos megjelenítéshez leginkább egy mátrix bejárása hasonlítható. Soronként, azon belül pedig oszloponként minden egyes karakter pozíción el szükséges dönteni, hogy kell-e, és ha kell mit kell megjeleníteni. A tábla 40 mezője egy zárt alakzatot alkot, bár természetesen az algoritmus működése szempontjából mindez lényegtelen. Igen körülményes és áttekinthetetlen volna ha speciális elágazásokkal, esetleg vektorokkal próbálnánk leírni a pályát, illetve a játék szempontjából szintén lényeges elemeket (kik és mennyien várakoznak indulásra, kik értek már be a célba). Sokszor jó módszer lehet egy vizuális hatást valamilyen forráskódban is vizuálisan felírható modellel definiálni. Ehhez első lépésként vegyük számba, milyen különböző mezők szerepelnek a játéktáblán (a valóságban)[19]:

[19] Elsősorban attól lesz különböző típusú egy mező, ha azt valamilyen formában máshogy kell kezelni az alkalmazásban. De mindez kontextus függő. A *Ludo* játék a bemutatottól eltérő implementációjában más típusok is számba vehetőek.

- A játéktábla mezői (40 darab)
- A várakozási mezők (16 darab, játékosonként 4 darab)
- A cél mezők (16 darab, játékosonként 4 darab)

Ha a felsorolásunk két elemű, akkor valamilyen logikai adatszerkezet elegendő lehet. Ha ennél több, akkor valamilyen egész típus jöhet szóba. Kód olvashatóság szempontjából a **char** típus általában jó választás. A 6.39. kódrészlet egy - a tábla osztály megjelenítését vezérlő - külön osztály, a *TableView* részletét tartalmazza. A mátrix literál felvétele néhány kiírt sor végi megjegyzés jel elhelyezésével egy olyan leírást ad a társasjáték táblájának, melyet nagyon könnyű elképzelni "futás közben is".

```
1  public class TableView
2  {
3
4    public const char FIELD = 'F';
5    public const char START = 'S';
6    public const char DESTINATION = 'D';
7
8    public static readonly char[][] TYPE = { //
9      new char[]{ 'S', ' ', 'S', ' ', 'F', 'F', 'F', ' ', 'S', ' ', 'S' }, //
10     new char[]{ ' ', ' ', ' ', ' ', 'F', 'D', 'F', ' ', ' ', ' ', ' ' }, //
11     new char[]{ 'S', ' ', 'S', ' ', 'F', 'D', 'F', ' ', 'S', ' ', 'S' }, //
12     new char[]{ ' ', ' ', ' ', ' ', 'F', 'D', 'F', ' ', ' ', ' ', ' ' }, //
13     new char[]{ 'F', 'F', 'F', 'F', 'F', 'D', 'F', 'F', 'F', 'F', 'F' }, //
14     new char[]{ 'F', 'D', 'D', 'D', 'D', ' ', 'D', 'D', 'D', 'D', 'F' }, //
15     new char[]{ 'F', 'F', 'F', 'F', 'F', 'D', 'F', 'F', 'F', 'F', 'F' }, //
16     new char[]{ ' ', ' ', ' ', ' ', 'F', 'D', 'F', ' ', ' ', ' ', ' ' }, //
17     new char[]{ 'S', ' ', 'S', ' ', 'F', 'D', 'F', ' ', 'S', ' ', 'S' }, //
18     new char[]{ ' ', ' ', ' ', ' ', 'F', 'D', 'F', ' ', ' ', ' ', ' ' }, //
19     new char[]{ 'S', ' ', 'S', ' ', 'F', 'F', 'F', ' ', 'S', ' ', 'S' }, //
20   };
21
22 }
```

6.39. kód. Különféle típusú elemek a táblán

Minden egyes mezőnek lehetnek kiegészítő tulajdonságaik (ezek egy része bevonható volna a típusba), melyeket teljesen hasonló módon leírhatunk. Kiegészítő tulajdonság a sorrendiség (az azonos típusú mezők egyértelmű megkülönböztetése), illetve a "tulajdonos személye", vagyis kihez tartozik a mező (lásd. 6.40. kódrészlet).

```
1  public class TableView
2  {
3    [..]
4
5    public static readonly int[][] OWNER = { //
6  //
7      new int[]{ 1, 0, 1, 0, 0, 0, 0, 0, 4, 0, 4 }, //
8      new int[]{ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 }, //
9      new int[]{ 1, 0, 1, 0, 0, 1, 0, 0, 4, 0, 4 }, //
10     new int[]{ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 }, //
11     new int[]{ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 }, //
12     new int[]{ 0, 2, 2, 2, 2, 0, 4, 4, 4, 4, 0 }, //
13     new int[]{ 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0 }, //
14     new int[]{ 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0 }, //
15     new int[]{ 2, 0, 2, 0, 0, 3, 0, 0, 3, 0, 3 }, //
16     new int[]{ 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0 }, //
17     new int[]{ 2, 0, 2, 0, 0, 0, 0, 0, 3, 0, 3 } //
18   };
19
20   public static readonly int[][] INDEX = { //
21 //
22     new int[]{ 0, -1, 1, -1, 0, 39, 38, -1, 1, -1, 0 }, //
23     new int[]{ -1, -1, -1, -1, 1, 3, 37, -1, -1, -1, -1 }, //
24     new int[]{ 2, -1, 3, -1, 2, 2, 36, -1, 3, -1, 2 }, //
```

```
25      new int[]{ -1, -1, -1, -1, 3, 1, 35, -1, -1, -1, -1 }, //
26      new int[]{ 8, 7, 6, 5, 4, 0, 34, 33, 32, 31, 30 }, //
27      new int[]{ 9, 3, 2, 1, 0, -1, 0, 1, 2, 3, 29 }, //
28      new int[]{ 10, 11, 12, 13, 14, 0, 24, 25, 26, 27, 28 }, //
29      new int[]{ -1, -1, -1, -1, 15, 1, 23, -1, -1, -1, -1 }, //
30      new int[]{ 2, -1, 3, -1, 16, 2, 22, -1, 3, -1, 2 }, //
31      new int[]{ -1, -1, -1, -1, 17, 3, 21, -1, -1, -1, -1 }, //
32      new int[]{ 0, -1, 1, -1, 18, 19, 20, -1, 1, -1, 0 }, //
33    };
34
35  }
```

6.40. kód. Kivétel

A mátrix leírások lényege a könnyű feldolgozhatóság. A játék megjelenítése nem fog másról szólni, mint a literálként leírt mátrix bejárása, és a különféle típusú megjelenítendő mezők során egyedi akciók végrehajtása. Definiáljuk, hogy melyik mezőt miképpen fogjuk megjeleníteni:

- A játéktábla mezői: '[ ]' vagy '[@]' ha van rajta játékos, ahol '@' a játékos jele/színe
- A várakozási mezők: '( )' vagy '(@)' ha várakozik rajta játékos, ahol '@' a játékos jele/színe
- A cél mezők: ' ' vagy ' @ ' ha célba ért a játékos, ahol '@' a játékos jele/színe

Mivel a játéktábla szabályos négyszög (négyzet), ezért törekedjünk arra, hogy minden típus megjelenítése azonos karakterhosszúságú legyen (jelen esetben három hosszúak). A játékoshoz tartozó start és cél figurák kirajzolásához létrehozhatunk segédfüggvényeket a *Player* osztályban (lásd. 6.41. kódrészlet).

```
1  public class Player
2  {
3    [..]
4
5    public String PrintStartFigure(int index)
6    {
7      return "(" + this.PrintFigure(index, this.numOfFiguresAtStart) + ")";
8    }
9
10   public String PrintFinishFigure(int index)
11   {
12     return " " + this.PrintFigure(index, this.numOfFiguresAtFinish) + " ";
13   }
14
15   private char PrintFigure(int index, int numOfFigures)
16   {
17     return numOfFigures >= index + 1 ? this.Sign : ' ';
18   }
19
20   public override String ToString()
21   {
22     return this.name + " (" + this.Sign + ") Hit: " + this.numOfHits + " Death: " +
         this.numOfDeaths;
23   }
24 }
```

6.41. kód. Kivétel

A játéktábla megjelenítése a *Table* osztály *Print()* metódusának a felelőssége (lásd. 6.42. kódrészlet). A korábban létrehozott mátrixokat bejárva minden egyes mező típusának függvényében kirajzoljuk a játékos jelét a megfelelő helyekre. A tulajdonos adatokat leíró mátrix segít minket abban, hogy megcímezzük a játékosok tömbjét, az indexeket tartalmazó mátrix pedig az ezen belüli pozicionálást teszi lehetővé.

```
public class Table
{
  [..]

  public String Print(Player[] players)
  {
    StringBuilder builder = new StringBuilder(100);
    for (int i = 0; i < TableView.TYPE.Length; i++)
    {
      for (int k = 0; k < TableView.TYPE[i].Length; k++)
      {
        char type = TableView.TYPE[i][k];
        int owner = TableView.OWNER[i][k];
        int index = TableView.INDEX[i][k];
        switch (type)
        {
          case TableView.FIELD:
            Figure figure = this.FindFigure(index);
            builder.Append('[').Append(figure != null ? figure.Sign : ' ').Append(']');
            break;
          case TableView.START:
            builder.Append(players[owner - 1].PrintStartFigure(index));
            break;
          case TableView.DESTINATION:
            builder.Append(players[owner - 1].PrintFinishFigure(index));
            break;
          default:
            builder.Append("   ");
            break;
        }
      }
      builder.AppendLine();
    }
    return builder.ToString();
  }
}
```

6.42. kód. Kivétel

A játék legvégén a szimuláció során érdemes megjeleníteni a dobogós játékosokat és azok statisztikáit (lsád. 6.43. kódrészlet).

```
public class Game
{

  [..]

  public String PrintPalpitating()
  {
    StringBuilder builder = new StringBuilder(50);
    for (int i = 0; i < this.palpitating.Length; i++)
    {
      builder.Append("[" + (i + 1) + "] ").Append(this.palpitating[i]).Append("
          R-").Append(this.palpitating[i].WinningRound).Append("\n");
    }
    return builder.ToString();
  }

  public override String ToString()
  {
    return this.table.Print(this.players);
  }

}
```

6.43. kód. Kivétel

6.3.9. Szimuláció

A játék szimulációja során addig lépegetünk a játékosokkal, amíg a játék véget nem ér. Utóbbi a dobogós helyek kiosztását ellenőrzi. Bár elméletben elképzelhető, hogy a játékosok folyamatosan leütik egymást, bízhatunk benne hogy a ciklusunk egyszer véget fog érni. A játék osztály *Step()* metódusa visszaad egy karakterláncot, mely leírja az adott lépésben történt eseményeket (aktuális játékos dobásának értéke, és akciója, melyet ezzel végrehajtott).

```
1  private static void GameSimulation(Random random)
2  {
3    Game game = new Game(random);
4    game.AddPlayers("Alice", "Bob", "Charlie", "Delta");
5
6    while (!game.IsFinish())
7    {
8      Console.Clear();
9      Console.WriteLine(game.Step());
10     Console.WriteLine(game);
11     Thread.Sleep(100);
12   }
13   Console.WriteLine();
14   Console.WriteLine(game.PrintPalpitating());
15   Console.ReadKey();
16 }
```

6.44. kód. Kivétel

Számos ponton lehet még csiszolni az elkészült *Ki nevet a végén?* társasjáték szimulációján. Az eredeti játékban a célba éréshez pontos dobás szükséges (illetve a cél mezőkön is lehet mozogni, de csak a mezők gazdájának), és elképzelhető hogy a hatos dobások esetén nem "végtelenszer" dobhat ismét a játékos. Az is egy érdekes módosítás volna, ha egyfajta "személyiséget" adunk a játékosoknak. Pl. Alice és Bob egymás leütésére játszanak (szimuláljuk hogy nem szeretik egymást), miközben Charlie Bob-ra szintén utazik, de Deltát semmiképpen nem üti le, csak akkor ha mást nem tud lépni. A szimulációkat nagy számban futtatva érdekes jelenségeket lehetne megfigyelni, majd ezt követően keresni mindennek a matematikai magyarázatát.

7. fejezet

Tervezési minták

Ismeretszerzés

formalizált leírással rendelkező *best-practice* technikák gyakorlati implementációja, szálkezelés és szinkronizálás alapjai

A tervezési minták, vagy ahogy sokszor "magyarul" is említik, a *design patternek* világa a *programozási tételek* mellett a másik olyan tényező, melynek alapos ismerete elengedhetetlen ahhoz, hogy megfelelő minőségű alkalmazásokat készítsünk. Fontos különbség, hogy a tervezési minták elsősorban az objektum-orientált nyelvekhez készültek, így elsősorban ezekkel való munka során hasznos [1], azonban az is lényeges különbség, hogy a tervezési mintáknak általában nem létezik egy "működő" demoja.

Egy *lineáris keresés* algoritmusához készíthetünk egy metódust, mely pl. egy karakterláncokat tartalmazó tömbben megkeresi az első személynevet, az algoritmus lényegi része nem abból fog állni, hogy melyik szó számít személynévnek, ez csupán egy apró feltétel lesz a kódsorok között. A programozási tétel minta algoritmusát legtöbbször könnyedén átemeljük egy másik kontextusba, és ebből kifolyólag a fejlesztő is könnyen azonosítja a mintákat (ha nem tudja azonosítani, az már a kód tisztaságának[9] a problémája).

A tervezési minták inkább szólnak egy gyakorlati probléma elvi megoldásáról, mintsem konkrét implementációról. Egy adott szituáció megoldásaként alkalmazott tervezési minta legtöbbször számos egyedi, kontextusfüggő változást igényel, ezért nehéz volna egy "demo" alkalmazásból kihámozni a lényegi részt. Egy megvalósított *design pattern* ennek ellenére a szakember számára azonnal azonosíthatóvá válik, és pontosan az fogja növelni a kód érthetőségét, hogy pl. felismeri a *Composite Design Pattern*-t (lásd. 7.2. fejezet), és pl. a kód továbbfejlesztése végett erre már tud építeni.

A tervezési mintákat szokás csoportosítani, de a jegyzet ettől most eltekint. Nem cél első körben az összes tervezési mintát megismerni, elegendő néhány megismerése annak érdekében, hogy saját bőrünkön érezzük azok hasznosságát. Nem várható el, hogy több tucat, sokszor számos osztály implementációját igénylő kódot memorizáljon bárki is cél nélkül. Ha ráérez a fejlesztő ezek hasznosságára, ennek többszörösére is képes lesz önként.

[1] Bár a legtöbb nem oo nyelvben is megvalósíthatóak az objektum-orientált elvek, e végett a tervezési minták értelmezése és haszna is lehet szélesebb körű.

7.1. Singleton design pattern

Előfeltétel

osztályszintű felelősség, referencia típusú felsorolás típus, generikus listák, generikus szótárak, indexer, kivételkezelés, programozási tételek

Készítsünk egy egyszerű pénz átutalást szimuláló alkalmazást. A feladat során tároljunk el a pénzszámlákat (számlaszám, deviza nem, érték), és valósítsuk meg az átutalást oly módon, hogy az utalandó összeg a forrás számláról eltűnik, míg a cél számlán a számla saját deviza nemében megjelenik! Az árfolyamok egy e célt szolgáló osztálytól legyenek lekérdezhetőek az átutalás pillanatában (az aktuális árfolyamokat HUF-ban tároljuk, pl. 1 EUR = 310 HUF, 1 HUF = 1 HUF, 1 CHF = 250 HUF). Legyen lehetőségünk az árfolyamokat bárhol módosítani az alkalmazásban, és ennek hatása a következő átutalásnál már érvényesüljön.

A valóságban a bankok bár foglalkoznak pénzváltással, és ehhez a törvény által meghatározott keretek között saját árfolyamokat határoznak meg, az árfolyamokat nem önkényesen "találják ki". Ezeket más piaci események és szabályok vezérlik, melyeket a bankok rendszerei átvesznek, majd a bank adott gyakorisággal meghatározza a saját átváltási értékeit. Ha szeretnénk ezt a felelősség szétválasztást az objektum-orientált világban is érvényesíteni, az aktuális árfolyamokat és a bankok által kezelt pénzmozgásokat külön rendszerben valósítjuk meg. Kicsit szakszerűbben fogalmazva: nem alkalmazunk aggregációt a létrejövő osztályok között, mivel egyikre sem igaz az, hogy tartalmazza a másikat.

Értelemszerűen adódik a kérdés: miként fognak az osztályok kommunikálni egymással? Talán eseménykezelés lenne a helyes megvalósítás? Nem valószínű. Bár az árfolyam változása kiválthat egy eseményt, melyre a bankok feliratkozhatnak, és elindíthatják saját üzleti folyamataikat e téren, de itt most konkrét pénzmozgások szimulálása során ez nem feladat. Akkor és ott van szükségünk a pontos árfolyam értékekre, amikor a pénzmozgás megtörténik. A legegyszerűbb megoldás az volna, ha az átutalási művelet paraméterként kapná meg az aktuális árfolyamokat kezelő osztály példányát. Ez az asszociáció bár működik, problémás volna elérni azt, hogy minden pontján az alkalmazásnak, ahol átutalást kezdeményeznek, ezen példány elérhető legyen.

Mi a programozási probléma, amit itt meg kell oldani? Elsősorban az, hogy van egy olyan referenciánk (az árfolyamokat kezelő osztály példánya), melyet bárhol az alkalmazás életében elérhetővé kell tennünk. Vegyük észre, hogy egy olyan speciális példányról beszélünk, melyből mind a valóságban, mind a szimulált világban egyetlen egy darab létezik. Abból az osztályból, mely leírja az árfolyamok kezelését, egyetlen egyszer fogunk példányt készíteni, és utóbbit kellene elérhetővé tennünk mindenki más számára. Ha véletlenül valaki létrehozna még egy példányt, az súlyos hiba volna, hiszen egy időben létezhetne egy példány, melyben 1 EUR 300 Forint, míg egy alternatív valóságban csak ötven.

7.1.1. Definíció

Az olyan osztályt, melyből egyetlen egy példányt hozunk létre, és biztosítjuk azt, hogy ez, és kizárólag ez a példány lesz elérhető mások számára, *singletonnak* nevezzük[2].

[2] Magyar fordításban *egykeként* szoktak hivatkozni a *singletonokra*.

A legfontosabb e cél elérése érdekében, hogy felügyeljük a példányok létrehozását. Ezt a következőképpen tudjuk megtenni:

- A konstruktor láthatóságát **private**-ra állítjuk. Csak és kizárólag az adott osztályon belül vagyunk képesek ezt követően példányosítani az osztályt. A példányosítást ezzel a leszármazottak elől is elvágjuk (hacsak nem készítünk egy *protected overload*-olt konstruktort, de *singleton* esetében ez természetesen nem célszerű).
- Az osztály példánymetódusaiból bár tudnánk létrehozni új példányt, ezzel tyúktojás problémába ütköznénk (a példánymetódus meghívásához már szükség volna előzőleg egy létrehozott példányra), illetve rögtön két példányunk lenne, ha mindez sikerülne.
- Kizárólag statikus metódusokból hozhatunk létre új példányt. Egy nyilvános, tipikusan *GetInstance()* elnevezésű osztályszintű metódust fogunk létrehozni e célból, mely a saját osztálya egy, s egyben egyetlen példányát hozza létre.
- Az egyetlen példányt az osztály saját magába aggregálja egy osztályszintű **private** láthatóságú adattag képében.
- Ha az osztály klónozható, akkor megtiltjuk klónozhatóságát, pl. egy kivétel dobásával.

A *singleton* osztályoknak a felsorolt szabályok alkalmazását követően semmiben sem kell másként viselkedniük, mint hagyományos társaik. Bár természetüknél fogva általában nem képezünk belőlük hierarchiát, bármely más nyelvi lehetőség nyitott számunkra az implementáció során. Nyilvános példány szintű metódusok segítségével valósítjuk meg az üzleti igényeknek megfelelő viselkedést.

7.1.2. Pénzszámla és deviza nem

Miután megvalósítottuk a *pénzszámlát* modellező entitásunkat (lásd. 7.1. kódrészlet), mely példánya egy egyedi számlaszámot (*number*), deviza nemet (*currency*, felsorolás érték, lásd. 7.2. kódrészlet[3]) és egyenleget (*value*) zár egységbe, illetve *Transfer()* metódusán keresztül lehetőséget biztosít a számlaérték előjel helyes felügyelt módosítására, elkezdhetjük a korábban leírt korlátozásoknak megfelelő *singleton* osztály megtervezését.

```
1  public class Account
2  {
3    private readonly String number;
4    private readonly Currency currency;
5    private double value;
6
7    public String Number
8    {
9      get { return this.number; }
10   }
11
12   public Currency Currency
```

[3] A bemutatott implementáció referencia típusként valósítja meg a deviza nemet annak érdekében, hogy a felsorolt elemekhez minden esetben hozzá tudjuk kötni az alkalmazás indulásakor érvényes árfolyamot. A programozási technika leírását az 5.1. fejezet írja le bővebben. A megoldás természetesen nem mutat állapottartó viselkedést. Ha az alkalmazást újraindítják, minden esetben visszaáll az "eredeti" árfolyam. Megoldás lehetne az alkalmazás indulásakor valamilyen online szolgáltatástól elkérni a pontos értékeket, illetve akár erre feliratkozva vezérelni a változásokat.

```
13    {
14      get { return this.currency; }
15    }
16
17    public double Value
18    {
19      get { return this.value; }
20    }
21
22    public Account(String number, Currency currency, double value)
23    {
24      this.number = number;
25      this.currency = currency;
26      this.value = value;
27    }
28
29    public void Transfer(double value)
30    {
31      this.value += value;
32    }
33
34    public override string ToString()
35    {
36      return this.number + ": " + this.value + " " + this.currency;
37    }
38
39  }
```

7.1. kód. Pénzszámla

```
1   public class Currency
2   {
3     public static readonly Currency EUR = new Currency("EUR", 310);
4     public static readonly Currency HUF = new Currency("HUF", 1);
5     public static readonly Currency CHF = new Currency("CHF", 250);
6
7     private static List<Currency> values;
8
9     private readonly String name;
10    private readonly double initialExchangeRate;
11
12    public static Currency[] Values
13    {
14      get
15      {
16        return values.ToArray();
17      }
18    }
19
20    public String Name
21    {
22      get { return this.name; }
23    }
24
25    public double InitialExchangeRate
26    {
27      get { return this.initialExchangeRate; }
28    }
29
30    private Currency(String name, double initialExchangeRate)
31    {
32      this.name = name;
33      this.initialExchangeRate = initialExchangeRate;
34      add(this);
35    }
36
37    private static void add(Currency item)
38    {
39      if (values == null)
40      {
41        values = new List<Currency>();
42      }
43      values.Add(item);
44    }
```

```
45
46    public override String ToString()
47    {
48      return this.name;
49    }
50  }
```

7.2. kód. Deviza

7.1.3. A singleton osztály

Az *ExchangeRateHolder* osztály hordozza magában a *singleton design pattern* által megfogalmazott ismérveket (lásd. 7.3. kódrészlet). Osztályszintű *GetInstance()* elnevezésű metódusa (35. sor) saját maga példányával tér vissza, melyet egyébiránt aggregálva tárol egy értelemszerűen osztályszintű adattag formájában (3. sor). E mező neve tipikusan INSTANCE szokott lenni, kezdeti értéke **null**. A *GetInstance()* metódust legelőször meghívó kliens esetén lesz kizárólag igaz a 37. sorban felállított feltétel, mely hatására az INSTANCE mezőt példányosítjuk. Az ezt követő hívások során a feltétel már nem fog teljesülni, és minden esetben a legelőször létrehozott - egyetlen - példányt kapja vissza a kliens.

Az osztálynak a szabályok értelmében **private** konstruktora van (lásd. 7.3. kódrészlet 19. sor), mely a példány árfolyamokat tároló szótárát inicializálja az alkalmazásban elérhető pénznemek és azok kezdeti értékei szerint. A példányszintű *Convert()* üzleti metódus használható arra, hogy egy tetszőleges pénznemben meghatározott értéket egy másik pénznemre összegszerűen átalakítson. Ez az a szolgáltatás, melyet az alkalmazásban bárhol el kell tudnunk érni.

```
1  public class ExchangeRateHolder
2  {
3    private static ExchangeRateHolder INSTANCE;
4
5    private readonly Dictionary<Currency, Double> rates;
6
7    public double this[Currency key]
8    {
9      get
10     {
11       return this.rates[key];
12     }
13     set
14     {
15       this.rates[key] = value;
16     }
17   }
18
19   private ExchangeRateHolder()
20   {
21     this.rates = new Dictionary<Currency, Double>();
22     foreach (Currency currency in Currency.Values)
23     {
24       this.rates.Add(currency, currency.InitialExchangeRate);
25     }
26   }
27
28   public double Convert(Currency from, Currency to, double value)
29   {
30     double fromExchangeRate = this[from];
31     double toExchangeRate = this[to];
32     return value * fromExchangeRate / toExchangeRate;
33   }
34
35   public static ExchangeRateHolder GetInstance()
36   {
```

```
37      if (INSTANCE == null)
38      {
39        INSTANCE = new ExchangeRateHolder();
40      }
41      return INSTANCE;
42    }
43  }
```

7.3. kód. Árfolyam

7.1.4. Bank

A *bankot* reprezentáló osztály felelőssége az egyedi számlák kezelése és a pénzátutalási megbízások kezelése (lásd. 7.4. kódrészlet). *Add()* metódusa (10. sor) elfedi a számlák létrehozását, delegálva a létrehozást az *Account* konstruktorának meghívásával. A szimuláció végett lényeges átutalásokat a *Transfer()* metódus valósítja meg. A paraméterben kapott egyedi számlaszámok alapján előkeressük az érintett számlákat (**private** *Find()* metódus, 27. sor), majd ha a forrásszámlán van elegendő összeg, a terhelést (23. sor), illetve a jóváírást (24. sor) elvégezzük. Az utóbbi programsor az, ahol szükségünk van a pontos váltószámra a két devizanem között. Az *ExchangeRateHolder.GetInstance()* hívás visszaadja számunkra azt az egyke példányt, melyen a *Convert()* példányszintű metódus rögtön végre is hajtható.

```
1  public class Bank
2  {
3    private List<Account> accounts;
4
5    public Bank()
6    {
7      this.accounts = new List<Account>();
8    }
9
10   public void Add(String number, Currency currency, double value)
11   {
12     this.accounts.Add(new Account(number, currency, value));
13   }
14
15   public void Transfer(String fromNumber, String toNumber, double value)
16   {
17     Account fromAccount = this.Find(fromNumber);
18     Account toAccount = this.Find(toNumber);
19     if (fromAccount.Value < value)
20     {
21       throw new NotEnoughMoneyException(fromAccount, value);
22     }
23     fromAccount.Transfer(-1 * value);
24     toAccount.Transfer(ExchangeRateHolder.GetInstance().Convert(fromAccount.Currency,
           toAccount.Currency, value));
25   }
26
27   private Account Find(String number)
28   {
29     Account result = null;
30     foreach (Account account in this.accounts)
31     {
32       if (account.Number.Equals(number))
33       {
34         result = account;
35         break;
36       }
37     }
38     if (result == null)
39     {
40       throw new UnknownAccountExceotion(number);
41     }
42     return result;
```

```
43    }
44
45    public override string ToString()
46    {
47      StringBuilder builder = new StringBuilder(100);
48      builder.AppendLine("---< Bank >---");
49      foreach (Account account in this.accounts)
50      {
51        builder.AppendLine(account.ToString());
52      }
53      return builder.ToString();
54    }
55  }
```

7.4. kód. Bank

7.1.5. Konkurencia kezelés

A bemutatott alkalmazásban a *singleton* példány lekérése során nincs *null* vizsgálat ellenőrizve. Egy szálon futó alkalmazás esetén erre nincs is szükség, hiszen biztosított az, hogy az első *GetInstance()* hívás inicializálja a példányt. Több szál esetén azonban ez korántsem hibabiztos megoldás. Az volna a minimum hogy szinkronizáljuk a *GetInstance()* metódust [4], azonban ez lenne a *bottleneck*-je az alkalmazásunknak, hiszen minden pénzátutalási megbízás (és egyéb árfolyamokat igénylő művelet) e lépés miatt "sorban állna" az *ExchangeRateHolder lock*-jáért. Ennek kiküszöbölésére a szinkronizációt a feltétel vizsgálat után illik megvalósítani (C#-ban ehhez egy *monitor* objektumra szükség van), lévén így ha már létrejött a *singleton* példány, ennek lekérése blokkolás nélkül megvalósulhat. Azonban még így is felléphet az első inicializálás során egy furcsa szituáció, miszerint a hirtelen sok szálon történő "első hívás" a feltétel vizsgálat után "áll sorba". Ezt egy megismételt *null* vizsgálattal védhetjük meg (lásd. 7.5. kódrészlet). A jegyzet a többszálú alkalmazás fejlesztés részleteire nem tér ki, csupán a *singleton* többszálú környezetben való gyakori alkalmazása végett egy bekezdés erejéig kitekintettünk ebbe az irányba.

```
1  public class ExchangeRateHolder
2  {
3    private static readonly Object MONITOR = new Object();
4
5    private static ExchangeRateHolder INSTANCE;
6
7    [..]
8
9    public static ExchangeRateHolder GetInstance()
10   {
11     if (INSTANCE == null)
12     {
13       lock (MONITOR)
14       {
15         if (INSTANCE == null)
16         {
17           INSTANCE = new ExchangeRateHolder();
18         }
19       }
20     }
21     return INSTANCE;
22   }
23 }
```

7.5. kód. Nincs elég pénz a forrásszámlán

[4] Szinkronizáció során a megjelölt programrészen mindig csak egy szál dolgozhat. Ez egy költséges művelet, melyet bár könnyedén "kérhetünk" a futtatókörnyezettől, minden ilyen esetben körültekintően járjunk el.

7.1.6. Speciális esetek kezelése

Az átutalás több ponton meghiúsulhat, hiszen előfordulhat hogy nincs elegendő pénz a forrásszámlán (lásd. 7.6. kódrészlet), illetve a megadott számlaszámokat "elírták" (lásd. 7.7. kódrészlet). A célból két saját kivételt hozhatunk létre, melyekre ez úttal az jellemző, hogy argumentumként megkapott adatokból állítják össze a kivétel üzenetét (és ezzel az API dokumentált részévé tesszük a hibaüzenetet).

```
1 public class NotEnoughMoneyException : ApplicationException
2 {
3   public NotEnoughMoneyException(Account account, double value)
4     : base(account.Number + " hasn't got enough money to transfer " + value + " " +
          account.Currency + ".")
5   {
6   }
7 }
```

7.6. kód. Nincs elég pénz a forrásszámlán

```
1 public class UnknownAccountExceotion : ApplicationException
2 {
3    public UnknownAccountExceotion(String accountNumber)
4      : base("Unknown account (number: " + accountNumber + ")")
5    {
6    }
7 }
```

7.7. kód. Ismeretlen számla

7.1.7. Tesztelés

A szimuláció során egy fejben követhető esetet vezessünk végig (lásd. 7.8. kódrészlet), melyben "hirtelen" a Forint drámai emelkedésnek indul, és 200 Ft-ba kerülő Euróért rögtön két svájci frankot is vehetünk (23-25. sor). Az árfolyam változását *indexeren* keresztül valósítja meg a példa implementáció (lásd. 7.3. kódrészlet, 7. sor).

```
1  Bank bank = new Bank();
2  bank.Add("123-123", Currency.EUR, 100);
3  bank.Add("456-456", Currency.HUF, 2500);
4  bank.Add("789-789", Currency.CHF, 500);
5
6  Console.WriteLine(bank);
7
8  Console.WriteLine("Transfer 10 EUR from 123-123 to 456-456.");
9  bank.Transfer("123-123", "456-456", 10);
10
11 Console.WriteLine(bank);
12
13 Console.WriteLine("Transfer 20 CHF from 789-789 to 123-123.");
14 bank.Transfer("789-789", "123-123", 20);
15
16 Console.WriteLine(bank);
17
18 Console.WriteLine("Transfer 3000 HUF from 456-456 to 789-789.");
19 bank.Transfer("456-456", "789-789", 3000);
20
21 Console.WriteLine(bank);
22
23 ExchangeRateHolder exchangeRates = ExchangeRateHolder.GetInstance();
24 exchangeRates[Currency.EUR] = 200;
25 exchangeRates[Currency.CHF] = 100;
26
27 Console.WriteLine("Transfer 2000 HUF from 456-456 to 123-123.");
```

```
28 bank.Transfer("456-456", "123-123", 2000);
29
30 Console.WriteLine(bank);
```

7.8. kód. Szimuláció

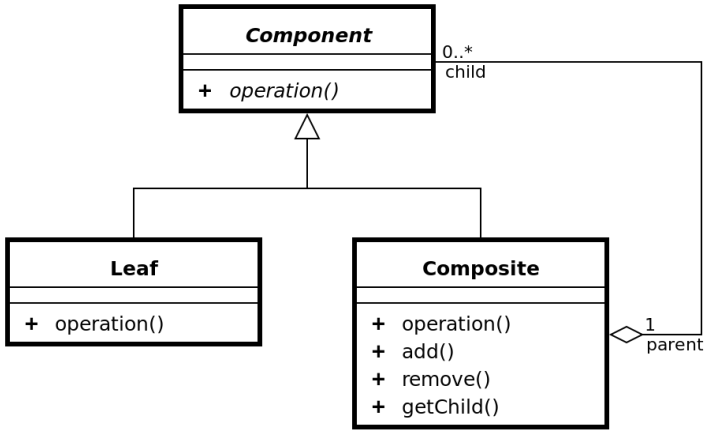
7.2. Composite design pattern

Előfeltétel

öröklés, absztrakció, generikus listák, StringBuilder, osztályrelációk

Készítsünk egy alkalmazást, mely képes formalizált módon egy XHTML dokumentum előállítására és megjelenítésére. Az elemekhez lehessen gyermek elemeket, illetve értéket rendelni. Amelyik elemnek értéke van, gyermek elemeket nem tartalmazhat. A dokumentumot akár egyetlen referencián keresztül is lehessen kezelni, azonban bármely része is egy dokumentum legyen (létezzen referenciája).

Kissé talán erőltetett a feladat, ráadásul számos könyvtár létezik a probléma megoldására, azonban kihámozható a feladatból az ún. *Composite design pattern*. Ha van egy olyan modellünk, mely egy fa adatszerkezettel leírható (léteznek elágazások, melyekből további elemek "nőnek ki", egészen addig, amíg minden ág egy levél elemben nem végződik), akkor elég nagy a valószínűsége annak, hogy ez a tervezési minta ráilleszthető a problémára. Különféle szervezeti topológiák, jól formalizált XML dokumentumok, XHTML állományok mind-mind ebbe a kategóriába sorolhatóak. Másként megközelítve az is egy árulkodó jel a tervezési minta mellett, ha az elemek egy csoportján értelmezni tudjuk ugyanazt a műveletet, amit egy egyszerű elemen is (a megfogalmazott feladatban egy XHTML elemet meg kell tudnunk jeleníteni, mindamellett hogy egy olyan elemet, mely további XHTML elemeket tartalmaz, ugyanezen megjelenítés szintén követelmény). Ez a csoporton illetve tagon megfogalmazott közös művelet is leolvasható a 7.1. ábráról, mely a megvalósítandó osztályok UML modelljét mutatja be.



7.1. ábra. Composite design pattern UML modellje[8]

A tervezési minták esetén - ha felismertük alkalmazhatóságukat - nem kérdés az, hogy hogyan tervezzük meg az osztályainkat, hiszen ezt meghatározzák számunkra. Hasonlít ez a megközelítés az ún. *best practice* elvére, mely esetén egy már bizonyított eljárást pontosan azért alkalmazunk, mert számunkra meggyőző erővel bír az, hogy akár mi, akár mások már sikeresen alkalmazták azt egy hasonló téma megoldására. Ahhoz, hogy hatékonyan és tudatosan újra tudjunk használni kódrészeket, azok alapos elméleti és gyakorlati ismeretére van szükség. Tévedés azt hinni, hogy pusztán másolással el lehet érni professzionális munkát.

7.2.1. A tervezési minta realizációja

A 7.1. ábra tanulsága szerint három osztályt kell készítenünk a probléma megoldására. Szükségünk van egy ősosztályra, mely definiálja azokat a műveleteket, melyek mind a levél, mind a csoport elemeken értelmezhetőek. A követelmények alapján ez a *GetContent()* metódus lesz, mely visszaadja azt a karakterláncot, mely az adott elem megjelenítéseként értelmezhető (ha ez egy értéket tartalmazó levél elem, akkor csupán ezt adja vissza, csoport elem esetén azonban minden gyermek elem kimenetét is szolgáltatja).

```
1  public abstract class Node
2  {
3    private readonly String name;
4
5    private String Begin
6    {
7      get { return "<" + this.name + ">"; }
8    }
9
10   private String End
11   {
12     get { return "</" + this.name + ">"; }
13   }
14
15   public Node(String name)
16   {
17     this.name = name;
18   }
19
20   public override String ToString()
21   {
22     return this.Begin + this.GetContent() + this.End;
23   }
24
25   protected abstract String GetContent();
26
27 }
```

7.9. kód. Elem

A *Node* (lásd. 7.9. kódrészlet) tipikusan **abstract** osztály, hiszen a valóságban a fában csak levelek és elágazások vannak. Az osztály azért jelenik meg a modellben, mert definiálja a közös viselkedést a valós entitások között. Magának a megjelenítésnek is lehetnek olyan részei, mely közös implementációt igényel a végpontokon. Ez esetben az *abstract* ősosztály a megjelenítésnek egy olyan tervét tartalmazza, mely részlegesen definiált. A nem definiált részeken *abstract* metódusokat hív. A *Node* osztály *ToString()* metódusa pontosan így jár el (20. sor). Az XHTML elemek nyitó- és zárótagjeinek megjelenítése azonos módon történik mind az érték, mind a csoport leszármazottak esetén, azonban a "belső" rész kimenetét csak a leszármazottakban tudjuk definiálni. Ez a technika is széles körben ismert, és - nem véletlenül - egy tervezési minta is egyben: *Abstract template design pattern*[10] a megnevezése.

```
1  public class SingleNode : Node
2  {
3    private readonly String value;
4
5    public SingleNode(String name, String value) : base(name)
6    {
7      this.value = value;
8    }
9
10   protected override String GetContent()
11   {
12     return this.value;
13   }
```

```
14 }
```

7.10. kód. Levél elem

A *SingleNode* osztály (lásd. 7.10. kódrészlet) aggregálja az elem értékét, valamint a definiált virtuális *GetContent()* alprogramot, mely a gyakorlatban ezen érték *accessor* metódusa.

```
1  public class CompositeNode : Node
2  {
3    private readonly List<Node> children;
4
5    public CompositeNode(String name)
6      : base(name)
7    {
8      this.children = new List<Node>();
9    }
10
11   public CompositeNode Append(Node child)
12   {
13     this.children.Add(child);
14     return this;
15   }
16
17   protected override String GetContent()
18   {
19     StringBuilder builder = new StringBuilder();
20     builder.AppendLine();
21     foreach (Node element in this.children)
22     {
23       builder.AppendLine(element.ToString());
24     }
25     return builder.ToString();
26   }
27 }
```

7.11. kód. Összetett elem

A *CompositeNode* osztály (lásd. 7.11. kódrészlet) felelőssége tetszőleges számú gyermek elem aggregálása, és természetesen ezen elemek karbantartása. Az *Append()* metódust (11. sor) érdemes hasonló elven elkészíteni, ahogyan a *StringBuilder* osztály *Append()* metódusa működik: visszaadja a példánymetódus saját maga példányát annak érdekében, hogy a műveletet láncban lehessen végrehajtani. Ez a módszer hasonlít a *varargs* (C# **params** kulcsszó) esetére, azonban literállal nem rendelkező típusok esetén ez ritkán segíti a kód olvashatóságát (illetve egyéb nem felügyelt veszélyeket is magában foglal).

7.2.2. Tesztelés

```
1 <html>
2 <head>
3 <title>UNI-OBUDA</title>
4 </head>
5 <body>
6 <h1>Hello World</h1>
7 </body>
8 </html>
```

7.12. kód. Program kimenete

Az elkészült osztályok tesztelésének céljából készítsük el a 7.12. blokk által leírt XHTML dokumentumot. A gyökér elemnek két gyermek eleme van (*head* és *body*), és mindegyikhez tartozik egy-egy további egyszerű gyermek elem (*title* és *h1*).

A követelményekben egy jól formalizált leírási mód volt definiálva a feladat kapcsán, ahol minden elem egy külön referenciaként igény esetén elérhető, de maga a modell egyben is kezelhető (a gyökér elem segítségével). A 7.13. kódrészlet tartalmazza az elvárt XHTML dokumentum formalizált leírását. Az 5. sorban meghívott *ToString()* metódus az *Abstract template design pattern*[10] alkalmazása során fogja meghívni azt a *GetContent()* metódust, mely a *Composite design pattern* kulcs eleme a feladatban.

```
1 CompositeNode head = new CompositeNode("head").Append(new SingleNode("title",
      "UNI-OBUDA"));
2 CompositeNode body = new CompositeNode("body").Append(new SingleNode("h1", "Hello
      World"));
3 CompositeNode html = new CompositeNode("html").Append(head).Append(body);
4
5 Console.WriteLine(html);
```

7.13. kód. Kliens kód

7.2.3. Finomhangolás

Egy dokumentum létrehozása ebben a formában kicsit "kusza". E jellemzőjének legjobb ismérve a redundáns "szóismétlések" megjelenése (*SingleNode* és *CompositeNode*). Az objektum-orientált programozás e redundanciát is könnyedén eliminálni tudja, ha létrehozzuk a konkrét XHTML elemek modellezését is. Ez azzal a kézenfekvő előnnyel is jár, hogy ha egy dokumentum pl. több *h1* elemet tartalmaz, az elem tag nevének *String* literálja ("h1") nem fog redundánsan megjelenni a kódban.

```
1 public class Html : CompositeNode
2 {
3   public Html() : base("html") { }
4 }
```

7.14. kód. Html

```
1 public class Head : CompositeNode
2 {
3   public Head() : base("head") { }
4 }
```

7.15. kód. Head

```
1 public class Body : CompositeNode
2 {
3   public Body() : base("body") { }
4 }
```

7.16. kód. Body

```
1 public class Title : SingleNode
2 {
3   public Title(String value) : base("title", value) { }
4 }
```

7.17. kód. Címsor

```
1 public class H1 : SingleNode
2 {
3   public H1(String value) : base("h1", value) { }
4 }
```

7.18. kód. Header

Az elkészítendő leszármazott osztályok (lásd. 7.14-7.18. kódrészletek) felelőssége a gyakorlatban a *tag* nevének definiálására szorítkozik, azonban az alkalmazás továbbfejlesztése során könnyen elképzelhető, hogy pl. *tag* specifikus attribútumok definiálására is lehetőség tud adni (pl. egy *img* elemnek van *src* attribútuma, de ugyanez nem értelmezett pl. egy *div* esetén).

```
1 Head head = (Head) new Head().Append(new Title("UNI-OBUDA"));
2 Body body = (Body) new Body().Append(new H1("Hello World"));
3 Html html = (Html) new Html().Append(head).Append(body);
4
5 Console.WriteLine(html);
6
7 Html root = (Html) new Html().Append(new Head().Append(new
      Title("UNI-OBUDA"))).Append(new Body().Append(new H1("Hello World"))));
8 Console.WriteLine(root);
```

7.19. kód. Tömörebb leírás

Az új osztályok segítségével (lásd. 7.19. kódrészlet) egy kicsit tömörebb megfogalmazásban definiálni tudjuk ugyanazt az XHTML dokumentumot, mellyel korábban is teszteltük a tervezési minta működését. Akár egy sorban is létre tudjuk hozni (7. sor), de ez az esetek döntő többségében értelemszerűen nem vezet egy átlátható, tiszta kódhoz[9].

Irodalomjegyzék

[1] Dr. Sergyán Szabolcs: Algoritmusok, adatszerkezetek I. (Óbudai Egyetem, ÓE-NIK 5014) 2014

[2] Dr. Szénási Sándor: Algoritmusok, adatszerkezetek II. (Óbudai Egyetem, ÓE-NIK 5013) 2015

[3] Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language, 2nd edition (Prentice Hall), ISBN-13: 978-0131103627, 1988.03.22

[4] Douglas Adams: The Hitchhiker's Guide to the Galaxy (Pan, 2009.09.01) (magyar cím: Galaxis útikalauz stopposoknak), ISBN-13: 978-0330508537, 1979.10.12

[5] Carl Adam Petri: Communication with automata, ASIN: B00ACXW46I, 1966

[6] Iterator design pattern (Wikipedia Enciklopédia) `https://en.wikipedia.org/wiki/Iterator_pattern` (látogatás ideje: 2015.09.27.)

[7] Tom McHugh, Dave Nutting: Wizard of Wor (Midway Games 1980, Wikipedia Enciklopédia) `https://en.wikipedia.org/wiki/Wizard_of_Wor` (látogatás ideje: 2015.09.27.)

[8] Composite design pattern (Wikipedia Enciklopédia). `https://en.wikipedia.org/wiki/Composite_pattern` (látogatás ideje: 2015.08.25.)

[9] Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship (Prentice Hall) ISBN-13: 978-0132350884, 2008

[10] Abstract template design pattern (Wikipedia Enciklopédia) `https://en.wikipedia.org/wiki/Iterator_pattern` (látogatás ideje: 2015.09.27.)