A biliárd játék leírása

A játékot ketten, esetenként csapatokban – 2 x 2 -es felállásban - játszák a biliárdasztalon. A biliárd kezdetekor a golyókat a kezdőállapotnak megfelelően rakjuk fel az asztalra. A játékosok felváltva löknek, és dákójukkal csak a fehér golyóhoz érhetnek. Az első eltett golyó színe határozza meg a játékos saját színét; ezzel a színnel kell a játék további részében játszania. A gurítások célja: a játékosok úgy lökjék meg a fehér golyót, hogy az, saját színű golyójukat lyukba juttassa. Ha sikerült saját golyót eltenni, a játékos még egyszer lökhet, különben ellenfele következik. Minden lökésnél a fehér golyónak először saját golyót kell érintenie. Ha a fehér először nem saját golyót talál el, vagy nem is koccan; esetleg a játékos nem saját színű golyót rak el a lyukba (ellenfele golyóját vagy a fehéret), akkor a soronkövetkező játékos kétszer jöhet. Akinek nincs saját golyója az asztalon, annak a fekete golyót abba a lyukba kell eltennie, amely átellenesen van azzal, amibe legutolsó golyója esett. Az a győztes, akinek ez elsőként sikerül. Az a játékos, aki a fekete golyót ennél hamarabb, vagy rossz lyukba helyezi, elvesztette a játékot.

Szótár

Biliárdasztal: a golyók súrlódási együtthatójának növelése érdekében posztóval borított téglalap alakú asztal, mely peremmel - fallal - rendelkezik. 4 sarkánál illetve a két hosszabbik oldalának felezőpontjában egy-egy lyuk található.

Posztó: biliárdasztalok esetében általában zöld színű durva anyag, mely növeli a golyók és az asztal közötti súrlódási együtthatót, így a golyók egy lökés után hamarabb állnak meg.

Fal: a biliárdasztal pereme, mely a golyókkal kb. azok magasságának 70%-ánál érintkezik. Ha a golyó nekiütődik a falnak, onnan kismértékű energiaveszteséggel verődik vissza.

Lyuk: az asztalon található mélyedések a golyók végcéljai. A lyukba esett golyó nem játszik többet, kivéve a fehéret.

Golyó: 16 db formára, méretre és súlyra azonos gömbök. Ezek közül kettő speciális: a fekete elhelyezése a játék végét jelenti, míg a fehér érintkezik egyedül a dákóval. A golyók egymással és a fallal csaknem tökéletesen rugalmasan ütköznek.

Saját szín: a játékos golyójának színe. Ez attól függ milyen mintájú golyót rakott el elsőként. A játék további részében az ilyen mintázatú golyókat kell eltennie a lyukakba.

Dákó: tulajdonképpen egy bot. Ezzel löki meg a játékos a fehér golyót.

Lökés: a dákónak olyan mozgása, aminek végeredménye, hogy a golyó mozgásba lendül

Eltenni: a golyót az egyik lyukba gurítjuk

Játszma: két résztvevő játéka egészen addig, míg a fekete golyó lyukba nem kerül.

Kör: a játszma egy másik elnevezése. A biliárd játék körökre oszlik, és minden körnek van egy nyertese.

Pottyan: valamelyik golyó lyukba esik.

Kezdőállapot: a játékosok a golyókat felállítják a játékszabályban meghatározott alaphelyzetbe. Egyik kezdőpontra a fehér, a másikra a fekete golyó kerül. A fekete köré rakjuk a többi golyót háromszög alakban, melynek alapja párhuzamos az asztal rövidebbik oldalával. Ebben segítségünkre van egy háromszög alakú keret, mely a golyók pontos elhelyezését segíti elő.

Kezdőpont: két kitűntetett pozíció az asztalon. Kezdéskor a fehér az egyik, míg a fekete a másik helyre kerül. Játék közben akkor van jelentősége, ha a fehér golyót eltettük. Ekkor a golyót az egyik pontra helyezve rakhatjuk vissza a játékba.

Scratch: a fehér golyó valamelyik lyukba kerül

Játékos: a biliárdjáték aktív résztvevője

Győztes: aki az utolsó golyót és végül a feketét is a megfelelő helyre tette el

Vesztes: aki idő előtt tette el a fekete golyót, vagy rossz helyre tette el azt, illetve a fehéret is elrakta ugyanazon lökés alkalmával

Követelmény definíció

A rendszer célja, alapvető feladata:

A MagicBalls program a biliárd játék számítógépes szimulációja. Két játékos küzdhet egymás ellen. A cél a fekete golyó mielõbbi lyukba lökése. Egy játékos addig lökhet ameddig a szabályoknak megfelelően, pottyantja lyukba a hozzá tartozó (csíkos vagy teli) golyókat. (A szabályok megtalálhatóak a feladat leírásában.) A rendszer célja a játék élethű, hangulatos és élvezetes szimulációja a mindenki számára könnyen hozzáférhető személyi számítógépeken, viszonylag kicsi erőforrás igénnyel.

A fejlesztőkörnyezet:

A rendszerfejlesztés nyelve az C++ programozási nyelv. A programfejlesztés BorlandC++ 3.1 alatt folyik. A rendszernek futtathatónak kell lennie ezen fordítókon, konkrétan a HSZK laboratóriumaiban található PC-k BorlandC++ 3.1-es compilerein.

A rendszer számára szükséges környezet:

- A program működtetéséhez Intel alapú IBM kompatíbilis AT PC szükséges, i80486 illetve azzal kompatibilis Intel processzorral.
- A rendszer, lévén grafikus jellegű alkalmazás, igényel grafikus kártyát és monitort, melyek képesek legalább 640x480 felbontású 16 színű grafikus megjelenítésre, és megfelelnek a VESA szabványnak.
- A program memóriaigénye kicsiny. Mindössze az alsó memóriát használja így legfeljebb 640Kb elégséges futtatásához.
- A rendszer irányításához input periféria szükséges. A Magic Balls vezérléséhez billentyűzet kell.
- A program működéséhez némi háttértároló terület is elengedhetetlen, de nem tart igényt sok helyre. A tároláshoz legfeljebb néhány MB elégséges.

A felhasználói felület:

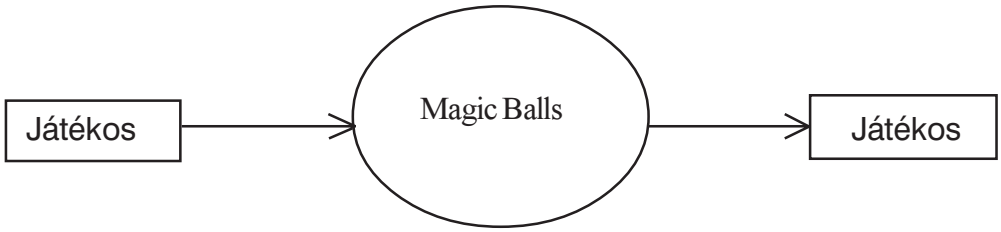
A Magic Balls grafikus felhasználói felülettel működik. A játékosok a képernyőn láthatják az asztalt, a rajta guruló golyókkal, illetve nyomon követhetik a kijelzőn a játék aktuális állapotát.

A játékosok csak akkor adhatnak inputot a gépnek, ha a golyók mozgása már megállt. Az inputok a következők lehetnek: a játékosok a játszma kezdetén megadják a nevüket, játszma közben pedig az aktuális ütés paramétereit. Az ütés paramétereit úgy adhatják meg, hogy a billentyűzeten található 6 ($\rightarrow$), 4 ($\leftarrow$) gombokkal beállítják az ütés szögét, a 8 ($\uparrow$), 2 ($\downarrow$) gombokkal pedig az ütés erejét. Ezt követően a beállítások érvényességét az ENTER gomb leütésével jelzik.

A játékból a felhasználó az ESC gomb leütésével léphet ki.
A játék aktuális állását a képernyő alján található kijelző állandóan mutatja, segítve ezzel a játékosokat a játék menetében való eligazodásban.

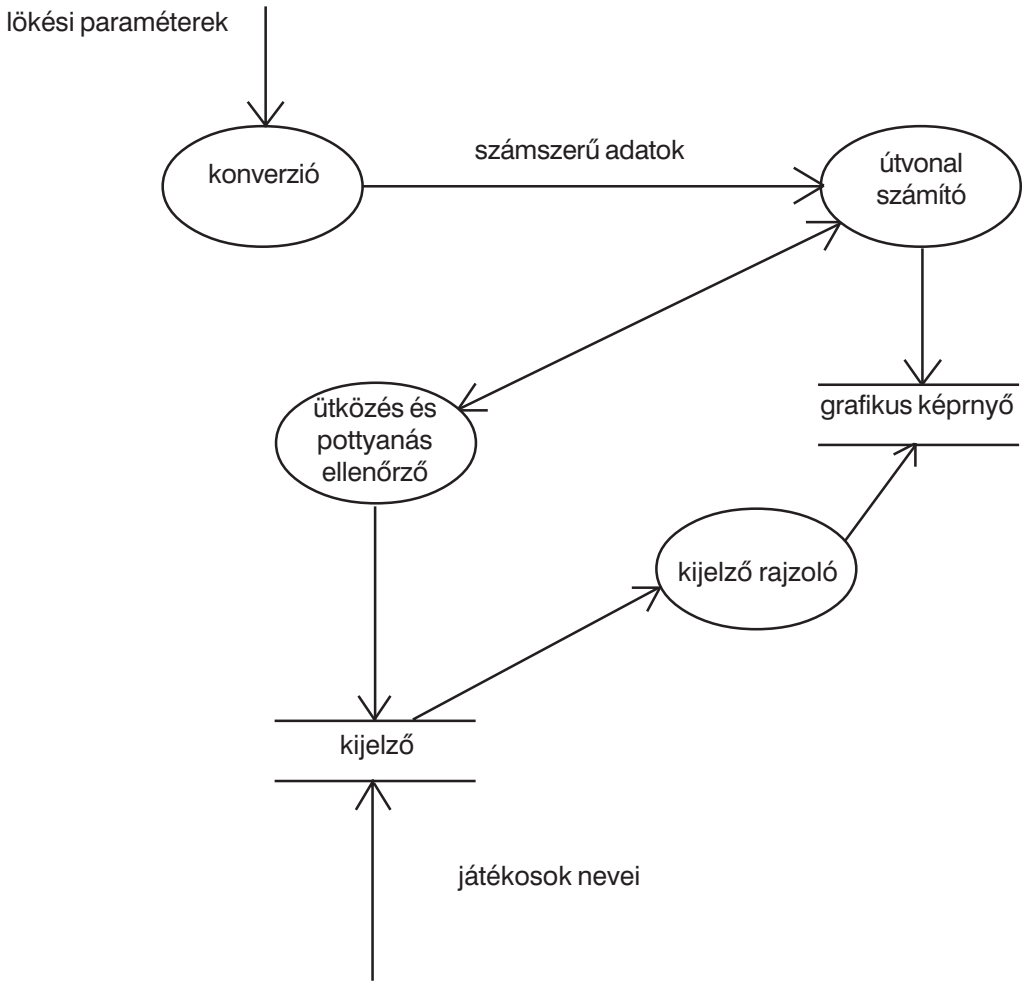
Funkcionális specifikáció

Context diagram:



0. szintű felbontás:

A rendszerbe belépő adatok, és a rajtuk végzett konverzió.



Minőségi faktorok

Teljesítmény:

A programnak a rendszerkörnyezetben definiált géptípuson olyan sebességgel kell futnia, hogy az animáció folyamatos és az ütés erejének megfelelően gyors, valósidejű legyen.

Újrafelhasználhatóság:

A program nagymértékben újrahasznosítható ugyanis a grafikus felület teljes mértékben le van választva a szimulációs magtól, így tetszőleges grafikai felület illeszthetőa rendszerhez.

Rugalmasság:

Az előző pontban említett funkcionális szétválasztásnak köszönhetően a rendszer könnyedén portolható más környezetbe, mely alkalmas C++ állományok fordítására mivel egy esetleges platformváltás esetén a kódon végzett esetleges minimális módosítás mellet elég a grafikus felületet lecserélni.

Felhasználhatóság:

A program használata könnyű, nem igényel semmilyen konkrétan a rendszerhez kapcsolódó külön tudást. A játékos egyszerűen a leírást követve, mindössze a néhány funkcióbillentyűt használva élvezheti a játékot.

Minősítő követelmények

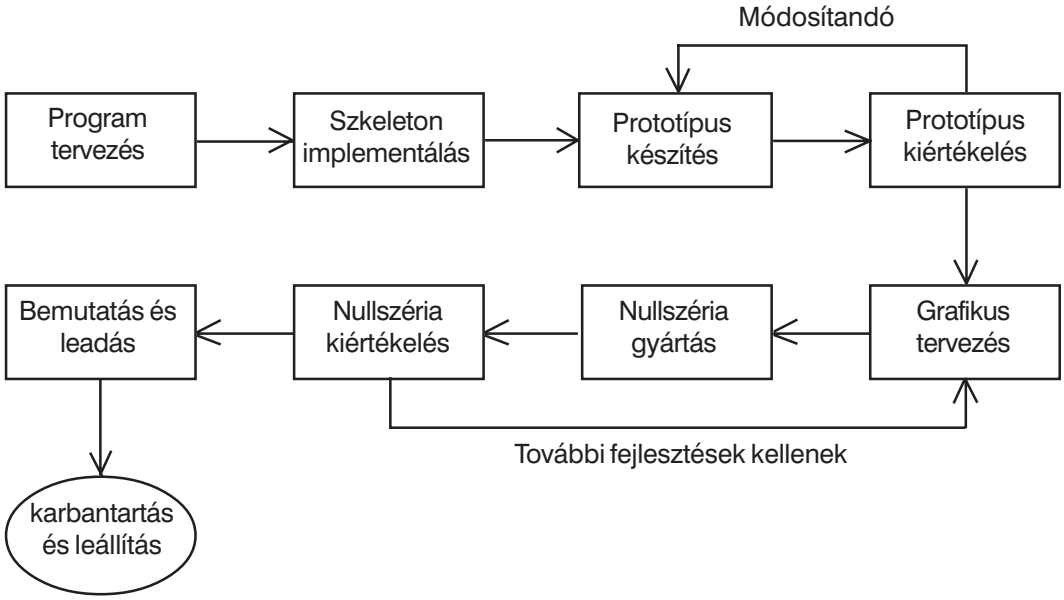
A rendszernek meg kell felelnie az ebben a dokumentumban leírt követelményeknek. A megfelelés ellenőrizhető egyszerűen a játék játszásával.

A kibocsátás módja

A program forráskódját kell e-mail-ben elküldeni tömörített formában.

Projekt terv

1. Életciklus Modell



Mivel a termék egyedi, egyszerűbb, mint a szériában gyártott társai; életciklusa rövidebb. Az életciklus követi a projekt definícióban leírtakat: a három lépcsős tagozódást. A termék életének rövidsége miatt (1 félév) a karbantartás nem játszik fontos szerepet az életciklusban, de kérésre elvégezhető. A leadás után a projekt magára marad, a fejlesztése és programozása leáll. Esetleges továbbfejlesztésről a team gondoskodik, ha látja annak szükségességét.

2. Szervezési struktúra

A csapat egyetemleges felelősséget vállal az elkészített munkáért. A felállás vezetett centralizált formán alapszik. Farkas a vezető és dokumentátor, Vásárhelyi a vezető programozó, Sója programozó és tesztelő, Erdei programozó. Mivel nincs elég ember a feladatok éles elkülönítésére, ezért mindegyikünk dokumentál, programoz és tesztel. A programozási rész nagyrészt Vásárhelyin alapszik, ő az, aki meghatározza a csapat irányvonalát.

3. Személyi és erőforráskövetelmény

A csapat minden tagja bitokában van a C++-ban való programozás képességének. Megszerezték a projekt tervezéséhez szükséges ismereteket, és készen állnak a kidolgozásra. Mindegyikünk rendelkezik a fejlesztéshez szükséges személyi számítógéppel, amelyen a kódolás folyik. Mivel nap, mint nap találkozik a csapat, ezért a felmerülő problémákat meg tudjuk beszélni személyesen, de a dokumentumokat e-mailen keresztül küldjük el egymásnak.

4. Fejlesztési ütmterv

A projekt tervezése három fő lépcsőre oszlik: szkeleton – prototípus – grafikus. A fejlesztési ütemterv betartja a *projekt definícióban* leírtakat.

Szkeleton:	1. hét (02.18.)	Brain Storming
	2. hét (02.25.)	Követelmény leírás elkészítése
	3. hét (03.03.)	Követelmény, projekt, funkcionalitás
	4. hét (03.10.)	Analízis modell kidolgozása
	5. hét (03.17.)	Szkeleton tervezése
	6. hét (03.24.)	Szkeleton beadása
Prototípus:	7. hét (03.31.)	Prototípus koncepciója
	8. hét (04.07.)	Részletes tervek elkészítése
	9. hét (04.14.)	Prototípus elkészítése, tesztelése
	10. hét (04.21.)	Prototípus beadása
Grafikus felület:	11. hét (04.28.)	Grafikus felület specifikálása
	12. hét (05.05.)	Grafikus változat készítése
	13. hét (05.12.)	Grafikus beadása

5. Költségek becslése

Felhasználva mind a múlt félévben szerzett tudásunkat, mind programozási ismereteinket, a csapat tagjai és a számítások egybehangzó megoldást adtak. A fejlesztésre szánt idõ 3 hónap. *COCOMO – modell* segítségével a program sorainak száma 1300 – 1400 közé tehetõ. Ezen számításomat, a vezető programozó is megerősítette, aki top – down módszerrel közelítette az értéket.

6. Fejlesztési program ellenőrzése és irányítása

A beérkező dokumentumok iktatása Farkas feladata. A koordinációt és irányvonalat Vásárhelyi határozza meg, hiszen ő a vezető programozó. Mindenki elvégzi a rá kiosztott feladatot, és az lekódolt programrészletet dokumentációval együtt elküldi a dokumentátornak, aki iktatja a hetente leadandó naplóba.

7. Eszközök és módszerek

A csapat tagjai a Borland cég C++ 3.1 rendszere alatt fejlesztik a programot. A dokumentáció elkészítése Adobe Page Maker kiadványszerkesztővel, az UML szabvány szerinti statikus struktúra diagramok, use case diagramok, szekvencia és statechart diagramok Rational Rose szoftverrel történnek. A dokumentáció elkészítése a RUP módszertan alapján történik, mely tartalmazza a szükséges UML részeket is.

8. Programozási nyelvek

A kikötés szerint: a fejlesztés C++-ban történjen. Az elkészült programnak Borland C++ alatt lefordíthatónak és futtathatónak kell lenni.

9. Tesztelési körülmények

A kis létszámra való tekintettel az egyes programrészleteket körforgó módszerrel fogjuk tesztelni. Mindenki átadja a programrészletét egyik társának tesztelés céljára; erről értékelést készít és visszajuttatja a készítőjéhez, aki a talált hibákat kijavítja. A módosított változatot most másik csoporttársának adja át, aki szintén elvégzi a szükséges vizsgálatokat.

10. Igényelt dokumentációk

A forráskódok mellett meg kell adni szükség szerint valamennyi olyan installálási útmutatást és kezelési leírást, amelynek alkalmazásával a tárgykód elõállítható és futtatható. A program fejlesztésével egyidőben készül majd a felhasználói kézikönyv, és a forráskód leírása. A verifikációs terv elkészülte a szkeleton beadási határidejéhez kötehető. Minden egyes lépcső tetején akceptancia tesztnek vetjük alá az addig elkészült programrészletet és ezekről értékelő jelentést készítünk, melyet a heti dokumentációs csomagban meg lehet találni.

11. Bemutatás és átadás módja

A bemutató a HSZK-s gépek valamelyikén az RIVP teremben történik. A gépre telepített állományok helyes működését a konzulens vizsgálja majd meg. Ha az értékelése megfelelő, akkor az átadás megtörtént. De a forrásprogramot és az összes dokumentációt még az *eniac*ra is el kell küldeni. A bemutató várható időpontja a szorgalmi időszak utolsó hete.

12. Betanítás – anyagok és ütemezés

A betanításra bemutatáskor kerül sor. Itt a csapat egyik tagja elmondja a konzulensnek, hogy pontosan hogyan működik a program.

13. Installációs terv

A merev lemezre való felrakás után, az otthon lefordított és futtatható file-t megpróbáljuk elindítani a HSZK-s gépen. Esetleges hibaüzeneteknél a fordítást ott helyben végezzük a teljes csapattal, hogy kiküszöböljük a felmerülő problémákat.

14. Karbantartási elvek

Az elkészített program igény szerint karbantartható. Ha bármilyen javítás szükséges lenne, kérjük keressen bennünket a megadott e-mail címeken.

15. Fizetés módja, ütemezése

Mivel csapatunk teljesen non-profit jellegű, ezért csak a félév végi aláírás és osztályzat ösztönöz bennünket a program megírásában. A „fizetés” a félév végén történik, írásos formában, melyet a csapat minden tagja megkap. A bejegyzésnek legkésőbb a vizsgaidőszak végére (2000.06.30.) szerepelnie kell a csapattagok indexében, illetve a NEPTUN tantárgyi adatlapjukon.

16. Információ források

Internetes referenciák:

A szabályok egyértelmű tisztázásához: *http://www.billiard.com*
Az UML használatához: *http://www.rationalrose.com*
A feladat kiírásának helye*: http://eniac.iit.bme.hu/~szglab4*

Publikációk:

Kondorosi-László-Szirmay: Objektum – orientált szoftverfejlesztés
Szirmay: Számítógépes grafika

Use Case diagramok

Actor Action	System Response
1. A játékos meglöki a fehér golyót a dákóval	2. A fehér golyó kezdősebességgel nagyság és irány) rendelkezik 3. A fehér golyó más golyókkal ütközik. 4. Miután minden golyó megállt, a rendszer elemzi a helyzetet, és eldönti, hogy melyik játékos következik.

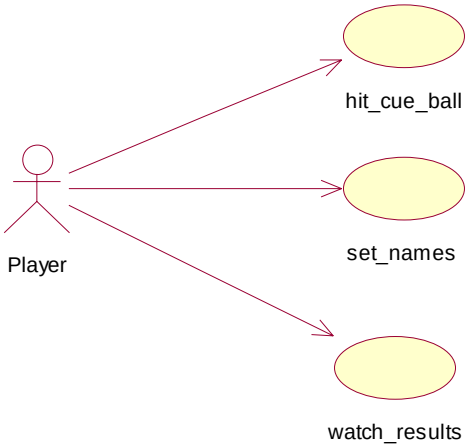
Use Case: hit_cue_ball
Actor: Player
Purpose: A játék egy lépésének elvégzése
Overview: A játékos meglöki e fehér golyót, amely több másik golyóval illetve fallal való ütközés után megáll. Az ütközés során minden, az ütközésben résztvevő golyó kezdősebességgel rendelkezik - tehát elmozdul - és esetleg más goluókkal és falakkal való ütközés után megáll vagy beleesik egy lyukba.

Essential Use Case és Use Diagram

Actor Action	System Response
1. A játékos beírja az adatait a billenytűzeten keresztül	2. A rendszer a beírt adatokat megjeleníti a kijelzőn.
3. A játékos megnézi a kijelzőn, hogy mit írt be, és ha szükséges, korrigálja azokat.	

Actor Action	System Response
1. A játékos elvégzi a játék egy lépését (Hit the CueBall), majd megszemléli a lökés eredményeit.	2. A rendszer elvégzi a játékos által kiadott “mozdulatsort”, ellenőrzi a golyókat, majd megjeleníti a kijelzőn az eredményeket: melyik játékos lökött, milyen golyók pottyantak, ki jön, stb.
3. A játékos megnézi a kijelzőt, és a saját maga által generált eredményeket felülbírálja a kijelző eredményeivel.	

Use Case Diagram



Use Case: watch_results
Actor: Player
Purpose: A játék pillanatnyi állásának és eredményeinek grafikus megjelenítése.
Overview: A játékos a képrnyőre nézve ellenőrizni tudja az általa megállapított eredményeket.

Use Case: hit_cue_ball
Actor: Player
Purpose: A játék egy lépésének elvégzése
Overview: A játékos meglöki e fehér golyót, amely több másik golyóval illetve fallal való ütközés után megáll. Az ütközés során minden, az ütközésben résztvevő golyó kezdősebességgel rendelkezik - tehát elmozdul - és esetleg más goluókkal és falakkal való ütközés után megáll vagy beleesik egy lyukba.

Use Case: set_names
Actor: Player
Purpose: A játékos adatainak beállítása
Overview: A játékos a billentyűzeten beírja a nevét, mely azonnal megjelenik a kijelzőn. Így a játékos korrigálhatja az esetleges hibákat. Miután a játékos véglegesítette az adatokat, azokon a játék folyamán többet nem változtathat.

Szótár

- **Biliárd játék:** a játék pontos szabályai megtekinthetőek a feladat definiálásában
- **szimuláció:** tetszőleges, az életben lejátszódó folyamat számítógépen történő utánzása, rekonstrukciója a számunkra lényegtelen elemek elhagyásával
- **pottyant:** ezen esemény definíciója megtalálható a feladat definíciójának szótárában
- **személyi számítógép:** Personal Computer, a ma leginkább elterjedt géptípus. Lényege, hogy a gép minden erőforrásait egyetlen felhasználó veszi igénybe.
- **erőforrások:** a számítógép részei, melyeknek a rendszerhez való hozzávétele elvétele illetve a rendszerben lévő mennyiségenagy hatással van a számítógép teljesítményére vagy egy adott funkciót ellátó képességére (pl. memória, nyomtató…)
- **fejlesztőkörnyezet:** a rendszerfejlesztő munka környezete. Azon körülmények definiálása, melyek meghatározóak a fejlesztés szempontjából és amelyeket ezért a rendszer fejlesztőinek egyeztetni kell (pl. a használt programozási nyelv)
- **rendszerfejlesztés:** azon tevékenység, mely a rendszer létrehozását szolgálja
- **programozási nyelv:** egy feladat megoldását elősegítő formális leírónyelv, mely közvetlenül fordítható a számítógép által végrehajtható leírássá.
- **fordító, compiler:** eszköz, mely a programozási nyelv gép által érthető nyelvvé való fordítását végzi.
- **környezet:** a programot körülvevő, a program által érzékelhető dolgok
- **PC:** Personal Computer, a személyi számítógép
- **processzor:** a számítógép központi feldolgozó egysége. Sebessége nagymértékben befolyásolja a számítógép működési sebességét
- **alkalmazás:** a számítógépet felhasználók különböző igényeit kielégítő, különböző feladatokat megoldó programok.
- **grafikus kártya:** a számítógép monitorát vezérlő nyomtatott áramköri lapka
- **monitor:** a számítógép információ-megjelenítésre használt eszköze. Egy TV-hez hasonló, képi, vagy szöveges megjelenítő
- **felbontás:** a monitor képi megjelenítésének finomsága. Általában két szám szorzatával jellemzik. A két szám a számítógép monitorán vízszintesen, és függőlegesen elhelyezkedő képpontok száma.
- **képpont:** a monitor elemi megjelenítési egysége
- **VESA szabvány:** a monitorok működését szabályozó, a megjelenítést egységesítő szabvány.
- **memóriaigény:** a felhasznált memória mennyisége.
- **alsó memória:** a DOS operációs rendszerben a felhasználó számára elérhető memória

- **input periféria:** olyan eszköz, mely a számítógépbe való információ-bevitelt szolgálja (pl. billentyűzet).
- **háttértároló:** merevlemez, nagy kapacitású tároló, mely a számítógép kikapcsolása után is megőrzi a tárolt információt.
- **felhasználói felület:** a rendszernek a felhasználóval érintkező felülete, kommunikációs csatornája.
- **input:** információ bevitel

Objektum katalógus

Main_program osztály

Alaposztály(ok):

(nincsenek)

Példány(ok):

1 példány a fõprogramban

Komponens(ek):

Game osztály

Változó(k):

`the_game`	Game	a konkrét játék (meccs)
`players`	2 karaktersorozat	a játékosok nevei
`background_color`	egész szám	a háttérszín
`button_color`	egész szám	a menü gombjainak a színe
`button_letter_color`	egész szám	a menü feliratainak a színe

Felelősség(ek):

```
void        start_up ()
void        menu ()
void        set_players ()
void        shut_down ()
```

Ez lényegében a keret. Minden ez alatt van, innen indul ki minden…

Game osztály

Alaposztály(ok):

(nincsenek)

Példány(ok):

1 példány a 'main_program' objektumban

Komponens(ek):

Billiardtable osztály, Scoreboard osztály

Változó(k):

`the_billiardtable`	Billiardtable	az asztal
`the_scoreboard`	Scoreboard	a kijelzõ

Felelősség(ek):

```
void        play(char*, char*)
```

Az itt szereplõ 'play' metódus bizonyos tekintetben a dolgok lelke. "Ő" a fő-koordinátor.

Scoreboard osztály

Alaposztály(ok):

(nincsenek)

Példány(ok):

1 példány a 'the_game' objektumban

Komponens(ek):

Vectors osztály

Változó(k):

shape	2 Vectors	a kijelző alakja
place_of_names	2 Vectors	a nevek helye a kijelzőn
place_of_ballnumbers	2 Vectors	a golyószámok helye a kijelzőn
place_of_balltypes	2 Vectors	a golyótípusok helye a kijelzőn
place_of_turnnumber	Vectors	a lökésszám helye a kijelzőn
background_color	egész szám	a háttér színe
letters_size	egész szám	a betüméret
letter_color	egész szám	a betüszín
highlighted_letter_color	egész szám	a kiemelt szöveg betüszíne
red_out_letter_color	egész szám	a "hibajelzett" szöveg betüszíne
names	2 karaktersorozat	a játékosok nevei
ball_numbers	2 egész szám	a játékosok fent levõ golyóinak száma
ball_types	2 egész szám	a játékosok célgolyóinak típusa
current_player	egész szám	a soron következő játékos
turn_number	egész szám	a hátra levő lökések száma
last_holes	2 egész szám	az a lyuk, ahova a feketét kéne belökni

Felelősség(ek):

```
void        set_ball (int, int)
void        set_data (int, int)
void        set_next_player (int, int)
void        set_names (char*, char*)
char *      get_name (int)
int         get_current_player ()
int         get_data_for_game (int)
dfg_struc   get_data_for_draw ()
void        draw()
void        nit ()
void        set_last_hole (int, int)
int         get_last_hole (int)
```

A 'Scoreboard' osztály lényegében magában foglalja a játékosokat is kikerülve ezzel az esetleges kommunikációs zavarokat (a kijelző és a játékosok között), ami komoly megvalósítási nehézségeket jelentene. Lényegében mindkét játékos minden adata itt foglal helyet. Az adatokat (nevek, lökésszám, stb.) a 'the_game' objektum "kezeli" (tartja karban). Alkalmasint érdemes az egyes játékosokhoz tartozó bizonyos adatokat egyedileg, struktúrákban tárolni a könnyebb kezelhetőség érdekében (file-ba írás). Az 'init' metódus a kijelző logikai koordinátáit és egyéb alapadatait állítja be (közvetlenül semmi köze nincs a grafikához).

Billiardtable osztály

Alaposztály(ok):
(nincsenek)
Példány(ok):
1 példány a 'the_game' objektumban
Komponens(ek):
Element osztály, Vectors osztály
Változó(k):

shape	2 Vectors	az asztal alakja
color	egész szám	az asztal színe
starting_points	2 Vectors	a két kezdõ pont az asztalon
elements_on_table	16 pointer	az asztalon lévő elemek
state_of_balls	15 egész szám	a golyók állapota az utolsó lökés eredményeként
where_to_put	egész szám	ahová a fekete golyót el kell tenni

Felelősség(ek):

```
void        init
void        start_up
void        place_cue_ball (int)
int         hit_cue_ball (int)
int         examine ()
int         motions ()
void        draw ()
int         where_to ()
int         last ()
```

Néhány "szabály" van, amit mindenféleképp be kell tartanunk. Az "elemek" sorrendje nagyrészt meghatározott (golyók, lyukak, falak), mert ilyen sorrendben érdemes vizsgálni az ütközéseket is (elõbb golyóval, azután lyukkal, majd fallal). Továbbá tudnunk kell, hogy melyik az utolsó golyó a sorban. Itt jön be a két irányú változat elõnye – ha ez a golyó pottyan, az utódját egy lépésben meg lehet találni… A 'state_of_balls' az egyes golyók utolsó lökés utáni helyzetét mutatja kellõen átgondolt és kidolgozott egész számú kódolásban.

A 'hit_cue_ball' lényegében a fehér golyó 'hit' metódusának és a 'motions' metódusnak ilyen sorrendbéli meghívásából áll. Visszatérési értékben jelzi a történteket. Ezután meghívva az 'examine' metódust, mivel is teljes képet lehet kapni a lökés okozta történésekrõl.

Az inicializálás az asztal logikai koordinátáinak és egyéb alapbeállításainak megadásáért felelõs.

Element osztály

Alaposztály(ok):
(nincsenek)
Példány(ok):
0 példány (alaposztály a falak, a lyukak és a golyók egytípusú kezeléséhez)
Komponens(ek):
Vectors osztály
Változó(k):

`position`	Vectors	az elem pozíciója
`color`	egész szám	az elem színe
`crash_coefficient`	valós szám	az elem ütközési együtthatója

Felelősség(ek):

```
virtual voidinit()
virtual voiddraw ()
virtual int crash (Element*)
void        set_color (int)
int         get_color ()
void        set_position (Vectors)
Vectors     get_position()
double      get_crash_coeff()
void        set_crash_coeff (double)
```

Az 'Element' csupán a fal, lyuk és golyó típusú objektumok "egyként" kezeléséért felelõs és csupán ezért jött létre alaposztályként – belõle valós objektum nem lesz majd csak névlegesen használjuk (listákban ténylegesen elõfordul, de csak és kizárólag az imént említett okból).

Golyóknál a 'color' változó tartalmazza a mintát is (sima, csíkos). Ezt oly módon tehetjük lehetõvé, hogy pl. az elsõ tizenhat színt használjuk a golyók színének meghatározására és azon egy egyértelmű és logikus leképezést kreálunk.

Wall osztály

Alaposztály(ok):
Element osztály
Példány(ok):
4 példány a 'the_billiardtable' objektumban
Komponens(ek):
Vectors osztály
Változó(k):

`shape`	2 Vectors	a fal alakja
`color`	egész szám	a fal színe
`border_shape`	2 Vectors	a fal szegélyének alakja
`border_color`	egész szám	a fal szegélyének színe
`id`	egész szám	a fal azonosítója (vízszintes ill. függőleges)

Felelősség(ek):

```
void        init ()
void        draw ()
int         crash (Element *)
void        set_position (Vectors)
```

A osztály az asztal négy falát hivatott modellezni. Működése sokban hasonlít a lyukak és a golyók működéséhez, bár sokban el is tér azokétól. Az 'init' metódus az alapbeállításokért (logikai koordináták, színek) felelõs.

Hole osztály

Alaposztály(ok):
Element osztály
Példány(ok):
6 példány a 'the_billiardtable' objektumban
Komponens(ek):
(nincsenek)
Változó(k):

`radius`	valós szám	a lyuk sugara

Felelősség(ek):

```
void        init ()
void        draw ()
int         crash (Element *)
```

Első ránézésre igen-igen sok a tisztázatlan pont, de csupán első ránézésre. Például minek ütközési együttható egy lyuk esetében. Valóban nincs érdemi feladata. Ez az extra lefoglalt memóriarész az ára a hasonszõrű objektumok hasonló kezelésének lehetővé tételéért kialakított szerkezetnek. Persze ki lehetett volna ezt kerülni, de minthogy kisméretű a szoftver ekkora felesleg belefér a memória-keretbe és semmi értelme nincs kiküszöbölni létrehozván ezzel egy meglehetősen bonyolult és nehezen átlátható osztály-rendszert. Másodszor, mit csinál az 'crash' metódus. Ha a golyó pottyant, akkor ezt a visszatérési értékén keresztül jelzi neki. Az 'init' metódus itt is az alapbeállításokért (logikai koordináták és méretek) felelős.

Ball osztály

Alaposztály(ok):
Element osztály
Példány(ok):
15 példány a 'the_billiardtable' objektumban
Komponens(ek):
Vectors osztály
Változó(k):

`radius`	valós szám	a golyó sugara
`velocity`	Vectors	a golyó sebessége

Felelősség(ek):

```
void        init ()
void        set_velocity (Vectors)
Vectors     get_velocity ()
void        move ()
void        draw ()
int         crash (Element *)
double      get_radius ()
```

A 'move' metódus biztosítja, hogy a golyók valóban mozognak. Mozgás közben szépen sorban meghívják a többi elem 'crash' metódusát az esetleges "egymásra hatások" szimulálása érdekében. Az inicializálás csakúgy, mint másutt, a logikai koordináták és méretek, valamint a színek beállításáért felelős.

Cue_ball osztály

Alaposztály(ok):
Element osztály, Ball osztály
Példány(ok):
1 példány a 'the_billiardtable' objektumban
Komponens(ek):
(nincsenek)
Változó(k):
(nincsenek)
Felelősség(ek):

```
hit (int)
```

Az egyetlen új felelősség, ami megjelent a lökés. Lényegében ez az egyetlen, ami funkcionális különbséget jelent a fehér golyó és társai között. Lényegében a 'hit' metódus végzi el mindazt, amit az osztály-szerkezetbe alkalmasint nehézkesen és feleslegesen beillesztett dákó objektumnak kellett volna véghez vinnie. Ez sokat egyszerűsít a képen és gyorsabbá teszi a programot.

Vectors osztály

Alaposztály(ok):
(nincsenek)
Példány(ok):

Elem / elemtípus	db.		elem db.		összesen
'the_scorecoard'	11	*	1	=	11
'the_billiardtable'	4	*	1	=	4
'Element' származékok	1	*	26	=	26
'Wall' típusú elemek	6	*	4	=	24
'Ball' típusú elemek	1	*	16	=	16
ÖSSZESEN					76

Komponens(ek):
(nincsenek)
Változó(k):

i, j	2 egész szám	a vektor összetevői

Felelősség(ek):

```
void        set(Vectors)
void        set(double, double)
double      length()
double      get_i()
double      get_j()
Vectors     operator+(Vectors)
Vectors     operator-(Vectors)
double      operator*(Vectors)
void        operator=(Vectors)
Vectors     operator*(double
Vectors     operator/(double)
```

A felelősségek halmaza még korántsem végleges, mivel majd csak a konkrét algoritmus kidolgozásánál dől majd el, hogy pontosan milyen vektor-műveletekre lesz majd szükség.

Gfx osztály

Alaposztály(ok):
(nincsenek)
Példány(ok):
1 példány globális változóként
Komponens(ek):
(nincsenek)
Változó(k):

screen_width	egész szám	a képernyő szélessége
screen_height	egész szám	a képernyő magassága
color_depth	egész szám	a grafikus mód színmélysége
number_of_pages	egész szám	a grafikus lapok száma
text_type_array	egész szám tömb	a használt betűk típusai
text_size_array	egész szám tömb	a használt betűk mérete
active_page	egész szám	az aktuális grafikus lap
saved_bitmap	pointer	az elmentett grafikus terület
rate_of_billiardtable	valós szám	a biliárdasztal rajzolási együtthatója
rate_of_scoreboard	valós szám	a kijelző rajzolási együtthatója
length_of_cue	valós szám	a dákó hossza

Felelősség(ek):

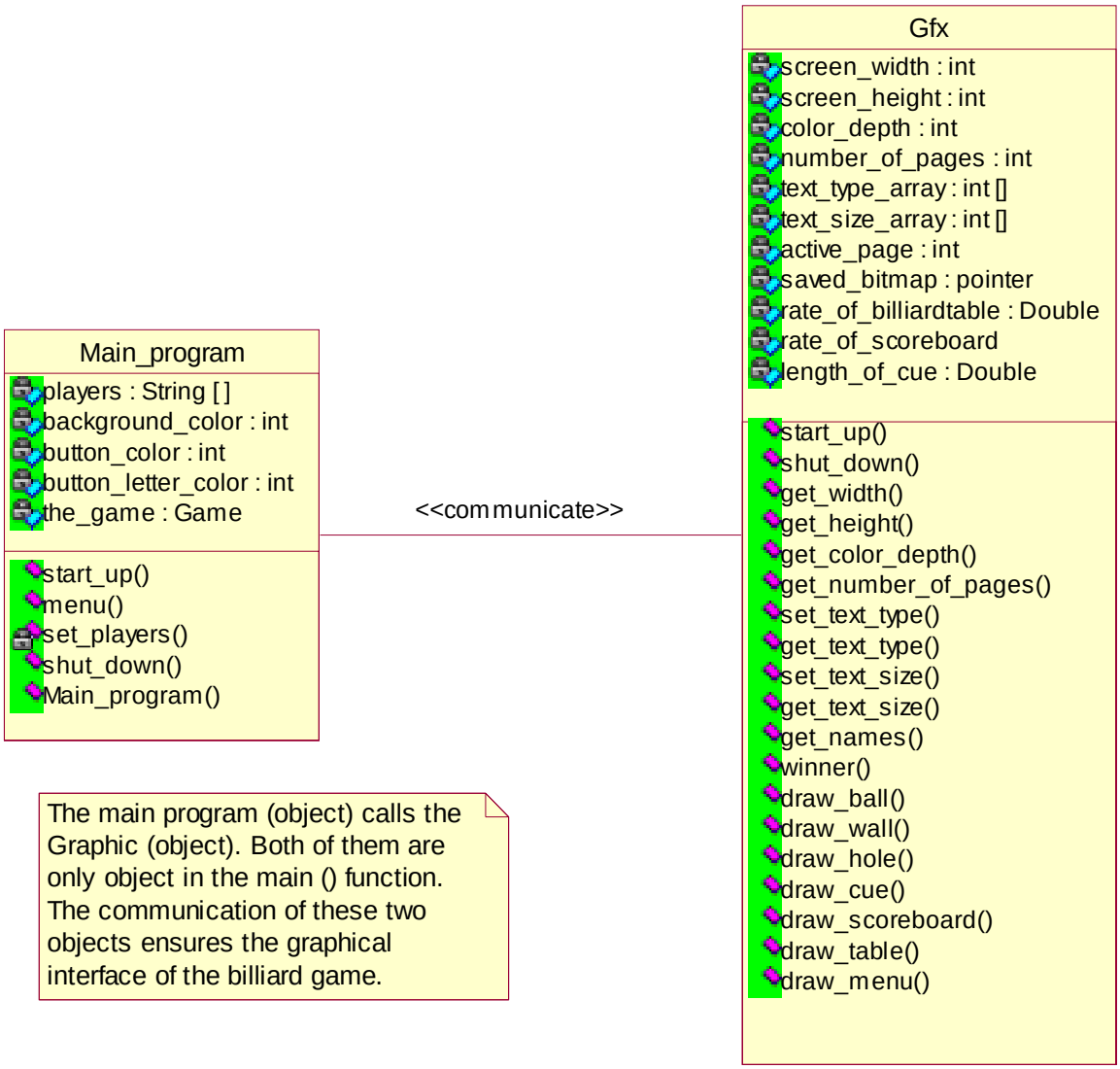
```
void        start_up()
void        shut_down()
int         get_width()
int         get_height()
int         get_color_depth()
int         get_number_of_pages()
void        swap_active_page()
void        set_text_type(int, int)
int         get_text_type(int)
void        set_text_size(int, int)
int         get_text_size(int)
void        get_names(char *, char *, int)
void        winner(char *)
void        draw_ball(Ball *)
void        draw_wall(Wall *)
void        draw_hole(Hole *)
void        draw_cue(double, double, double, double)
void        draw_scoreboard(Scoreboard *)
void        draw_table(Billiardtable *)
void        draw_menu(char *, char *, char *, int)
```

Az osztály elsődleges célja a grafika különválasztása a program egyéb részeitől, vagy is az, hogy a grafikus osztály kicserélésével egyszerűen kezelni lehessen a grafikus módváltásokat… Ez különösen leegyszerűsíti a proto és a végleges grafikus változat közötti átmenetet.

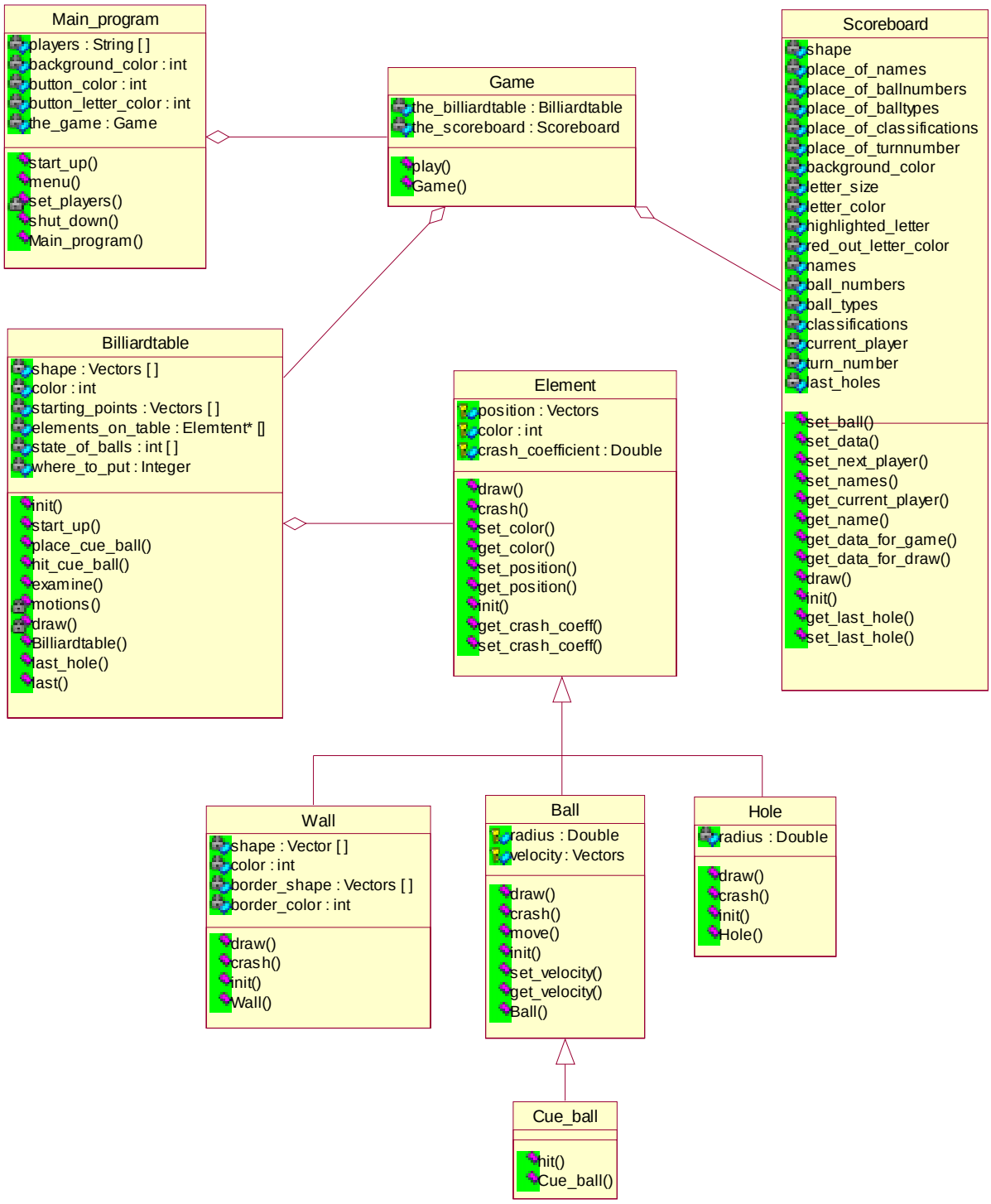
A tárolt alapadatok között van jó néhány, amit nem kellene feltétlenül tárolni, mert jelenleg csak BGI grafikával lesz dolgunk. Ennek ellenére jobb, ha mégis így teszünk az esetleges továbbfejlesztések megkönnyítése érdekében .

Osztálydiagramok

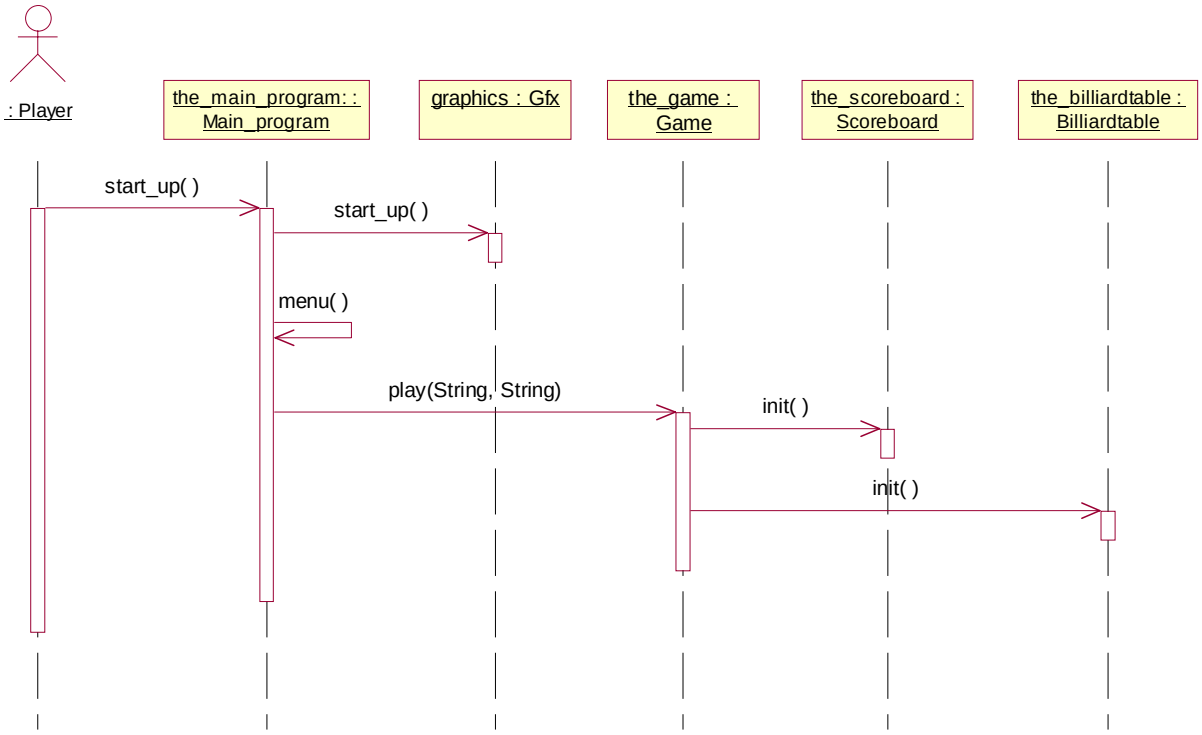
Átfogó diagram



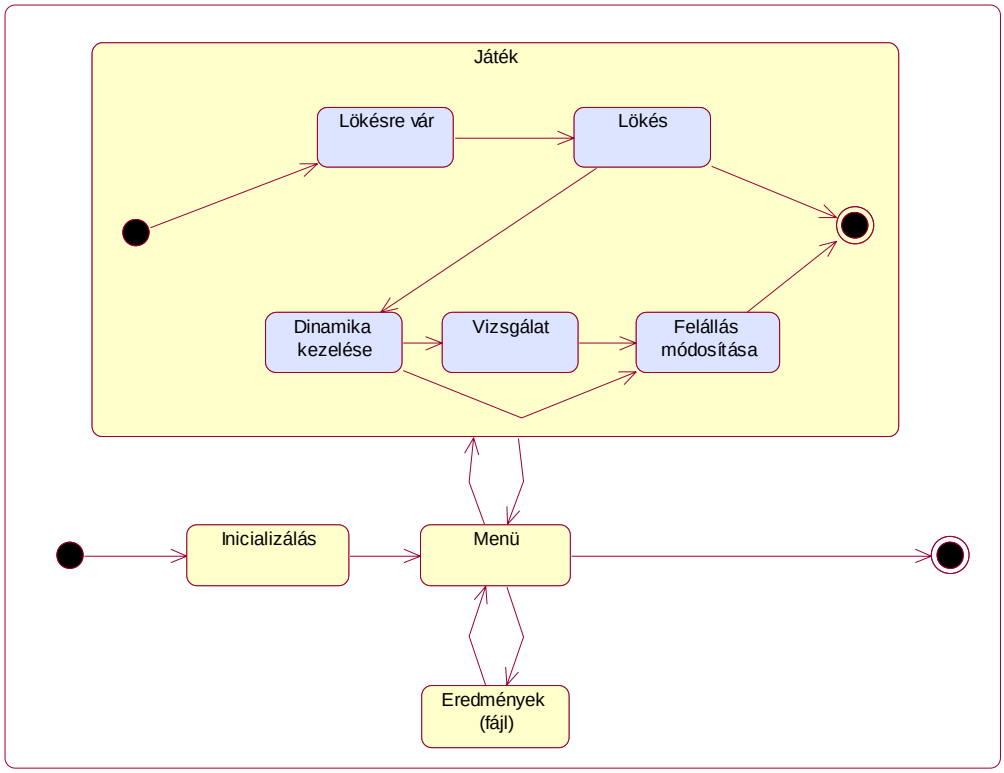
Részletes diagram



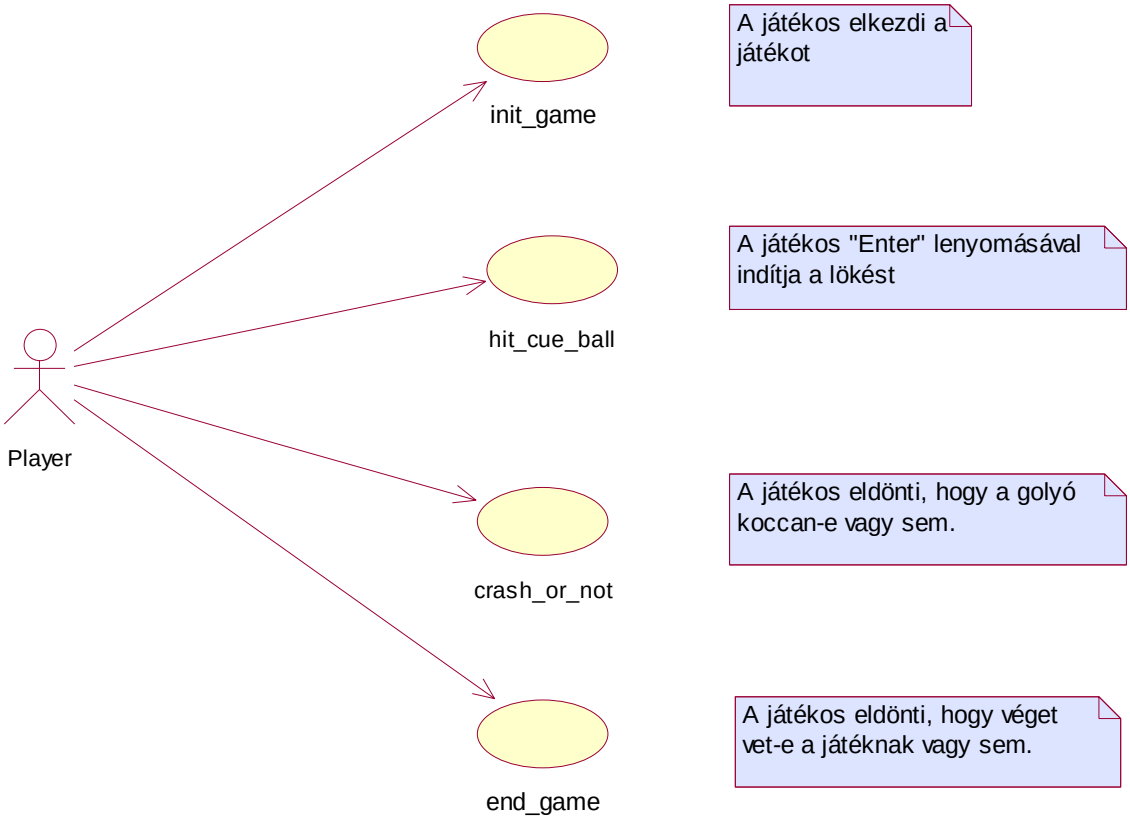
Szekvenciadiagram inicializálásra és Player use case-re



Statechart diagram



A Szkeleton valóságos use case-ei



Felhasználó tevékenysége	Rendszer Válasza
Játék elindítása, nevek bevitele	A rendszer létrehozza a megfelelő objektumokat (billiárdasztal, falak, lyukak, golyók) és lökésre bíztatja a játékost
A játékos meglöki a fehér golyót (ENETER lenyomásával)	A rendszer elindítja a fehér golyót, ami az összes golyótól lekérdezi, hogy ütközött-e vagy sem, majd a golyók is egymástól. (amit jelen esetben a játékos dönt el)
A játékos a játék alatt bármikor kifejezheti a játék befejezésére vonatkozó igényét.	Amint befejeződött a legutóbbi lökéssel járó mozgások lekezelése valamint az adminisztrációs teendők elvégzése, rendszer leállítja a program futását.

Use case-ek leírása

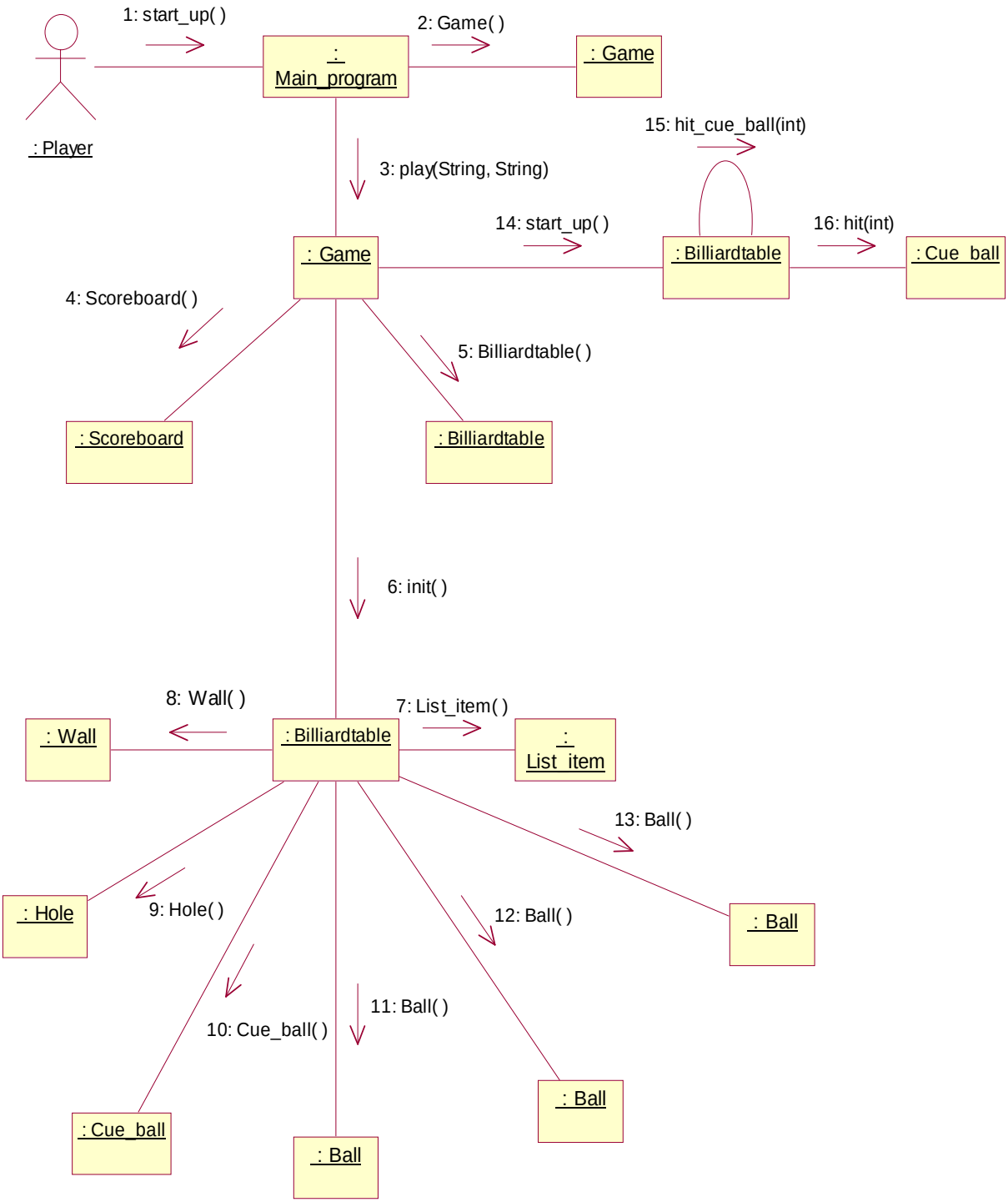
Use Case:	**init_game**
Aktorok:	Player
Cél:	A játék elindítása
Áttekintés:	A Player elindítja a játékot, megadja a nevét. Ezután kezdődhet a játék.

Use Case:	**hit_cue_ball**
Aktorok:	Player
Cél:	A fehér golyó meglökése
Áttekintés:	A Player meglöki a fehér golyót, aminek hatására az sebességgel fog rendelkezni.

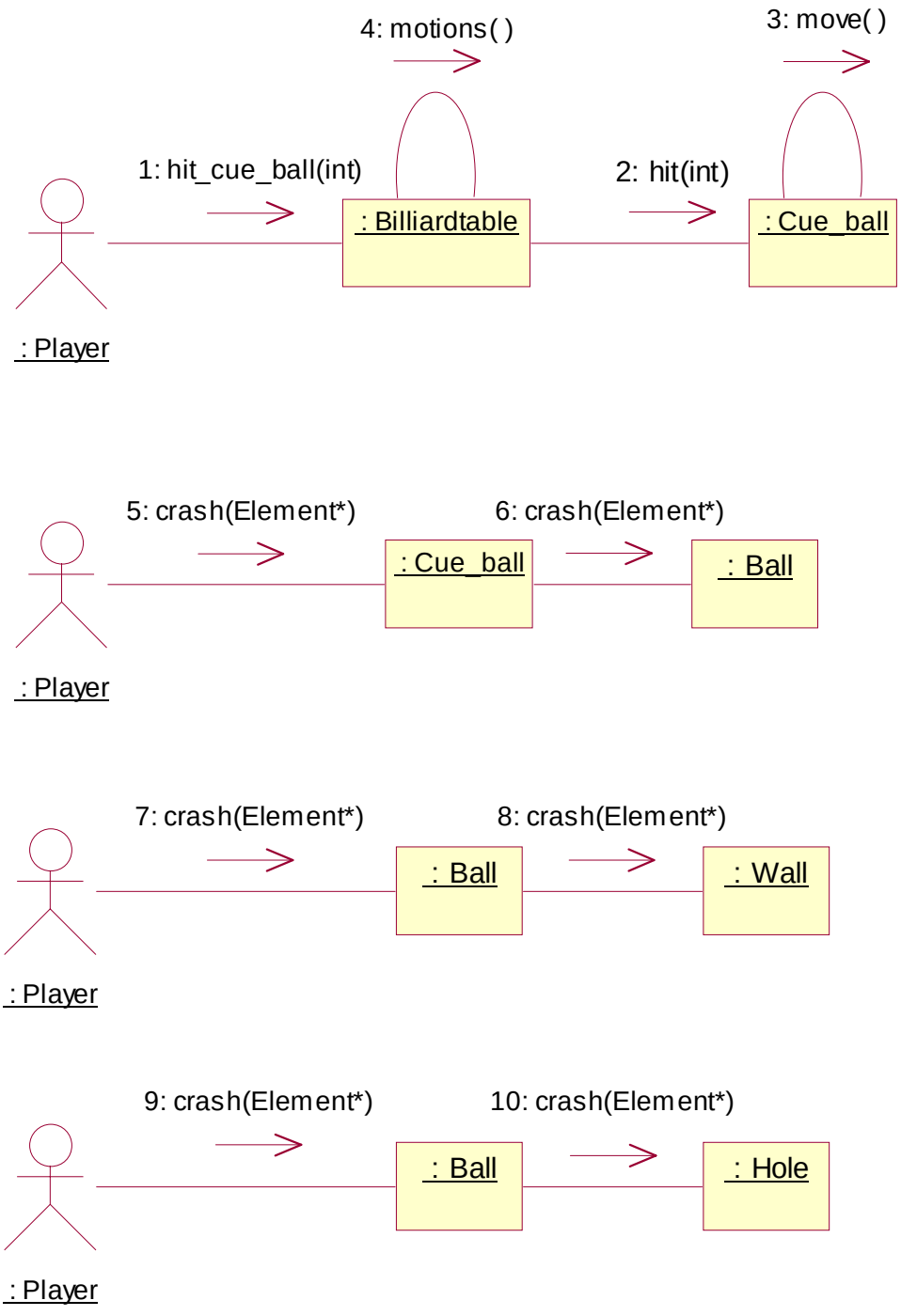
Use Case:	**crash_or_not**
Aktorok:	Player
Cél:	Annak eldöntése, hogy az adott golyók ütköznek-e vagy sem
Áttekintés:	A Player igennel vagy nemmel válaszol a kérdésre, hogy az adott elemek ütköznek-e.

Use Case:	**end_game**
Aktorok:	Player
Cél:	A játék befejezése
Áttekintés:	Ha a Player egy lökés után úgy dönt, hogy nem óhajt tovább játszani, ez irányú óhaját kifejezi, minek hatására a golyók megállása után a billiárdjáték program futása megszakad.

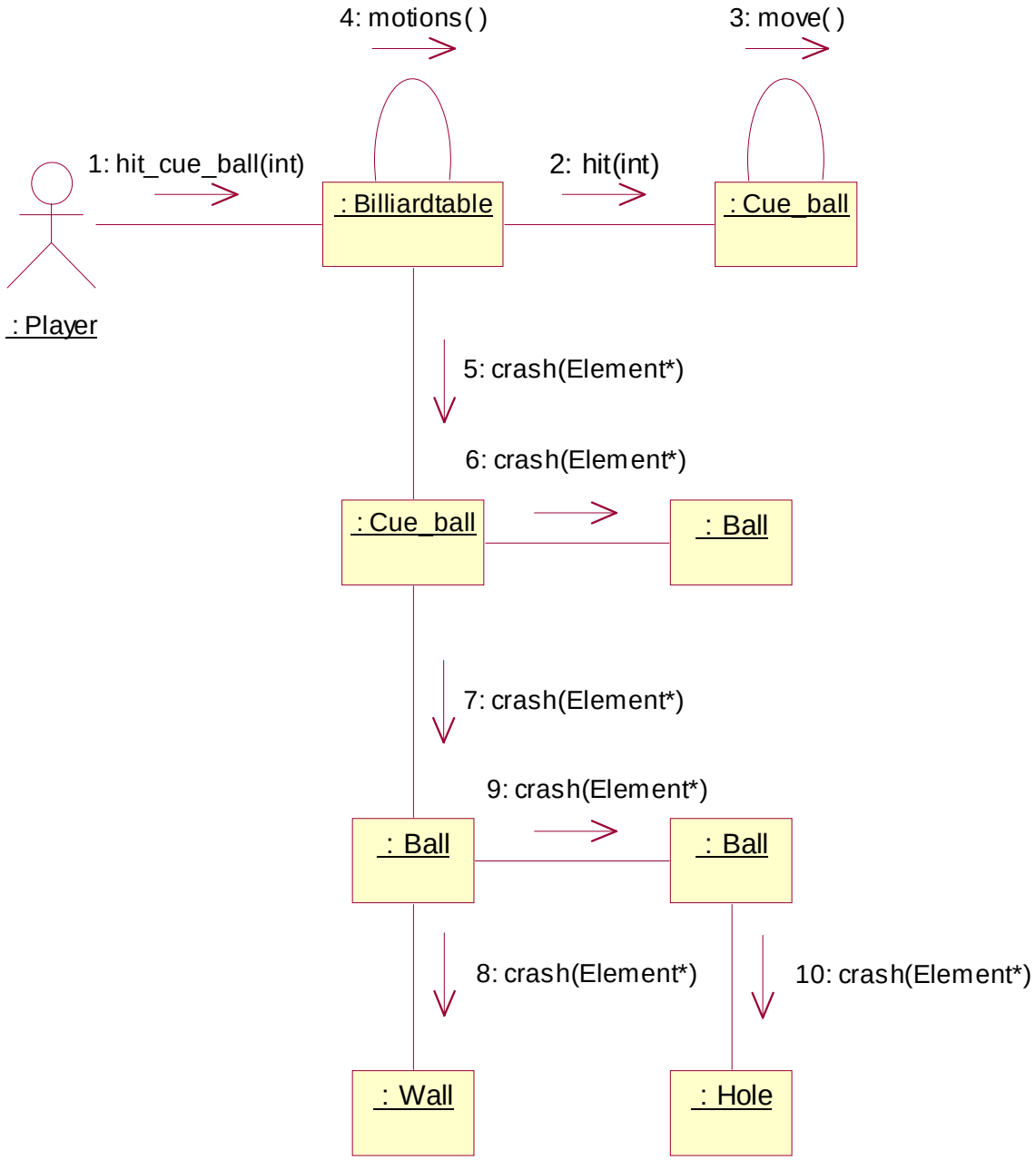
Kollaborációs Diagram Inicializálásra



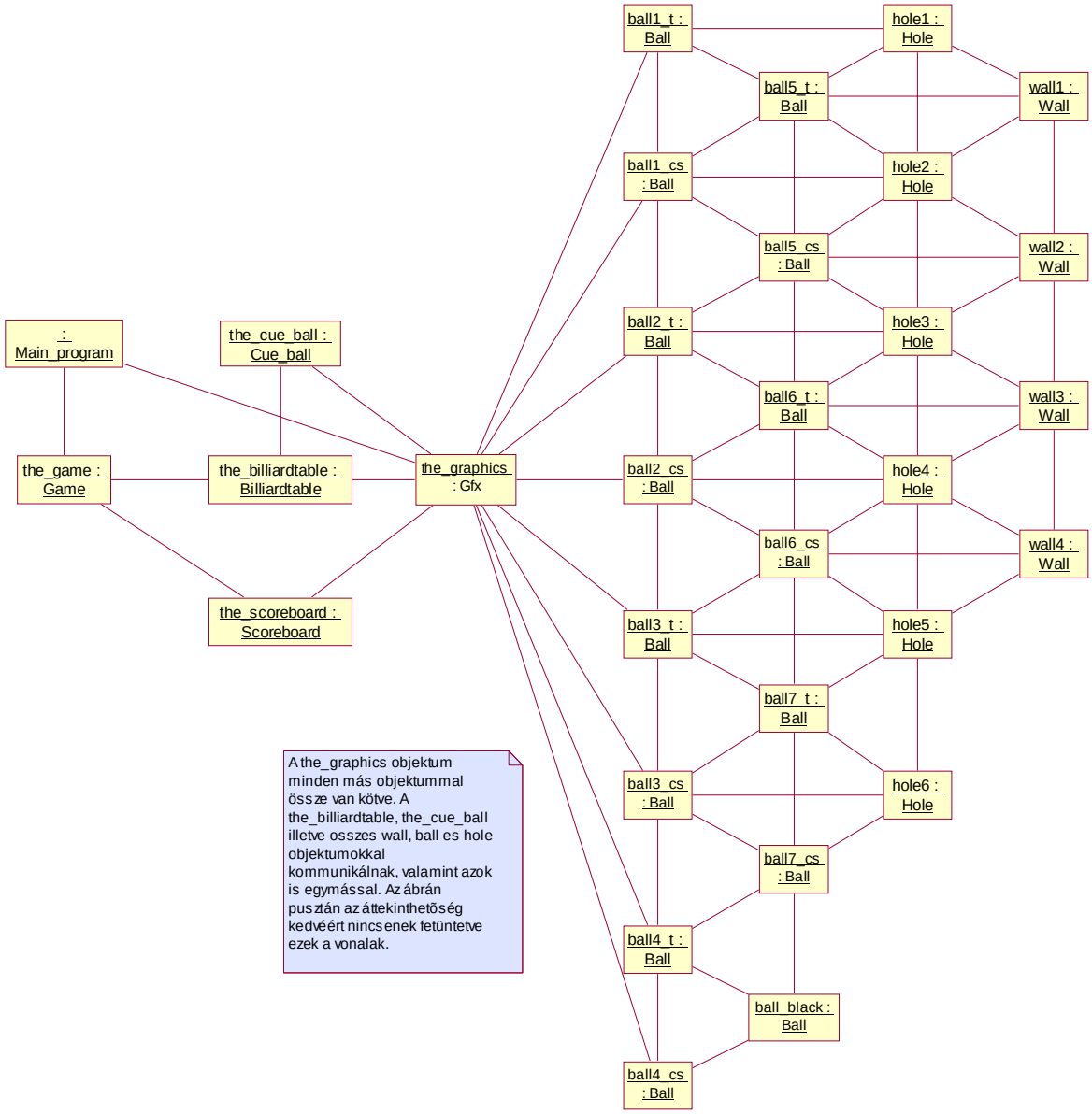
Kollaborációs Diagram a Játék Menetére (Szkeleton modell)



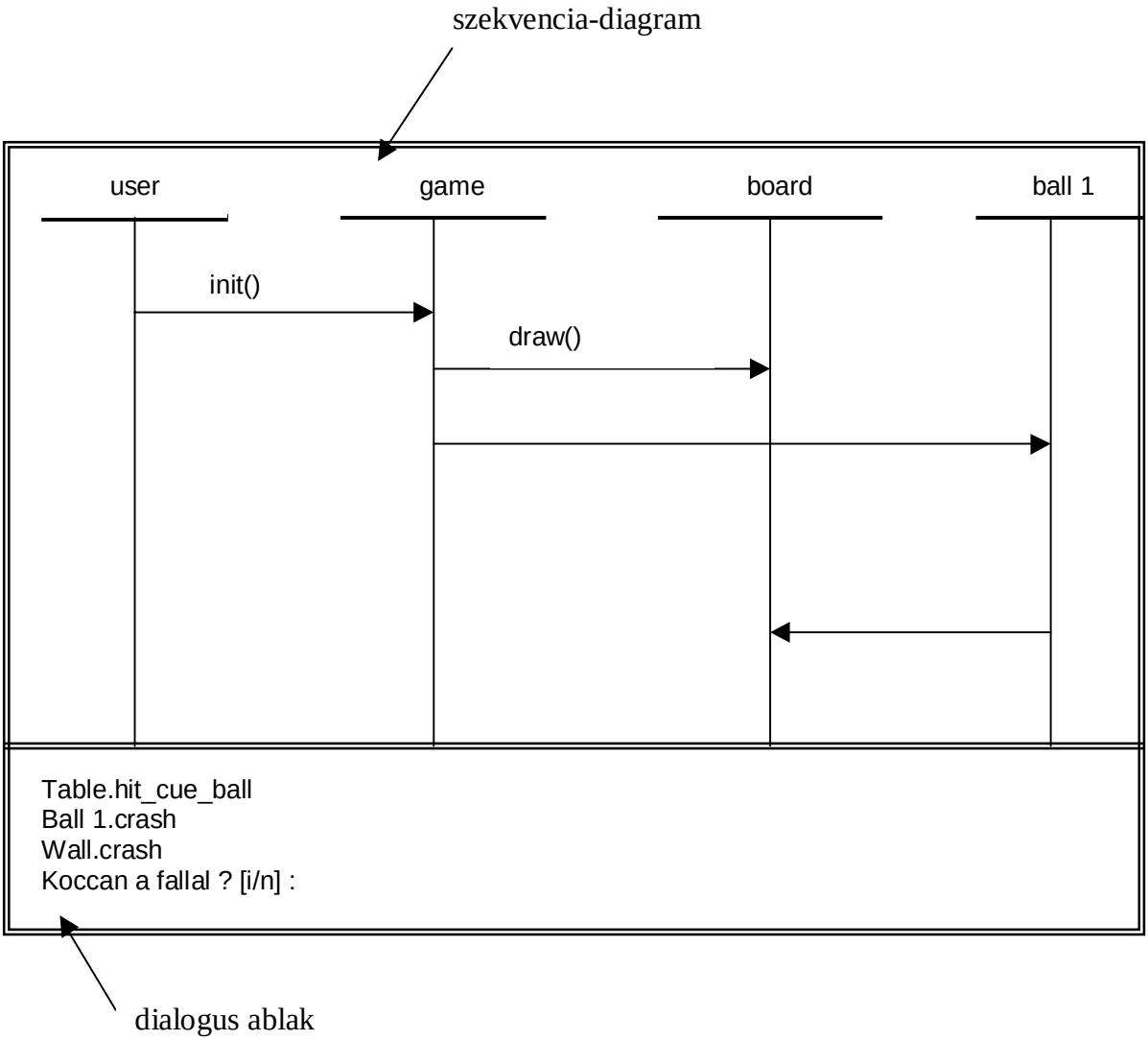
Kollaborációs Diagram a Játék Menetére



Arhitektúra



A kezelői felület terve



Dialógusok

A felhasználó, a dialógusok során mindössze eldöntendő kérdésekre válaszol, vagy jóváhagy döntéseket. Az eldöntendő kérdések a vezérlési elágazásokhoz tartoznak (pl. lyukba esett-e a golyó ? illetve: Nyomjon ENTER-t a fehér golyó meglökéséhez). A szkeleton a futás során mindig kiírja a dialógusablakba az aktuálisan meghívott metódus nevét.

Szkeleton

A program fordításához a tömörített file-ból minden file-t egy munkakönyvtárba kell másolni. A Borland C++ 3.1 fordítóval az Applicat.cpp nevű file-t kell megnyitni. Ezt követően be kell állítani a munkakönyvtár nevét a header-ekben definiált file-ok befordíthatóságához.

A fordítást követően következhet a program futtatása. A futtatás során a különböző fázisok között egy billentyű-leütést vár a program, hogy ily módon a felhasználó a saját ütemében követhesse nyomon a különböző futások szekvenciáit. Bizonyos időközönként a program feltesz kérdéseket. Ezekről a pontokról csak a kérdés megválaszolásával lehet túljutni. Ezek a kérdések a különböző vezérlési elágazásokhoz tartoznak.

A szkeleton moduljai:

Applicat.cpp 1236Byte 2000-03-27

A főprogramot tartalmazza, ami az egésznek a kerete. Ebben szerepelnek a hivatkozások a program többi részére, így ennek lefordításával rendelkezésre áll a szkeleton program.

Skeleton.cpp 15525 Byte 2000-03-27

Ez tartalmazza a tulajdonképpeni keretprogramot. Itt került megvalósításra a teljes képernyőkezelés egy Skeleton típusú skeleton objektumban. A program más pontjai csak a szkeleton objektum megfelelő metódusainak segítségével kommunikálnak a külvilággal.

Ball.hpp 6401 Byte 2000.03.27

A Ball objektum definíciójának helyszine. Ennek beillesztésével rendelkezésre áll a billiárd játék Ball osztályának teljes definícioja (ez az Element-ből öröklődik).

Btable.hpp 8206 Byte 2000.03.27

Ez a Billiardtable osztály definícióját tartalmazó header file.

Cue_ball.hpp 3231 Byte 2000.03.25

A Cue_ball osztály leíró file-ja (ez a Ball-ból öröklődik).

Declarat.hpp 240 Byte 2000.03.25

Az összes osztály előndeklarációját tartalmazó header file. Erre a kereszthivatkozások elkerülése végett van szükség.

Element.hpp 1105 Byte 2000.03.26

A Ball, Wall, Hole osztályok absztrakt szuper-osztályának definíciós file-ja.

Game.hpp 5661 Byte 2000.03.25

A Game osztály leírója.Az Element-bõl öröklõdõ Hole osztály definícióját tartalmazó header file.

Hole.hpp 3736 Byte 2000.03.27

Az Element-bõl öröklõdõ Hole osztály definícióját tartalmazó header file.

Main_pr.hpp 4575 Byte 2000.03.26

A Main_program osztályt leíró file. Az Applicat.cpp főprogram egy Main_program objektumot tartalmaz. Abban kerül megvalósításra az egész vezérlés.

Scboard.hpp 8962 Byte 2000.03.27

A kijelző-t definiáló header file.

Vectors.hpp 8186 Byte 2000.03.25

A grafikához használatos Vectors objektum definícióját tartalmazó header file.

Wall.hpp 4336 Byte 2000.03.27

Az Element-ből öröklődő Wall objektum definíciós file-ja.

Gfx.hpp 13102 Byte 2000.03.27

A grafikai osztályt implementáló header file.

A csapatmunka értékelése

A csapat tagjai körülbelül egyformán vették ki részüket a munkából. Úgy tűnik, hogy a csapat - a kezdeti kisebb súrlódások és nehézségek után - jól tud együtt dolgozni.

Problémáink főleg abból adódtak, hogy mindenki többet akart elvégezni, mint amennyit rá a megbeszéléseken kiosztottunk, így voltak olyan anyagrészek, amelyek egyszerre több példányban is rendelkezésre álltak a beadás napja előtti estén. Ilyenkor komoly dilemmát okozott annak eldöntése, hogy végül is melyik a jobb, melyiket adjuk be. Ez azonban inkább előnyére vált munkánknak, mintsem hátrányára.

A prototípus koncepciója

A prototípus koncepciója

A prototípus alapjában véve kétféle koncepció szerint készülhet és fog készülni. Az első koncepciónál főként a tesztelési lehetőség megteremtését tartjuk szem elõtt figyelmen kívül hagyva minden mást. Ez lényegében annyit tesz, hogy a kezelői felület nem túlságosan felhasználó barát (jobban mondva egyáltalán nem az), mivel az eredeti célkitűzés szerint a bemenet file-ból jön majd. Így a felhasználó interaktivitásának megszűnésével feleslegessé válik az adatok „kultúrált" megjelenítése, mivel azok úgy is túlságosan gyorsan változnak majd ahhoz, hogy azokat ellenőrizni vagy akár csak felfogni is képes legyen. Ennek az egésznek az a célja, hogy a file-ból érkező - elõre jól meghatározott – bemenetből használható formátumú kimenetet generáljon, ami nagyban elősegíti a tesztelést, így végeredményben ez a koncepció adja majd a teszteléshez használt prototípus elvi alapjait.

A másik koncepció szerint az is fontos, hogy a felhasználó valamely számára is (könnyen) értelmezhető felbontásban kapja az számára szükséges adatokat. Természetesen azt ennél az alapszemléletnél sem feltételezhetjük, hogy a felhasználó minden egyes mozzanatnál látni kívánná a történteket. Inkább csak amolyan történés-áttekintési feladatot lát el a proto. Nyilvánvalóan átláthatónak kell lenni, hogy érzékelhető legyen, ki / kivel / mit / mikor / hogyan tesz vagy tenne. A bemenet és a kimenet itt is lehet file-ból illetve file-ba irányított, csak legfeljebb ez utóbbi közelről sem lesz szemet gyönyörködtető.

Lényegében a két alapszempont összeegyeztethető lenne, de csak abban az esetben, ha szakítanánk a csapat egy másik, nem kevésbé fontos irányelvével, a platform-függetlenséggel. Ugyanis hatékonyan használható file-kimenet generálása mellett átlátható képernyő-kimenettel csak 'gotoxy' jellegű függvények alkalmazása mellett szolgálhatnánk, ami komoly megkötést jelentene az operációs rendszert illetõen. Ez pedig a felhasználó körülményeit és igényeit nem ismervén lehetőség szerint kerülendő.

A prototípus interface definíciója

A prototípus kezelői felülete egyszerű kell legyen. A felhasználónak ugyan kell valamit értenie abból, ami történik, de nem muszáj, hogy feltétlenül világos legyen számára az egész folyamat. Fontos, hogy lehetőleg minél kevesebb billentyűre legyen szükség a proto kezeléséhez. Jelen esetben (a nevek beírásától eltekintve) bőven elegendő hat billentyű (négy billentyű az állításhoz /'8', '4', '6', '2'/, egy a lövéshez /ENTER/ és egy a kilépéshez /ESC/).

Az első koncepció szerint a felhasználó majd nem lát semmit az adatokból (közvetlen futtatásnál azért, mert gyorsan „fut” a képernyő, közvetettnél pedig azért, mert az adatok el lesznek rejtve a felhasználó szeme elől /file-ba lesznek irányítva a tesztelés elősegítése érdekében/). A közvetett futtatás egy keretprogram segítségével végezhető, amely egy bemeneti file-ból adja a programnak az adatokat és egy kimeneti file-ba várja az adathalmazt, amit azután feldolgoz és a képernyőre csak az esetleges hibákat illetve a statisztikai adatokat írja ki. A kimeneti file-ban az adatok táblázatos elrendeződésben helyezkednek majd el méghozzá úgy, hogy minden egyes lépés adatai egy sorban legyenek (ez megkönnyíti a feldolgozást)…

A második koncepció szerint a kiírás minden esetben „szép” és átlátható kell legyen, tehát egy szemnek kellemes elrendeződést kell létrehozni a képernyőn, amit legalább is a funkcionális működés helyességének ellenőrzését lehetővé teszi. Ez a kimenet i file-ba írható ugyan, de kezelése jóval nehézkesebb, mint az első koncepcióé.

Az első koncepció messzemenőkig alkalmasabb a jelen feladat megoldására, mivel a proto csupán tesztelési feladatot lát el és így semmi értelme a felhasználó elé tárni olyan adatokat, amelyekkel amúgy sem tud mit kezdeni.

A file-ból történő bemenet formátuma mindkét esetben ugyanaz. A bemenetet szolgáltató file-ban ASCII kódokkal kell megadni az adatokat. A használható karakterek ASCII kódjai a következők:

a, b, c… z (kis angol „abc”)	97, 98, 99… 122
A, B, C… Z (nagy angol „abc”)	65, 66, 67… 90
’2’, ’4’, ’6’, ’8’ (le, balra, jobbra, fel)	50, 52, 54, 56
ENTER	13
ESC	27

A bemenet pontos formátuma a proto beadásakor, annak mellékleteként lesz pontosan specifikálva.

A kimenet – mint az fentebb említésre került – a két koncepció esetén különbözik. Az elsőnél a kimeneten először megjelennek a program adatai (kik készítették, mikor és milyen célból) majd a játék alapadatai táblázatosan (játékosok nevei, asztal adatai / falak, lyukak helyzete logikai koordinátákkal megadva/). Ezek a játszma folyamán nem változnak, így többször szükségtelen megjeleníteni őket. Ezek után fordulónként megjelennek (szintén táblázatosan) a kijelző adatai (golyó típusok, golyó számok, stb.) és a lökés adatai. Itt az interaktivitásból adódóan megjelennek az állítás pillanatnyi eredményei is (ezt persze ki lehetne küszöbölni, de csak úgy, hogy az ’stderr’ kimenetet is használatba vesszük, ami indokolatlan esetben nem igazán elegáns megoldás és az állítások korrekt lefolyásának vizsgálata sem utolsó szempont). Innentől minden egyes „lépés” után kiírjuk az összes golyó összes releváns adatát (pozíció, sebesség, kivel ütközött /ezt mondjuk két 16 bites számmal ábrázolva/). Ez jól használható formátum a tesztelés során (egyszerre vizsgálandó adatok egy sorban…).
Példaként itt egy kimeneti file struktúrájának mintája:

Magic Balls proto v0.96,
Copyright © 2000 RoothLess

*** PLAYERS ***
player1 - <elsõ játékos neve>
player2 - <második játékos neve>

*** TABLE ***

element	position
wall #1	(<x>, <y>)
…	…
wall #4	(<x>, <y>)
…	…
hole #6	(<x>, <y>)

The Game

*** SCOREBOARD ***

data	player1	player2
balltype	<golyótípus>	<golyótípus>
ballnumber	<golyószám>	<golyószám>

*** HIT ***

Angle: <lökés iránya>
Stregth: <lökés nagysága>
…
Angle: <lökés iránya>
Stregth: <lökés nagysága>

*** MOTIONS ***

ball #1				…
position	velocity	crash1	crash2	…
(<x>, <y>)	(<x>, <y>)	<ütk.kódja>	<ütk.kódja>	…
…	…	…	…	…
(<x>, <y>)	(0,0)	0	0	…

*** SCOREBOARD ***

…

A játszma után jövő rész akárhányszor ismétlődhet (ez a játék menetétől függ). És ezen belül az lökés adatai is akárhányszor megjelentetnek (ez a beállítást végző bemeneteken múlik).

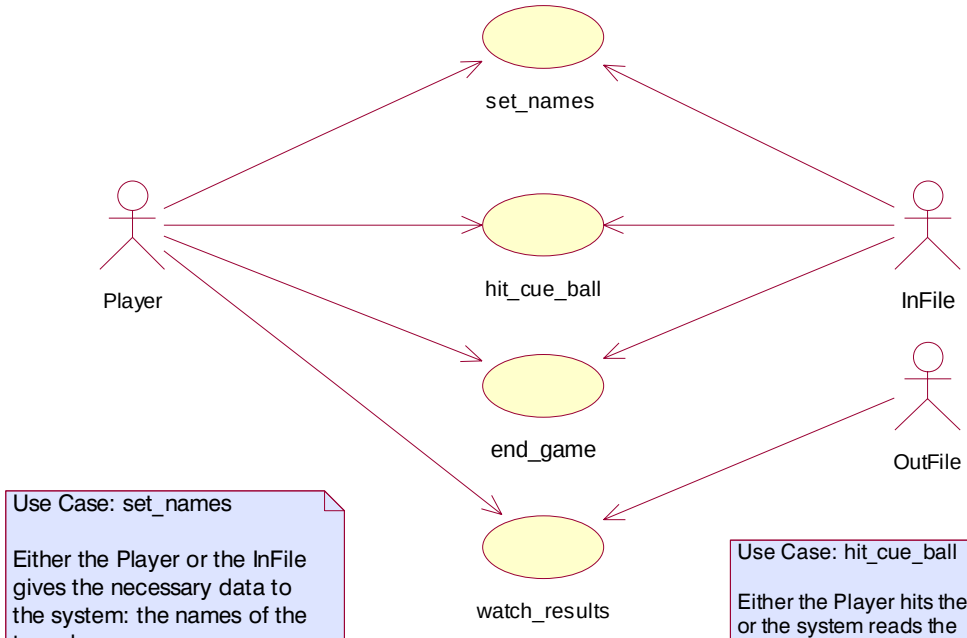
A második koncepció szerint a kimenetnél a képernyőn is „értelmezhetően” és szemkímélően jelennek meg az adatok. Ez azt jelenti, hogy az adatok kiírásának menete ugyan megmarad, de az elrendeződésük más lesz. Még pedig annyiban, hogy a golyók adatai pozíciójukra nézve statikusan kerülnek kiírásra (állandó felülírással). Ennek a kiírásmódnak a részletes specifikációja lényegében szükségtelen, mivel különösebb jelentőséggel nem bír. A file-ba irányított formája ugyan felhasználható tesztelési célokra, de nehézkesen és emiatt bölcsebb a fentebb leírt módszert alkalmazni…

Tesztelési terv

Meglehetősen nehéz a tesztelés tárgyának meghatározása. Hatékonyan (nagy adathalmaz vizsgálatával) csak a megírt algoritmusok helyességét van módunk ellenőrizni mégpedig oly módon, hogy újra alkotjuk azokat és a kimeneti file-ban levő adatokra alkalmazzuk őket, majd az eredményeket összevetjük a szintén kimenetként kapottakkal. A kimeneti file természetesen nem árt, ha olyan formátumú, hogy a tesztelés elvégeztével a felhasználó is elemezhesse a történteket, ha kívánja…

Ezen túl csak aránylag kis mennyiségű minta tesztelésével juthatunk közelebb az esetleges hibák felderítéséhez. Ez kézi ellenőrzést jelent, ami kevéssé hatékony, viszont talán biztosabb hibafelderítést eredményez (egy algoritmust el lehet rontani kétszer is ugyanúgy /programozás-technikailag/).

A prototípus átfogó use casei



Use Case: set_names

Either the Player or the InFile gives the necessary data to the system: the names of the two players

Use Case: hit_cue_ball

Either the Player hits the cueball or the system reads the important settings (strength and direction) from the InFile.

Use Case: watch_results

The system writes the results either onto the screen (where the Player can watch them) or in the OutFile (which the tester program can compare to the file with the good results.

Use Case: end_game

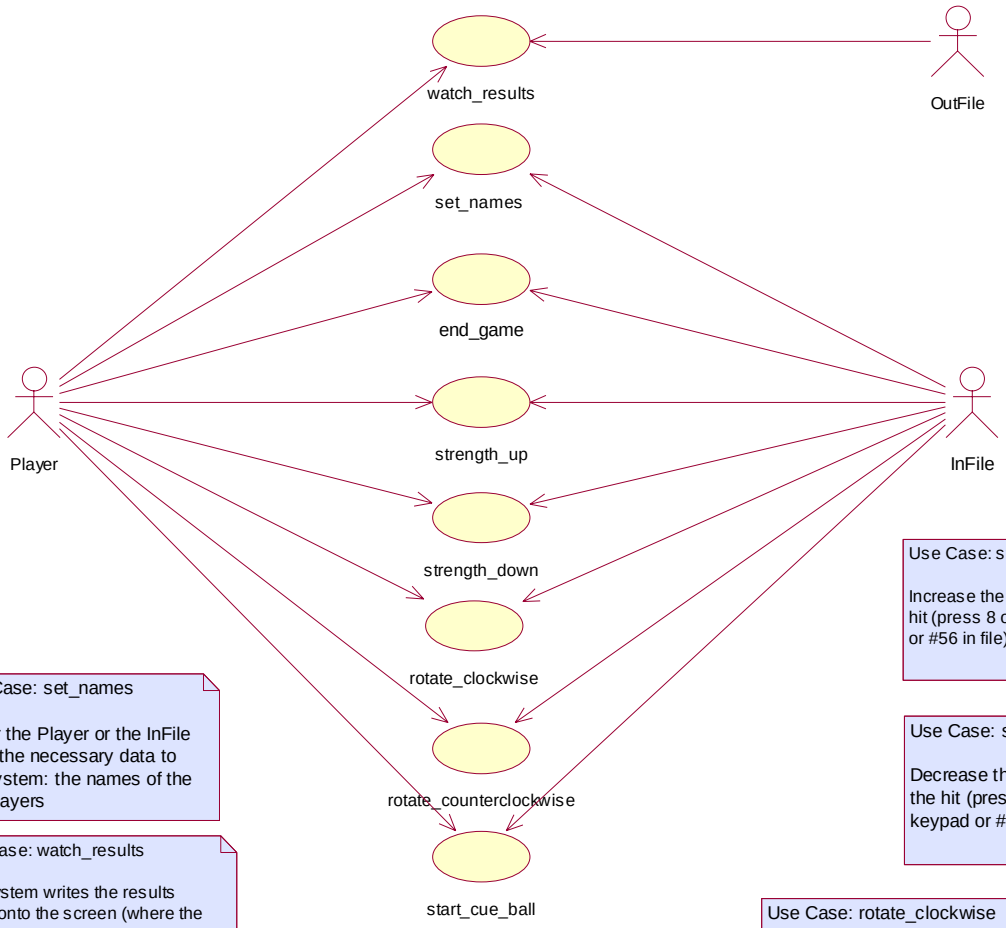
The Player is allowed to express his/her wish to finish the game. Anywhere in the InFile there can be a special character or string which causes the same effect that the player would when trying to finish the game.

Átfogó use case-k részletes leírása

Felhasználó tevékenysége	Rendszer Válasza
Játék elindítása, nevek bevitele	A rendszer létrehozza a megfelelő objektumokat (billiárdasztal, falak, lyukak, golyók) és lökésre bíztatja a játékost
A játékos meglöki a fehér golyót (ENETER lenyomásával)	A rendszer elindítja a fehér golyót, majd az összes golyótól lekérdezi, hogy ütközött-e vagy sem (amit jelen esetben a játékos dönt el)
A játékos ESC leütésével a játék alatt bármikor kifejezheti a játék befejezésére vonatkozó igényét.	Amint befejeződött a legutóbbi lökéssel járó mozgások lekezelése valamint az adminisztrációs teendők elvégzése, rendszer leállítja a program futását.

Fájl “tevékenysége”	Rendszer “tevékenysége”
Külső személy indítja a játékot (vagy valamilyen paraméter formájában adja meg a fájl nevét, vagy a standard inputot és outputot irányítja át.	A rendszer létrehozza a megfelelő objektumokat (billiárdasztal, falak, lyukak, golyók) és beolvassa az InFile-ból a lökésre vontatkozó paramétereket (irány és erősség)
Az InFile “megadja” a rendszernek a szükséges paramétereket a lökéshez.	A rendszer elindítja a fehér golyót, majd az összes golyótól lekérdezi, hogy ütközött-e vagy sem (amit jelen esetben a játékos dönt el). Az eredményeket a rendszer az OutFile-ban rögzíti.
Az InFile-ban bármikor lehetspeciális karakter (ESC - 027), amely a játék befejezésére vonatkozó igényt jelzi.	Amint befejeződött a legutóbbi lökéssel járó mozgások lekezelése valamint az adminisztrációs teendők elvégzése, rendszer leállítja a program futását.

Részletes use case diagram



Use Case: strength_up

Increase the strength of the hit (press 8 on the keypad or #56 in file)

Use Case: set_names

Either the Player or the InFile gives the necessary data to the system: the names of the two players

Use Case: strength_down

Decrease the strength of the hit (press 2 on the keypad or #50 in file)

Use Case: watch_results

The system writes the results either onto the screen (where the Player can watch them) or in the OutFile (which the tester program can compare to the file with the good results.

Use Case: rotate_clockwise

rotate the direction of the hit clockwise (press 6 on the keypad or #54 in file)

Use Case: start_cue_ball

Launch the cue call with the parameters set before.

Use Case: end_game

The Player is allowed to express his/her wish to finish the game (ESC). Anywhere in the InFile there can be a special character (code 027) which causes the same effect that the player would when trying to finish the game.

Use Case: rotate_counterclockwise

rotate the direction of the hit counterclockwise (press 4 on the keypad or #52 in file)

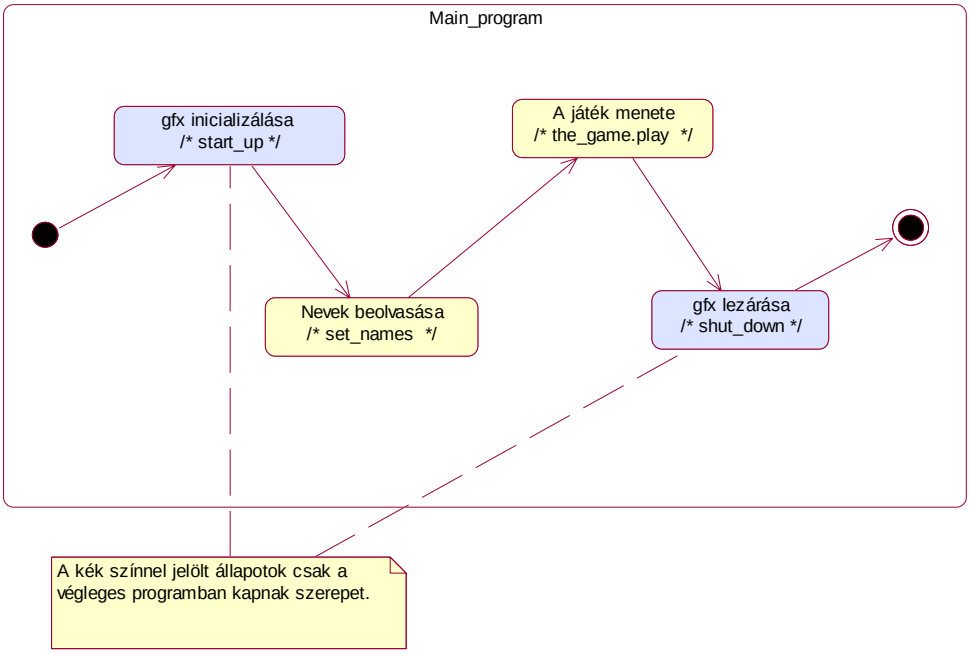
Újabb use case-k részletes leírása

Felhasználó tevékenysége	Rendszer Válasza
A játékos a keypad-en (Num Lock ON) a fel és a le nyíllal (8. ill. 2.) tudja szabályozni a lökés erősségét.	A rendszer megnöveli a lökés kezdősebességét annyi egységgel, ahányszor a játékos megnyomta a 8 gombot (illetve csőkkenti a 2 gombra)
A játékos a keypad-en (Num Lock ON) a balra és a jobbra nyíllal (4. ill. 2.) tudja szabályozni a lökés irányát.	A rendszer forgatja a dákót annyi egységgel óra járásával megegyező irányban, ahányszor a játékos megnyomta a 6 gombot (illetve ellenkező irányba a 4 gombra)
A játékos az ENTER lenyomásával indíthatja a fehár golyót.	A rendszer meglöki a fehér golyót, majd ezután az előzőekben leírt módon folytatja tovább működését.

InFájlban lévő adatok	Rendszer “tevékenysége”
Annyi db. 8 illetve 2 szám (integer), ahány egységgel növelni illetve csökkenteni akarjuk a lökés erősségét.	A rendszer beolvassa az InFile-ból az adatokat, majd annak megfelelően annyi egységgel változtatja lökés kezdősebességét, ahányszor 8 ill. 2 szám van az InFile-ban.
Annyi db. 6 illetve 4 szám (integer), ahány egységgel óra járásával megegyező illetve azzal ellenkező irányba akarjuk fordítani a dákót.	A rendszer beolvassa az InFile-ból az adatokat, majd annak megfelelően annyi egységgel változtatja lökés irányát, ahányszor 6 ill. 4 szám van az InFile-ban.
Az InFile-ban szintén az Enter (013) jelzi a lökés engedélyezését.	A rendszer beolvassa az adatot, meglöki a fehér golyót, majd ezután az előzőekben leírt módon folytatja tovább működését.

Statechart, activity diagrammok és részletes szöveges leírás

Main_program osztály



A Main program osztály metódusai

`void Main_program :: start_up()`

Meghívja a Gfx objektum `start_up` függvényét, ami a megjelenítést (grafikus vagy alfabetikus) inicializálja.

`void Main_program :: menu()`

A program indulásakor meghívja a `set_players()` metódust, majd a `the_game` objektum `play()` metódusát, aminek hatására elindul a játék.

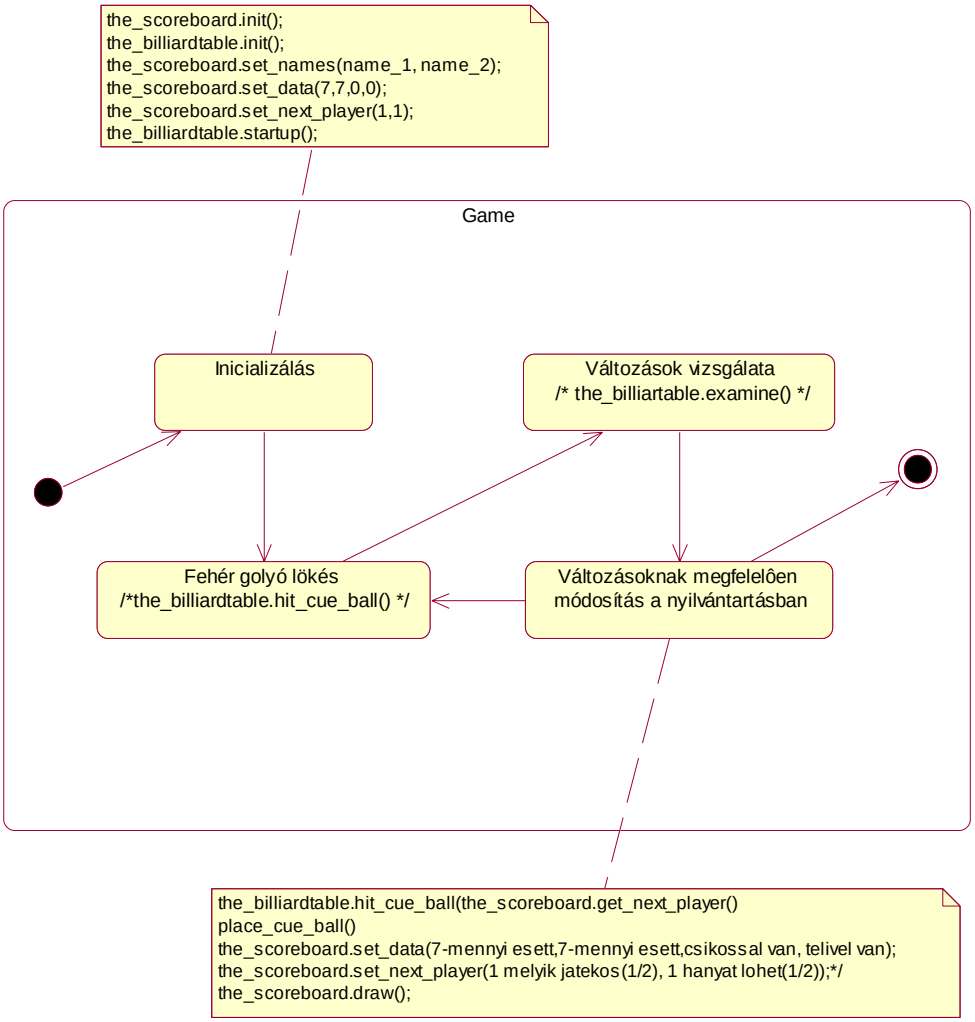
`void Main_program :: shut_down()`

Meghívja a Gfx objektum `shut_down()` metódusát, ami a játék befejezõdése utáni, az interface-re vonatkozó teendõket látja el.

`void Main_program :: set_players()`

A játékosok neveit állítja be, a program indulásakor.

Game osztály



A Game osztály metódusai

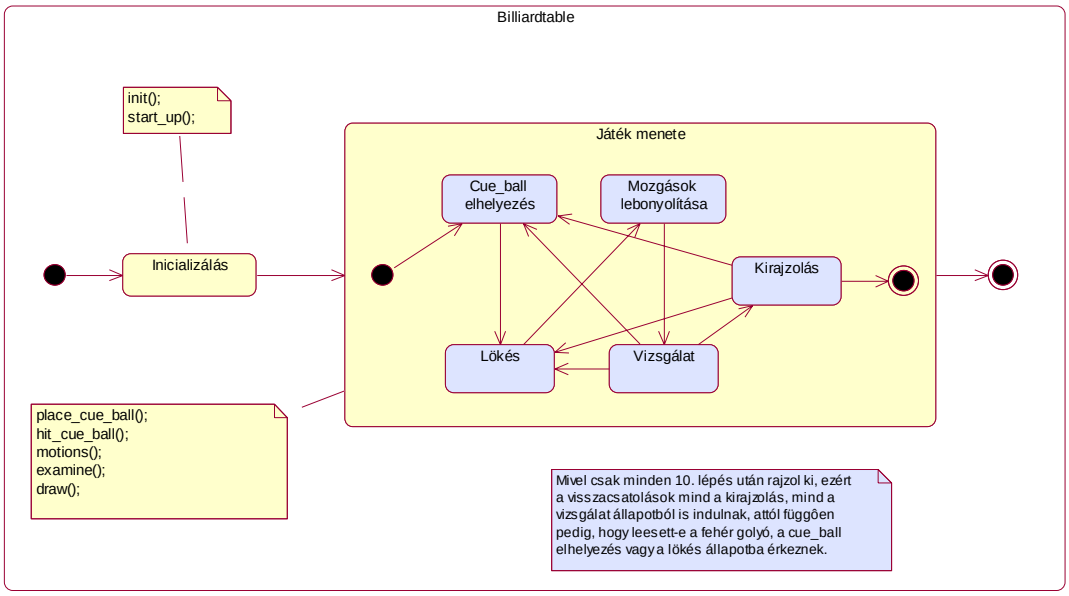
`void Game :: play(char* name_1, char* name_2)`

Ez a metódus vezényli le a játék futását. Elsőként meghívja a `the_billiardtable` objektum `init()` metódusát, mely a billiárdasztal megjelenítéséhez szükséges inicializálást végzi el, majd a `the_scoreboard` objektum `set_names()` metódusát hívja meg. Ez a kijelzőre helyezi a két játékos nevét. Ezt követően ugyaenezen objektum `set_data()` és `set_next_player()` metódusát hívja meg az indításkor alapértelmezett értékekkel.
Mindezek után a `the_billiardtable` objektum `start_up()` metódusa kerül meghívsra, mely inicializálja a `the_billiardtable` objektum adatmezőit, és az asztalra felhelyezi az objektumokat az alapéertelmezett helyükre.
Az előbb felsorolt inicializálási lépések után kerülhet sor a játék tényleges megkezdésére.
A játék egy hátultesztelő ciklusba lett ágyazva. Ez akkor ér véget, ha a fekete golyó pottyan vagy ha a játékos megnyomja a kilépés gombot. Ekkor a játéknak mindenféleképpen meg kell állnia.
A játék menete a következő:

- Meghívásra kerül a `the_billiardtable` objektum `hit_cue_ball()` metódusa az alapértelmezett játékossal. Ezzel a kezdő játékosnak nyilik lehetősége az első ütés elvégzésére. Ez a művelet egy láncolatot indít el melynek során a golyók a helyükre mozognak.
- Az állapot felméréséhez meghívjuk a `the_billiardtable` objektum `examine()` metódusát, mely végigvizsgálja a golyók állapotát. Ennek megfelelően
- meghívjuk a the_scoreboard objektum `set_data()` metódusát a változások mgjelenítésére, illetve
- az ugyanezen objektum `set_next_player()` metódusát, mely a következő játékost állítja be. Attól függően, hogy pottyant-e a fehér golyó
- meghívásra kerülhet a the_billiardtable `place_cue_ball()` metódusa, mely az ilyenkor esedékes fehér golyó-mozgatást végzi el.

Mindezek végén a `the_scoreboard` objektum `draw()` metódusát hívjuk meg, hogy az érvényesített módosítások megjelenjenek a képernyőn is.
A végén az egész folyamat újra indul.

Billiardtable osztály



A Billiardtable osztály metódusai

`void Billiardtable :: init()`

A billiárdasztal komponenseit hozza létre, és fűzi be egy kétirányban láncolt listába. A sorrend a következő : cue_ball, golyók, falak, lyukak. A sorrend fontosságára a mozgások vizsgálatánál még visszatérünk.
A lista elemei `List_item` objektumokból állnak. Minden `List_ item` objektumhoz hozzá van csatolva egy komponens.

`void Billiardtable :: start_up()`

A `start_up()` metódus által a listába fűzött elemek adatait állítja alapértelmezett értékre (pl. a golyók kezdeti háromszög alakú felállítása)

`void Billiardtable :: place_cue_ball(int player)`

Akkor kerül meghívásra, ha a fehér golyó pottyant. A `cue_ball` objektum pozícióját állítja be (a sebességet 0-ra állítja).

`int Billiardtable :: hit_cue_ball(int player)`

Meghívja a `cue_ball` objektum `hit()` metódusát, majd a `motions()` metódust, ami az ütközési láncolatot indítja el.

`int Billiardtable :: examine()`

A `state_of_balls` tömbön halad végig, és begyűjti az aktuális állapotot, amelyet kódolva ad vissza hívójának.

`void Billiardtable :: motions()`

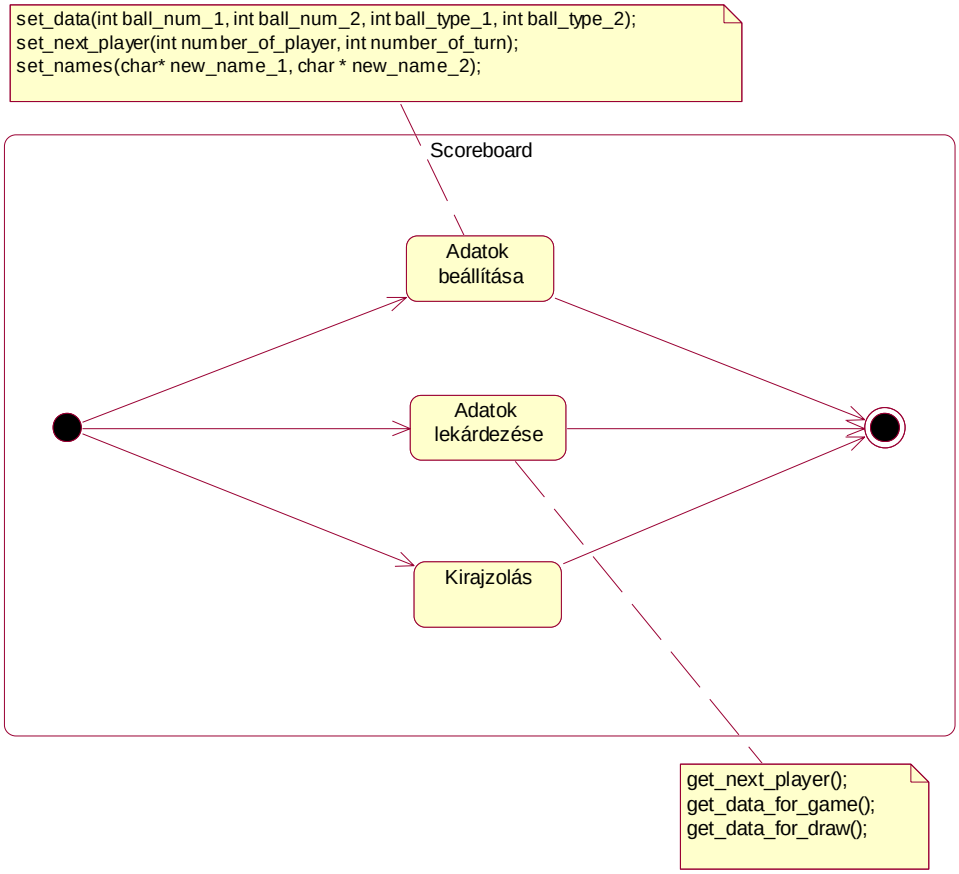
Ez végzi a golyók mozgtását. Egy ciklus indul, mely addig megy ameddig minden golyó sebessége (`Ball::get_velocity()` vektor nagysága) nulla nem lesz.

Sorra meghívja az elemek crash(), és draw() metódusait. Itt válik világossá, hogy miért fontos az elemek önkényesen megválasztott sorrendje: cue_ball,golyók, falak, lyukak. Ugyanis ezt a sorrendet követve a fő cikluson belül egy másik ciklust tudunk indítani, mely a lista elejétől az utolsó golyóig megy, ezen belül pedig még egy ciklus megy végig a lista elemein (egészen a lista végéig) a külső ciklus által meghatározott aktuális elemtől kezdve, és rendre meghívja az elemek `crash()` metódusát, átadva a kijelölt elemet argumentumként, majd a `draw()` metódust mely a megjelenítésért felelős.

`void Billiardtable :: draw()`

Egyszerűen meghívja a Gfx objektum `draw_table()` metódusát, átadva magát paraméterül.

Scoreboard osztály



A Scoreboard osztály metódusai

```
void Scoreboard :: set_data(int ball_num_1, int ball_num_2,
                        int ball_type_1, int ball_type_2)
```

A paraméterként átvett adatokra állítja be a propertyket.

```
void Scoreboard :: set_next_player(int number_of_player,
                                int number_of_turn)
```

A paraméterként átvett értékre állítja a `next_player`, és a `turn_number` változókat.

`void Scoreboard :: draw()`

A Gfx objektum `draw_scoreboard()` metódusát hívja meg mely a kijelző megjelenítését végzi.

`void Scoreboard :: init()`

A property-ket az alapértelmezett értékekre állítja.

`int Scoreboard :: get_next_player()`

Kódolva adja vissza, hogy kinél van a lökés joga, és hogy hányszor lökhet.

`dfg_struc Scoreboard :: get_data_for_game()`

A játékhoz szükséges minden adatot (pl. ki lök, kinek hány lökése van, ki melyik fajta golyóval játszik…) kiszolgáltat egy `dfg_struc` típusnevű rekordban.

`dfg_struc Scoreboard :: get_data_for_draw()`

A megjelenítéshez szükséges adatokat adja vissza hívójának (pl. színkódok, a kijelzőn elhelyezkedő adatok helyei…).

`void Scoreboard :: set_names(char* new_name_1, char* new_name_2)`

A játékosok neveit állítja be az átvett értékekre.

Az Element osztály metódusai

`void Element :: set_position(Vectors vector)`

Az elem pozícióját állítja be.

`Vectors Element :: get_position()`

Az elem pozícióját adja vissza.

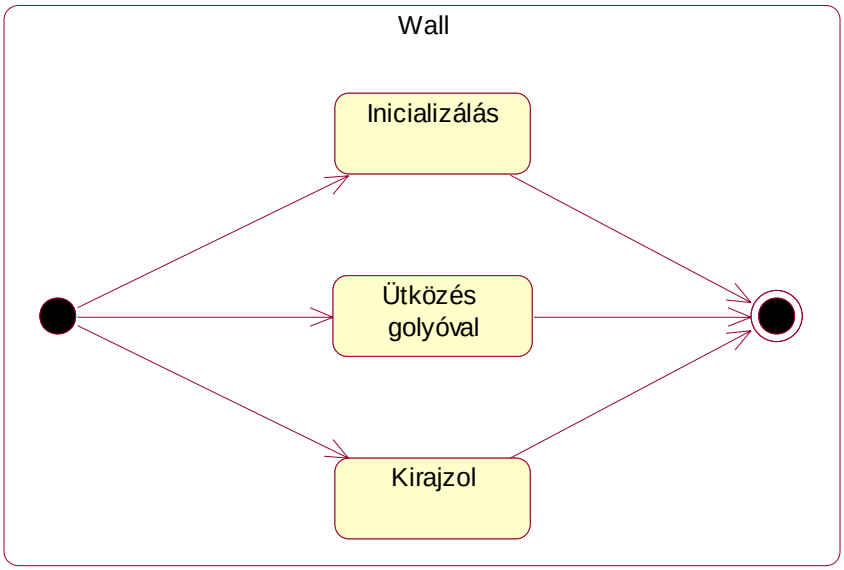
`void Element :: set_color(int color)`

Az elem színét állítja be.

`int Element :: get_color()`

Az elem színét adja vissza.

Wall osztály



A Wall osztály metódusai

`void Wall :: draw()`

A Gfx objektum draw_wall() metódusát hívja meg.

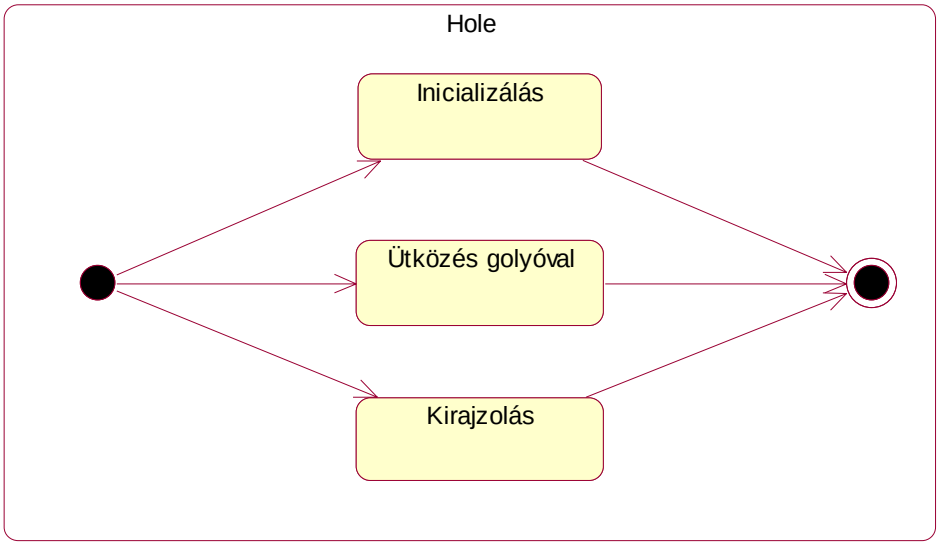
`int Wall :: crash(Element* element_to_crash_with)`

Megvizsgálja, hogy az árvett elem ütközött-e a fallal.

`void Wall :: init()`

A adatokat állítja az alapértelmezettre.

Hole osztály



A Hole osztály metódusai

`void Hole :: draw()`

A Gfx objektum draw_hole() metódusát hívja meg.

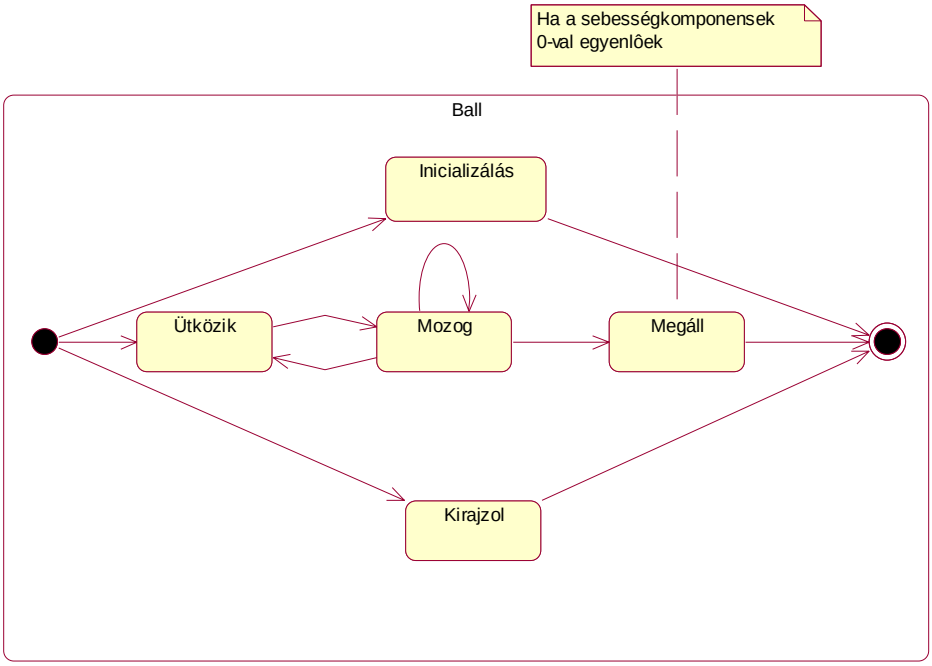
`int Hole :: crash(Element* element_to_crash_with)`

Megvizsgálja, hogy a paraméterként átvett golyó beleesett-e a lyukba, és ennek megfelelően módosítja a the_billiardtable objektum state_of_balls tömbjét.

`void Hole :: init()`

A property-ket állítja az alapértelmezett értékre.

Ball osztály



A Ball osztály metódusai

`void Ball :: draw()`

A Gfx objektum `draw_ball()` metódusát hívja meg, mely a golyó megjelenítéséért felelős.

`int Ball :: crash(Element* element_to_crash_with)`

Megvizsgálja, hogy a golyó ütközött-e a paraméterként átvett golyóval. Ha igen, akkor módosítja annak pozícióját, és sebességét a `set_position()` és `set_velocity()` metódusok segítségével.

Két golyó ütközésekor a sebességek ütközés síkjára merőleges összetevői felcserélődnek, az ütközés síkjával párhuzamos összetevői nem változnak.

`void Ball :: move()`

A golyó pozícióját módosítja a sebességvektorának megfelelően, egy előre meghatározott ds mértékkel.

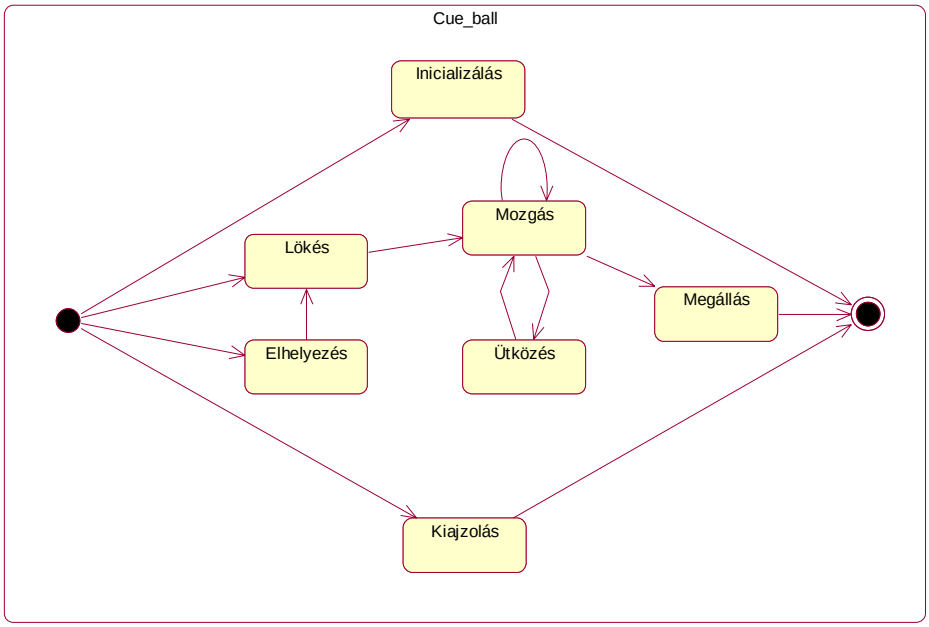
`void Ball :: init()`

A golyó paramétereit állítja a kezdeti értékekre.

`void Ball :: set_velocity(Vectors vector)`

A golyó sebességét leíró privát változót módosítja a megadott értékre.

Cue_ball osztály



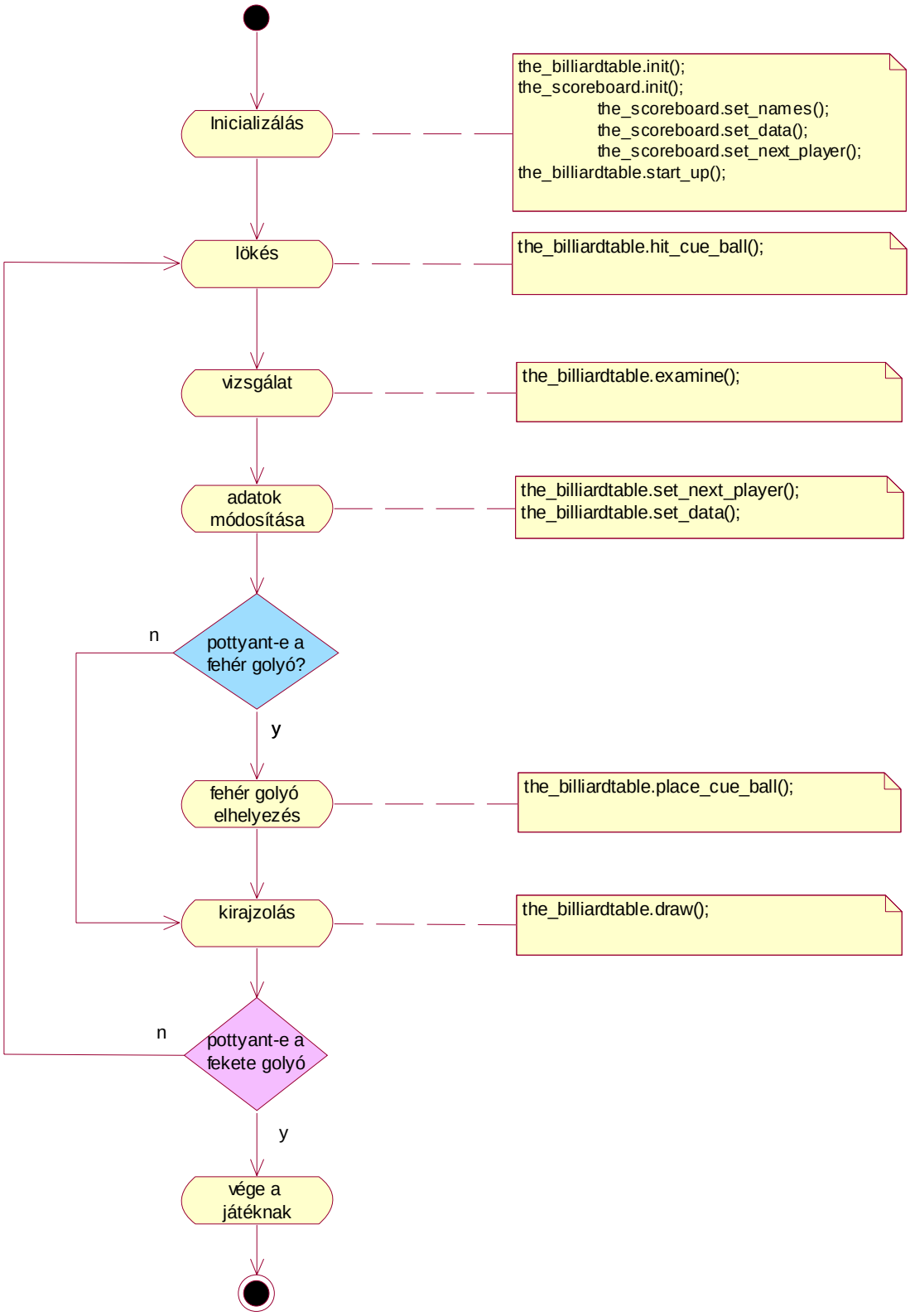
A Cue ball osztály metódusai

`void Cue_ball :: hit(int player)`

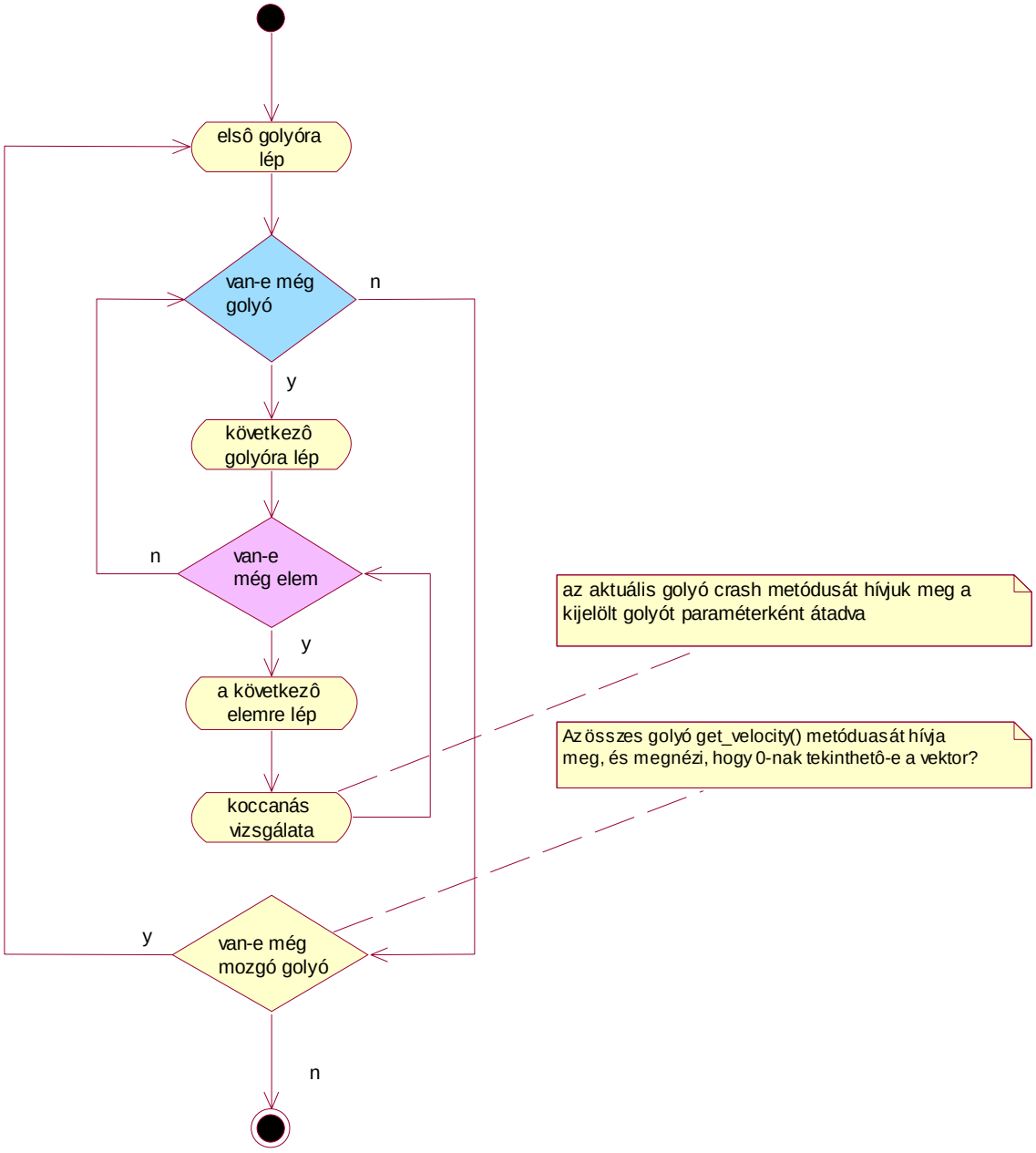
A billentyűzet által vezérelve ($\rightarrow(6)$,$\leftarrow(4)$,$\uparrow(8)$,$\downarrow(2)$) állítja az ütés szögét, illetve erejét, és a Gfx objektum draw_cue() metódusát hívja ennek megfelelően ami a dákót jeleníti meg a user interface-en. Ezt követően beállítja a cue_ball sebességvektorát.

Activity diagramok

A Game osztály `play()` metódusa



A Billiardtable osztály `motions()` metódusa



A tesztek részletes tervei

A tesztelés során különböző bemeneti adatsorozatokat adva a programnak, próbáljuk vizsgálni helyes működésének elvét. Az adatsorozatok különbözősége lehanyagolható, és a szélsőséges működés elvére korlátozódnak. A bemeneti sorozatok az üres járatot, illetve a normál játék inputját kapják. Több esetvizsgálat használata felesleges, mert üresebbnél, a program nem tud kevesebbel indulni, és normális – szabályos – használat esetén az adatoknak megfelelő helye van, több helyre nem illeszthető, a program lenyeli.

A logfile

A kimeneti file első sorai tartalmazzák a program nevét, a logfile keletkezésének dátumát, a csapat nevét. Az általunk beállított értékek a kimeneti fileban tárolódni fognak. Következő tárolásra kerülő adatok lesznek a játékosok nevei, és a konstans értékek (falak és lyukak). Ezen entitásokhoz tartozó pontos konstans értékek még nem ismeretesek, mert azok csak a futtatás során értékelődnek ki, a minél pontosabb és élvezhetőbb játékfelület érdekében. Ugyanis az értékekek fő meghatározója a grafikus felület lesz, hiszen felbontás alapján fogjuk megítélni a falak és a lyukak pontos elhelyeszkedését. A további tárolásra kerülő adatok már a játszma során kerülnek rögzítésre. A `scoreboard` objektumnak mindig az aktualizált változata kerül tárolásra, miután minden elrendeződött, és az asztalon minden nyugalomban van. Ezen aktuális kiírás tartalmazza a játékosok adatait: saját golyó színe és a hátralévő golyók számát. A következő fontos adat a dákó mozgását szemlélteti: leírja annak mozgatását, irányváltásait és leendő lökésének erősségváltoztatását. Minden egyes változtatás rögzítésre kerül a grafikus felület miatt, ugyanis ezt ábrázolni szeretnénk. A komplexitást a logfile elemzésekor a `motions` adja. Ebben tároljuk a golyók mozgását lépésről – lépésre, hiszen ezeket is szeretnénk felrajzolni az asztalra folyamatosan mozogásuk közben. Minden golyóhoz egy négyes tartozik: jelenlegi helyzet, sebesség, ütközés más golyókkal, ütközés fallal (`crash1`) vagy lyukkal (`crash2`). A `crash1` egy 16 bites szám, melyben tároljuk, hogy az adott golyó, mely másikkal ütközött abban a pillanatban- az adott bitet bebillentjük, és átadjuk a megfelelő sebességvektort a másik golyónak. A `crash2` a négy fal, hat lyuk bitjeit tartalmazza. A lyukkal való ütközés nem más, mint a golyót eltettük. Ez többé már nem játszik, adatait már nem kell változtatnunk. Ezalól a fehér golyó tesz kivételt, hiszen azt mindig fel kell helyeznünk az asztalra. Míg a másik speciális tulajdonsággal rendelkező golyó: a fekete, melynek rossz lyukba lövése, illetve idő előtt való elhelyezése a játék azonnali végét jelenti. Egy sorban az adott ütközéshez tartozó mozgások sorozata fog megjelenni. A sorok írása mindaddíg folytatódik, amíg van valmelyik asztalon lévő golyónak sebessége. Amint a nyugalmi helyzet beállt, az eredmények íródnak ki újra, és ez a játék végéig így folytatódik.

Üres jelsorozat

A program elindulás után azonnal várakozó üzemmódba áll. Lényegében ez az interaktív “sima” játék-üzemmód. Ilyenkor a műveleteket kézzel vihetjük be. Elsőként a játékosok neveit kell a rendszernek megadnunk. Ha nem történik értékadás, akkor az alapértelmezés marad érvényben: de ki akarná, hogy becses neve helyett a *Player x* díszelegjen. Utána megjelenő képernyőn a hagyományos játék mozgásait végezhetjük el. A játék folyamán minden szabályos művelet megengedett – irányítás, beállítás, lökés – és rögzítésre kerül a *logfile*ban a fenntebb leírt módon.

Inputról érkező sorozat

Ilyenkor a program nem interaktív módon működik, hanem végrehajtja a kapott sorozatot. Ez a megoldásmód azért sem szerencsés, mert szemmel követhetetlenné teszi az esetlegesen fellépő hibák keresését, hiszen nincs idő beavatkozásra és gépszerűen végrehajtja az utasításokat.

Amire számítunk

Tetszőleges bemenetnél a legalapvetőbb a program indulása. Nemcsak, hogy működjön, de hozza létre a megfelelő objektumokat a szkeleton szekvencia - diagramjának megfelelően. A program indítása aktivizálja a Main_program osztályt (`Main_program.start_up()`). Utána a Gfx indítása és a menük megjelenítése (`Main_program.menu()`).
A nevek beállítása után (`Main_program.set_players()` , `Game.play(<név1>, <név2>)`) az inicializálási rész következik a Billiardtable és a Scoreboard osztályon (`Scoreboard.init()`, `Billiardtable.init()`). Ekkor jönnek létre a konstansok – a falak és lyukak- illetve a golyók is. Mindegyiknek megtörténik az inicializálása. Beállítódik az eredményjelzőn a játékosok neve (`Scoreboard.set_names(<név1>, <név2>)`). Felrajzoljuk a golyókat a helyükre (`Ball.set_position()`, `Ball.set_color()`). Miután mindent kirajzoltunk elindulhat a játék. A dákó mozgatását a Gfx osztály rajzolja ki a képernyőre, itt változtathatunk kedvünk szerint a dákó beállításain. Befejezve lökjünk egyet (`Billiartable.hit_cue_ball()`). Az asztalon mozgások fognak történni, emlyeket lenaplózunk (`Billiardtable.motions()`). Ütközések vizsgálatánál figyeljük a mozgást (`Ball.crash(<mivel ütközhet>)`, `Ball.set_position()`, `Ball.set_velocity()`, `Ball.draw()`). A Ball::draw() vituális függvény, akárcsak az összes többi, ezért a rajzolást teljes egészében a Gfx osztály végzi a saját metódusaival. Az ütközés ellenőrzését minden lehetséges elemme elvégzi és módósítja azok adatait, ha szükséges. Lyukkal történő ütközés esetén a golyó eltűnik az asztalról és többé nem kerül elő, persze itt is vannak kivételek. Miután minden mozgás befejeződött az eredményjelzőn megjelenik az új állás. (`Scoreboard.set_data(<szükséges adatok>)`).
Ha nem pottyant semmi, akkor a kövekező játékos jöhet (`Scoreboard.set_next_player(<ki>, <hányszor>)`). És az egész kezdődik a lökésbeállítással előről.
Kilépéskor a Gfx.shut_down() hívása lezárja a grafikus felületet, ami a legutolsó lépés.

A prototípus

Fordítási útmutatás

A program készítése során azt az irányelvet követtük, hogy ameddig csak lehetséges megpróbáljuk fenntartani a viszonylagos platformfüggetlenséget. A platformfüggetlenség a prototípusnál még fenntartható, később azonban már elkerülhetetlen a specifikus kód írása. A specifikus kódnak Borland C++ 3.1–ben fordíthatónak kell lennie. Ezt szemelõtt tartva a prototípus készítése során is már a Borland C++ megvalósítások irányába mozdultunk el. Így például a modulok összeláncolását Borland C++ project file segítségével valósítottuk meg, mindazonáltal elkészíthetõ a programhoz Unix-os makefile melynek segítségével cc-vel is fordíthatóvá válik.

A program modulokból épül fel. Minden egyes osztály egy külön modulban helyezkedik el. Az osztályok deklarációit header-file-okban (*.hpp) a definícióit C++ file-okban (*.cpp) helyeztük el. Az egyes modulok között a kapcsolatot a header file-ok tartják, az összes modult a project file fogja össze egy egésszé.

A fordítás elsõ lépéseként Borland C++ 3.1–ben meg kell nyitni a mellékelt project file-t (Magicb.prj). Ebben definiálva vannak a program moduljai, így az összeláncolásról az integrált fejlesztõkörnyezet gondoskodik. A mi feladatunk mindössze a megfelelõ környezet biztosítása. Ez pedig a következõ lépésekbõl áll:

- Állitsuk a fordítást large model-re: Options/Compiler/Code generation/Model
- Állítsuk be a megfelelo könyvtárakat: Options/Directories Fontos, hogy a Source directory megfelelõen legyen beállítva, ugyanis a fordító itt fogja keresni a header-file-okat. Mindazonáltal a többi útvonal aktualizálása is elengedhetetlen a megfelelõ fordításhoz (pl. fontos, hogy a library útvonal is helyes legyen, mert a fordító itt keresi a könyvtári file-okat, melyek többek között például a large-modell-hez is kellenek).
- Állítsuk be a C++ értelmezést: Options/Compiler/Source/Borland C++
- Állitsuk be a C++ fordítást: Options/Compiler/C++ options/Use C++ compiler/ C++ always.
- Ezt követõen hozzáláthatunk a modulok befordításához. Ehhez válasszuk ki a Compile menü Build all parancsát. Ez elvégzi a fordítást.
- A program kiprobálásához nyomjon Ctrl+F9-et.

A prototípus header file-jai

header file neve	mérete	utolsó módosítás
Ball.hpp	5028	Apr 24 13:29
Btable.hpp	5658	Apr 24 13:29
Cue_ball.hpp	2887	Apr 24 13:29
Element.hpp	755	Apr 24 13:29
Game.hpp	3402	Apr 24 13:29
Gfx.hpp	7655	Apr 24 13:29
Hole.hpp	3005	Apr 24 13:29
Main_pr.hpp	3656	Apr 24 13:29
Scboard.hpp	6673	Apr 24 13:29
Use_mode.hpp	340	Apr 24 13:29
Vectors.hpp	4487	Apr 24 13:29
Wall.hpp	3901	Apr 24 13:29

A prototípus moduljai

modul neve	mérete	utolsó módosítás
Applicat.cpp	932	Apr 24 13:29
Ball.cpp	8220	Apr 24 13:29
Btable.cpp	13270	Apr 24 13:29
Cue_ball.cpp	4410	Apr 24 13:29
Element.cpp	4439	Apr 24 13:29
Game.cpp	6540	Apr 24 13:29
Gfx.cpp	10975	Apr 24 13:29
Hole.cpp	3409	Apr 24 13:29
Main_pr.cpp	4424	Apr 24 13:29
Scboard.cpp	7698	Apr 24 13:29
Vectors.cpp	8780	Apr 24 13:29
Wall.cpp	7880	Apr 24 13:29

Értékelés

A prototípus készítése során jelentkeztek az első olyan problémák, amelyekkel a csapatnak meg kellett küzdenie. Most derült fény arra, hogy az olyan gondosan megtervezett modelünk bővelkedik lyukakban, amelyeknek befoltozása ennek a két hétnek a feladata volt.
Ezen hiányosságok között említeném meg például a játék megnyerésére illetve elvesztésére vonatkozó kritériumokat, illetve ezek konkrét megoldási javaslatát. A kód megírása során többször kellett változtatni az osztályok interfészén akár láthatósági (paraméterátadási), akár hatékonysági okokból - nem említve továbbá az objektumon belüli kohézió növelését.
A prototípus implementálása során az alábbi mósodításokat tarottuk szükségesnek:

Billiardtable osztály:

Attrubútumok:

`where_to_put`	-	azt adja meg, hogy ha a legutóbbi golyó lett volna az utolsó színes golyó, akkor a következő körben hová kellene eltenni a feketét
`elements_on_table`	-	`Element *` -ból álló tömb lista helyett

Metódusok:

`int where_to()`	-	A `where_to_put` értékét adja vissza
`int motions()`	-	`void motions` helyett, visszatérési értéke megadja, hogy milyen golyóval ütközött először

Scoreboard osztály

Attribútumok:

`current_player`	-	`next_player` helyett a `current_player`-t érdemes tárolni
`last_holes[2]`	-	azt tárolja, hogy adott játékosnak melyik lyukba kellene löknie a fekete golyót, ha az előbbi lökése alkalmával tette volna el az utlsó színeset.

Metódusok:

`int get_current_player()`	-	`get_next_player` helyett
`int get_last_hole (int)`	-	az alábbi metódusokkal kellett a last_holes kezelése miatt kiegészíteni
`void set_last_hole (int, int)`		

Ball osztály

Metódusok:

`double get_radius(void)`	-	Be kellett vezetnünk, mivel a radius ismeretére szükség van az ütközés vizsgálatánál (crash metódus), és ez egy private mező.

Element osztály

Metódusok:

`double get_crash_coeff(void)`	-	Itt 2 új metódus bevezetésére volt szükség, az előző pontban említett okok miatt. Az ütközéshez szükség van az ütközési együtthatóra, ez azonban private mező, így szükség van a megfelelő accessor és modifier metódusokra.
`void set_crash_coeff(double)`		

Továbbá, amint az már a fordítási útmutatóban említésre került, a prototípus még platformfüggetlen konstrukció, azonban már lépéseket tettünk a Borland C++ 3.1 irányába, mivel a grafikus verzió úgyis mindenképpen platformfüggő lesz. Ez a prototípusnál oly módon jelentkezik, hogy lehetőséget biztosítunk egy, a platformfüggetlen megoldáshoz képest kényelmesebb (azonban BC3.1 függő) kód generálására is. Ez egy makróval szabályozható a use_mode.hpp header file-ban.

A program kétféle módon fordítható. Egyik lehetőség a platformfüggetlen, csak standard könyvtári függvényeket használó, egyben kötegelt feldolgozást lehetővé tevő (pl. gépesített tesztelés) verzió, a másik az interaktív, kényelmesebb, azonban csak Borland C++-szal kompatíbilis verzió. A tesztelés megfelelő szakaszához (kézi-, gépi-tesztelés) a megfelelő fordítás választható ki. Ehhez mindössze a use_mode.hpp-ben definiált makro állítása szükséges.

Tesztelés

A tesztelési munkákat két lépésben végeztük: egyrészt kézzel (ezt nevezhetjük úgymond a szemantikai debuggolásnak is), másrészt pedig az előre megtervezett tesztelőprogram segítségével.

Kézi tesztelés

Kézi tesztelést Erdei és Sója végezte. Ez tulajdonképpen még a debuggolás kategóriájába tartozik, A szemmel látható, igen feltűnő hibákat kerestük. Ide tartoznak pl. a program elszállásával járó hibák feldeeítése: tipikus példa erre a tömbből való kicímzés illetve a dinamikusan (nem) foglalt memóriaterületek használata.
Feltűnő hiba volt továbbá az, hogy ha egyszer egy golyó pottyant, akkor attól a pillanattól kezdve mindig pottyant. Ennek oka az volt, hogy egy tömbben gyűjtöttük az adott körven pottnanó golyók indexeit, de a tömböt nem töröltük a ciklus végén vagy a következő ciklus elején.
Szintén komoly fejtörés okozott a nem teljesen lefedett állapottérből származó hibák felderítése.

Az ellenőrzéseket nem pontos számolásokkal ellenőriztük, hiszen ez a gépi tesztelő (Vásárhelyi) feladata, és kézzel utánaszámolni komuly munkát és nagyon sok időt igényelne. Ezért karakteres képernyőn az egy lökés utáni statikus állapotot figyeltük: ellenőriztük a Scoreboard objektum megfelelő mezőiben szereplő adatokat (hiszen ezek majd a grafikus felületen megjelennek, de karakteres képernyőn nem, mert így is nagyon sok adat “áramlik” át a keprnyőn.)

Természetesen az igazán látványos tesztelés majd a grafikus felület használtával kezdődhet el. Hiszen a konkrét adatok ellenőrzésének nem sok értelme van, hiszen ugyanolyan algoritmus alapján számolnánk ki az adott sebességeket és helyvektorokat, amelyre gyakorlatilag ugyanazt a programot használnánk.

A kézi tesztelés során megelégedtünk azzal, hogy ellenőriztük a kritikus változókban szereplő adatok értékét, és ha nem stimmelt, akkor felderítettük a hibát.

Grafikus felület illesztése (architektúrális leírás)

A grafikus felület illesztése volt a szoftver megtervezésénél előálló egyik legkeményebb feladat. Végül is egy olyan architektúrát dolgoztunk ki, melyben a grafika teljes egészében le van választva a program egyéb részeiről. Ez úgy értendő, hogy külön objektumként kerül megvalósításra, (funkcionálisan) függetlenül minden más, a programban előforduló objektumtól – azok semmiféle kiviteli funkciót nem látnak el önmagukban. Bármiféle „kirajzolásra", „adat-megjelenítésre" kerüljön is a sor, az a grafikus felületet megtestesítő objektum valamely metódusának meghívásával kell az történjen.

Ennek a kialakításnak hátránya az olykor feleslegesnek tűnő bonyolultság és a metódushívások miatti időkiesés. Komoly előnye azonban ezzel szemben, hogy a kód átláthatóbb, a futtatható állomány mérete kisebb és a grafika a későbbiek során egyszerűen kicserélhető „a program alatt" anélkül, hogy azon akár a legcsekélyebb változtatást kellene végeznünk. Más szavakkal élve a grafikus felület a program egyéb részeinek szemszögéből átlátszó…

További fontos tervezői döntés volt, hogy a számolások mindenkori hibátlansága érdekében logikai koordinátákat használunk. Ez annyit jelent, hogy a program számoláshoz egy jelen esetben kicsinyített világban él, kicsinyített koordinátákkal számol és csak a tényleges rajzoláskor váltunk léptéket. Ez két szempontból is fontos mozzanat… Egyrészt a mindenkori számítások részeredményeinek is bele kell férniük a számoláshoz használt tárolási típus ábrázolási tartományába, másrészről pedig a grafika ezirányúlag is szabaddá válik, az adott felbontásnak megfelelően, rugalmasan, egyetlen konstanst megváltoztatva méretezhetjük kirajzoláskor a dolgokat…

Mint említettük, egyetlen grafikus objektum van. Az kezel le minden megjelenítésre vonatkozó kérést. Kapcsolata az alkalmazói rendszerrel pontszerű. A főprogram indításkor inicializáltatja a grafikát, bezáráskor pedig „bezárja" a grafikus felületet. Minden objektum (asztal, golyók, lyukak, falak, kijelző) ha rákerül a sor megjelenítés terén, a grafikus objektumhoz fordul, ami elvégzi a szükséges lépéseket. A grafikus objektum alapját képező osztály leírása:

A Gfx osztály struktúrája

Alaposztály(ok):

(nincsenek)

Példány(ok):

1 példány globális változóként

Komponens(ek):

(nincsenek)

Változó(k):

`screen_width`	egész szám	a képernyő szélessége
`screen_height`	egész szám	a képernyő magassága
`color_depth`	egész szám	a grafikus mód színmélysége
`number_of_pages`	egész szám	a grafikus lapok száma
`text_type_array`	egész szám tömb	a használt betűk típusai
`text_size_array`	egész szám tömb	a használt betűk mérete
`active_page`	egész szám	az aktuális grafikus lap
`saved_bitmap`	pointer	az elmentett grafikus terület
`rate_of_billiardtable`	valós szám	a biliárdasztal rajzolási együtthatója
`rate_of_scoreboard`	valós szám	a kijelző rajzolási együtthatója
`length_of_cue`	valós szám	a dákó hossza

Felelősség(ek):

```
void  start_up()
void  shut_down()
int   get_width()
int   get_height()
int   get_color_depth()
int   get_number_of_pages()
void  swap_active_page()
void  set_text_type(int, int)
int   get_text_type(int)
void  set_text_size(int, int)
int   get_text_size(int)
void  get_names(char *, char *, int)
void  winner(char *)
void  draw_ball(Ball *)
void  draw_wall(Wall *)
void  draw_hole(Hole *)
void  draw_cue(double, double, double, double)
void  draw_scoreboard(Scoreboard *)
void  draw_table(Billiardtable *)
void  draw_menu(char *, char *, char *, int)
```

Sok tulajdonság van, ami feleslegesnek tűnik, de esetleges grafikus mód-váltáskor jól jöhetnek. Az attribútumok talán csak az utolsó négy szorul magyarázatra. A `saved_bitmap` és a `length_of_cue` a dákó kirajzolásához kellenek. Az előbbi a dákó alatti terület átmeneti mentésének kezdőcíme a memóriában, az utóbbi a dákó kirajzolandó hossza. A másik két tulajdonság, a `rate_of_biiliardtable` és a `rate_of_scoreboard` az asztal illetve a kijelző kirajzolás előtti logikai-fizikai koordináta-transzformációhoz szükségesek. Azt, hogy mit is jelentenek a logikai koordináták, fentebb tárgyaltuk…

A metódusokat egyenként leírjuk az általuk megvalósítandó funkciókkal és azok algoritmusával együtt…

`int start_up()`

A grafikus motor elindításáért és az alaptulajdonságok beállításáért felelős. Algoritmusa egyszerű. Megpróbálja inicializálni a Borland C BGI grafikus motorját és ha sikerrel jár, akkor beállítja a kellő képernyőméret, színmélység, stb. értékeket. Ha nem sikerül, hibaüzenetet ad és hibakóddal visszatér a hívóhoz…

`void shut_down()`

A grafikus motor „lelövését” végzi lényegében egyetlen lépésben…

`int get_width()`

Visszatérési értékében megadja az aktuális grafikus mód által kezelt képernyő szélességét…

`int get_height()`

Visszatérési értékében megadja az aktuális grafikus mód által kezelt képernyő magasságát…

`int get_color_depth()`

Visszatérési értékében megadja az aktuális grafikus mód által kezelt színmélységet…

`int get_number_of_pages()`

Visszatérési értékében megadja az aktuális grafikus mód grafikus lapjainak a számát…

`void swap_active_page()`

Vált grafikus lapot (megjeleníti azt, ami eddig a háttérben volt. A rajzoláshoz használt lapot is beállítja – a háttérbe kerülőre. Ezt a metódust az asztal használja és csak akkor, ha mindent kirajzoltatott, amit akart…

`void set_test_type(int number_of_type, int type)`

A használt betűtípusok tömbjében állítja be a megadott sorszámú betűtípust a megadott típusra…

`int get_text_type(int number_of_type)`

A megadott sorszámú típust adja vissza a betűtípusok tömbjéből…

`void set_test_size(int number_of_size, int type)`

A használt betűméretek tömbjében állítja be a megadott sorszámú betűméretet a megadott méretre…

`int get_text_size(int number_of_size)`

A megadott sorszámú méretet adja vissza a betűméretek tömbjéből…

`void get_names(char *player_1, char *player_2, int max_length)`

Bekér két nevet a felhasználótól és azokat a két paraméterben (*player_1*, *player_2*) adja vissza. A nevek maximális hosszát a *max_length* paraméterben a hívó definiálja…

`void winner(char *player)`

A megadott játékos győzelmét „hirdeti ki” valamely módon (rajzzal, hangjátékkal, stb.)…

`void draw_ball (Ball *ball)`

A paraméterben megadott golyót rajzolja ki a képernyőre körként. Természetesen a felbontás miatti torzítást is figyelembe véve. A kirajzoláskor a golyó szín-tulajdonságából kell képeznünk a teli-csíkos rajzolási tulajdonságot (a már korábban specifikált kódolás segítségével ez megtehető)…

`void draw_wall (Wall *wall)`

A paraméterben megadott falat rajzolja ki a képernyőre. Felhasználva annak *shape* tulajdonságát, két menetben rajzolja ki. Elsőként kirajzolja a farészt (külső rész), majd a posztóval borított csíkot (belső rész)…

`void draw_hole(Hole *hole)`

A paraméterben megadott lyukat rajzolja ki a képernyőre körként. Természetesen a felbontás miatti torzítást is figyelembe véve…

```
void draw_cue (double old_angle, double new_angle,
               double old_strength, double new_strength)
```

Kirajzolja a dákót a megadott erőt és irányt jelezve. A régi erő és irány adatok az előzőleg elmentett kép visszahelyezésének paramétereit kiszámolandó kellenek. Először kiszámítjuk, hova is kell visszatenni a mentett képet, majd visszatesszük. Ezután kiszámítjuk, hogy honnan kell mentenünk és elmentjük a kellő képterületet. Végül kirajzoljuk a dákót a kellő irányba és távolságba (a fehér golyóhoz viszonyítva)….

`void draw_scoreboard(Scoreboard *scoreboard)`

Kirajzolja a kijelzőt a képernyőre. Paraméterben megkapja a kijelző objektumot, így le tudja kérdezni a kirajzolásához szükséges adatokat (nevek, golyótípusok, stb.) A kirajzolás az előző kijelző-rajz felülírásával történik. Tehát lényegében minden egyes kirajzoláskor a kijelző egészét rajzoljuk újra. Ez persze időpocsékolásnak tűnhet, de az sem tellene kevesebb időbe, ha kiszámolnánk, mely részeket kell törölni, mielőtt az új adatokat megjelenítjük…

`void draw_table(Billiardtable *billiardtable)`

A biliárdasztal kirajzolásáért felel. Paraméterben megkapja a hívótól az asztalt megtestesítő objektumot, hogy annak kellő tulajdonságait lekérdezhesse tőle. Csak az asztal színe és alakja kell a kirajzoláshoz. A mindenkori korrekt megjelenéshez az asztalnak először magát, majd a falakat és lyukakat végül pedig a golyókat kell kirajzoltatnia…

```
void draw_menu(char *item_1, char *item_2, char *item_3,
               int selection)
```

Kirajzolja a menüt, amely három nyomógombból áll. Az egyes gombok neveit paraméterként meg adja a hívó, valamint azt is közli, hogy melyik gombon áll éppen a felhasználó. Ez a kirajzoláskori megkülönböztetéshez szükséges adat. A kirajzoláskor a teljes képernyőt újrarajzoljuk (nem visz el túl sok időt és itt az időnek vajmi kevés jelentősége van még) – a hátteret és a gombokat is (ilyen sorrendben)…

„Végleges” grafikus felület

A tényleges grafikát a Borland C++ 3.1 beépített *graphics* könyvtárának függvényeivel valósítjuk meg. A grafikus mód, amit használni fogunk a VGA vezérlő VGAMED 640x350x16 módja. Azért döntöttünk emellett, mert 2 grafikus lappal rendelkezik, ami a megjelenítés villogásmentességének biztosításához szükséges. A grafikus felületet alapjaiban három különféle, önálló felülettel lehet jellemezni. Ezek rendre az alap-menü, a nevek bekérésének felülete és a játék felülete (asztal, kijelző). Ebben a sorrendben a felületek a következők…

Az alap-menü felülete némi háttérképet (rajta információt a programról – név, készítők nevei, stb.) és három térhatású nyomógombot tartalmaz. Ez a három gomb a „Játék”, „Alkotók…”, „Kilépés” neveket viseli majd. A gombok közötti mozgást a kurzormozgató billentyűk teszik lehetővé. Ha a felhasználó nem akar lépkedni a három, majdan eltérő színnel megjelenő kezdőbetűt használva azonnal aktiválhatja a gombok bármelyikét. Azt a gombot, amelyen a felhasználó éppen áll, megnövelt betűmérettel tesszük könnyedén megkülönböztethetővé a többitől. A kiválasztását, aktiválását az ENTER billentyű leütésével teheti meg. A programból való kilépésre adott továbbá az ESC billentyű is.

A „Játék” gomb aktiválásával érhetjük el a nevek bekérését, mivelhogy a játék kezdetén be kell kérjük a játékosok nevét. Itt a felület lényegében két térhatású beviteli mezőt és két – szintén térhatású – nyomógombot tartalmaz. tartalmaz, amelyekbe a felhasználó a játékosok neveit írhatja be (egyes számban említjük a felhasználót, mert a program számára kevéssé érdekes, hogy hány felhasználóval is van dolga). A két gomb az „OK” és a „Vissza” neveket viselik. A két mező között a TAB billentyűvel lehet váltani. A két gombot pedig az ENTER és az ESC billentyűkkel lehet elérni. Addig nem mehet tovább a felhasználó, míg valamelyik mező üres…

Magának a játéknak a felülete alapjaiban két részre bomlik. Az egyik rész az asztal képe, a pillanatnyi állásnak megfelelően. A másik a kijelző.

Az asztal a képernyő felső részét fogja beteríteni. Hosszabbik oldalai vízszintesen helyezkednek majd el, alkalmazkodva a felbontáshoz. A falakat a posztótól megkülönböztetett (például barna) színnel, szélükön posztószínű (például zöld) csíkkal ábrázoljuk az asztal négy oldalán. A asztal belsejét posztósínű terület jelzi majd rajta két fehér kereszttel, amik a fehér golyó felhelyezési helyeit mutatják majd. A lyukak sötét (például fekete) körként lesznek ábrázolva, a golyók pedig szintén köralakban, de már a nekik megfelelő színben jelennek meg.

A kijelző a képernyő megmaradt, alsó felét kitöltő téglalapban jelenik meg. Térhatású lesz, mind a kijelző, mind a felirathalmaz rajta.

Szükséges változtatások

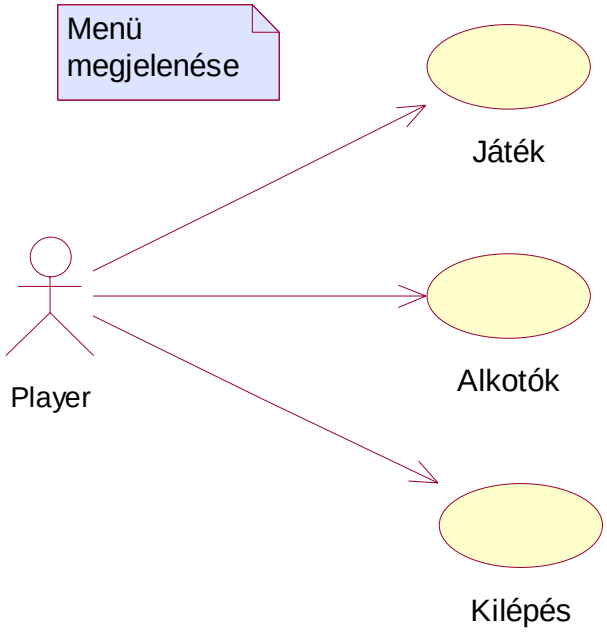
A kódban szükségessé vált változások oroszlánrésze a grafikus objektum alapját képező Gfx osztály metódusaiban található. Ezekre értelemszerűen szükség van, hiszen innen kezeljük a kimenetet, amit most grafikussá szándékozunk tenni.

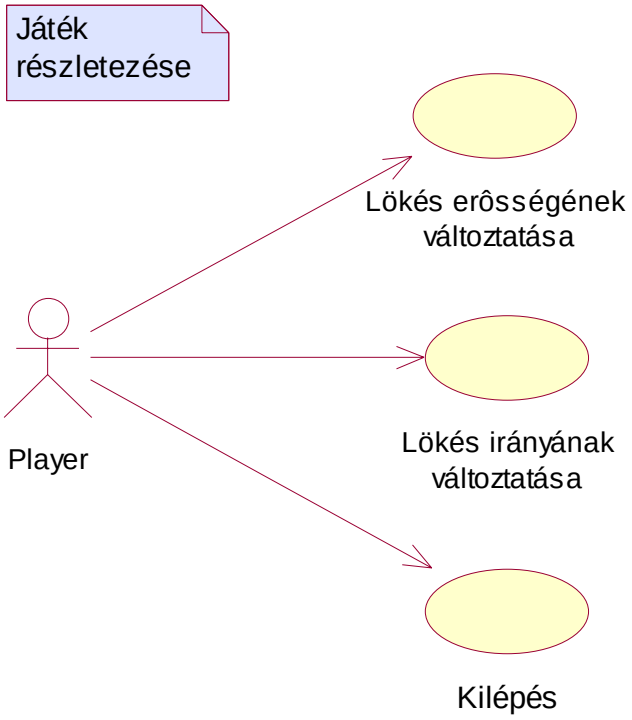
Minden egyéb változás a tesztelés nyomainak eltűntetése céljából szükséges. Ez kétféle lépést jelent Az egyik az, hogy a tesztkimenet előállításához használt `sprintf` illetve `gfx.print_out` hívásokat töröljük.

A másik módosítás több magyarázatot igényel. A kirajzolás „élethűségét” elősegítendő nem csak a koordináták logikaiak, amiket használunk, hanem a golyók egyszeri elmozdulása is az. Ez azért lényeges, mert máskülönben, túl nagy lépésenként vizsgálva az ütközéseket, elsiklanánk olyan esetek felett, amikor „éppen csak hogy” ütközni kellett volna két golyónak. Ez nem megengedhető hiba, ezért egy lépésben csak a valódi lépésmennyiség törtrészét teszik csak meg a golyók az egyes vizsgálatok között. A tesztelés során minden egyes logikai lépés után „kirajzoltuk” a golyókat (kiírtuk azok adatait). Erre nyilván a végleges verzióban se szükség, se kellő erőforrás nincsen (mert ez nem csak felesleges, de lassú is lenne). Emiatt módosul egy kissé a `Billiardtable` osztály `motions()` metódusa.

Más érdemi módosításra a grafikus felület miatt nincs szükség (és a tesztelés miatt sem kell mást megváltoztatnunk)…

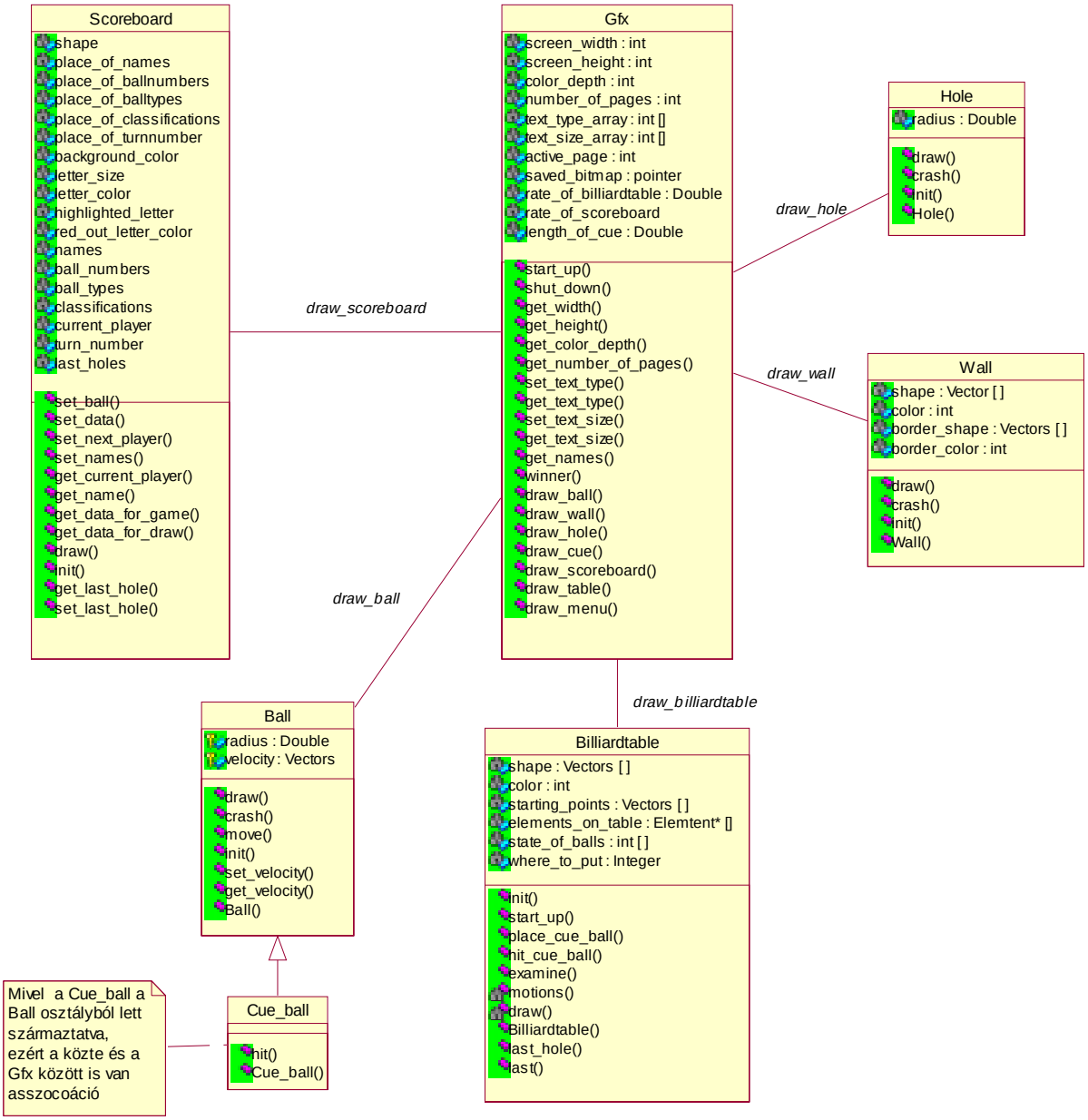
A Grafikához Kapcsolódó Use Case Diagramok





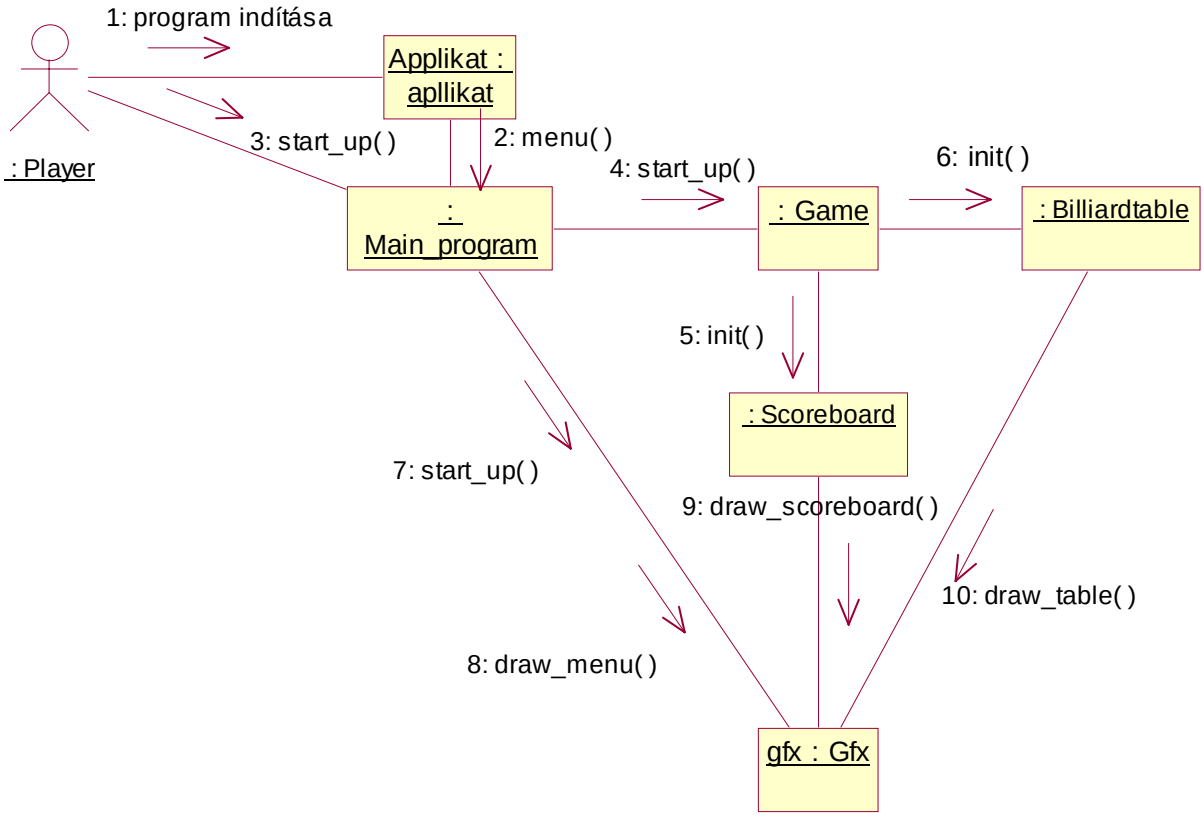
Statikus Struktúra Diagramok

Osztálydiagram a Gfx szempontjából



Ezen az ábrán nincs feltűntetve az összes objektum, csak a Gfx-szel való asszociációkat jelenítettük meg. Az összes objektum kapcsolata az analízis modellben szerepel, viszont ott még a Gfx nincs részletezve.

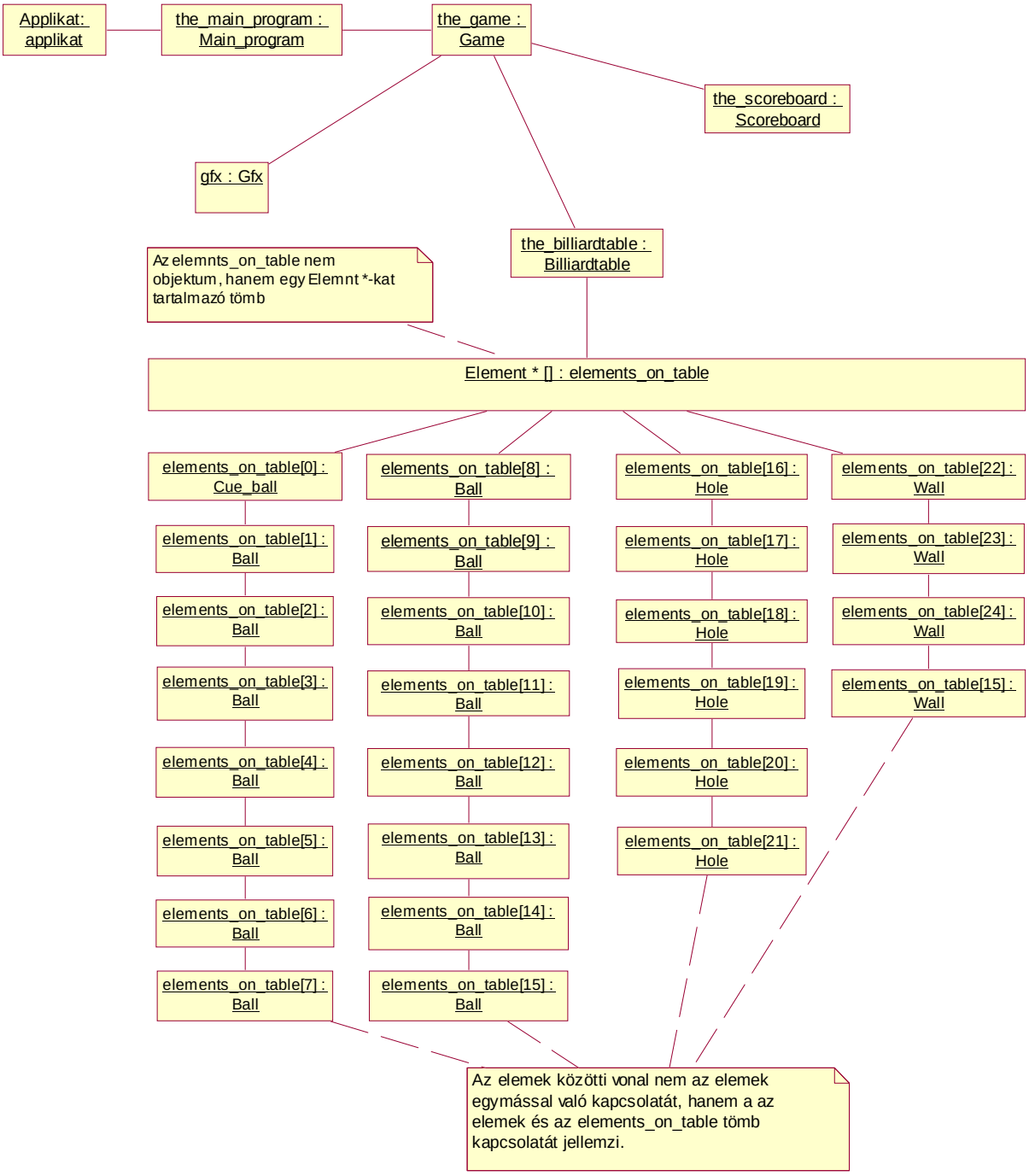
Kollaborációs Diagram az Inicializáláshoz



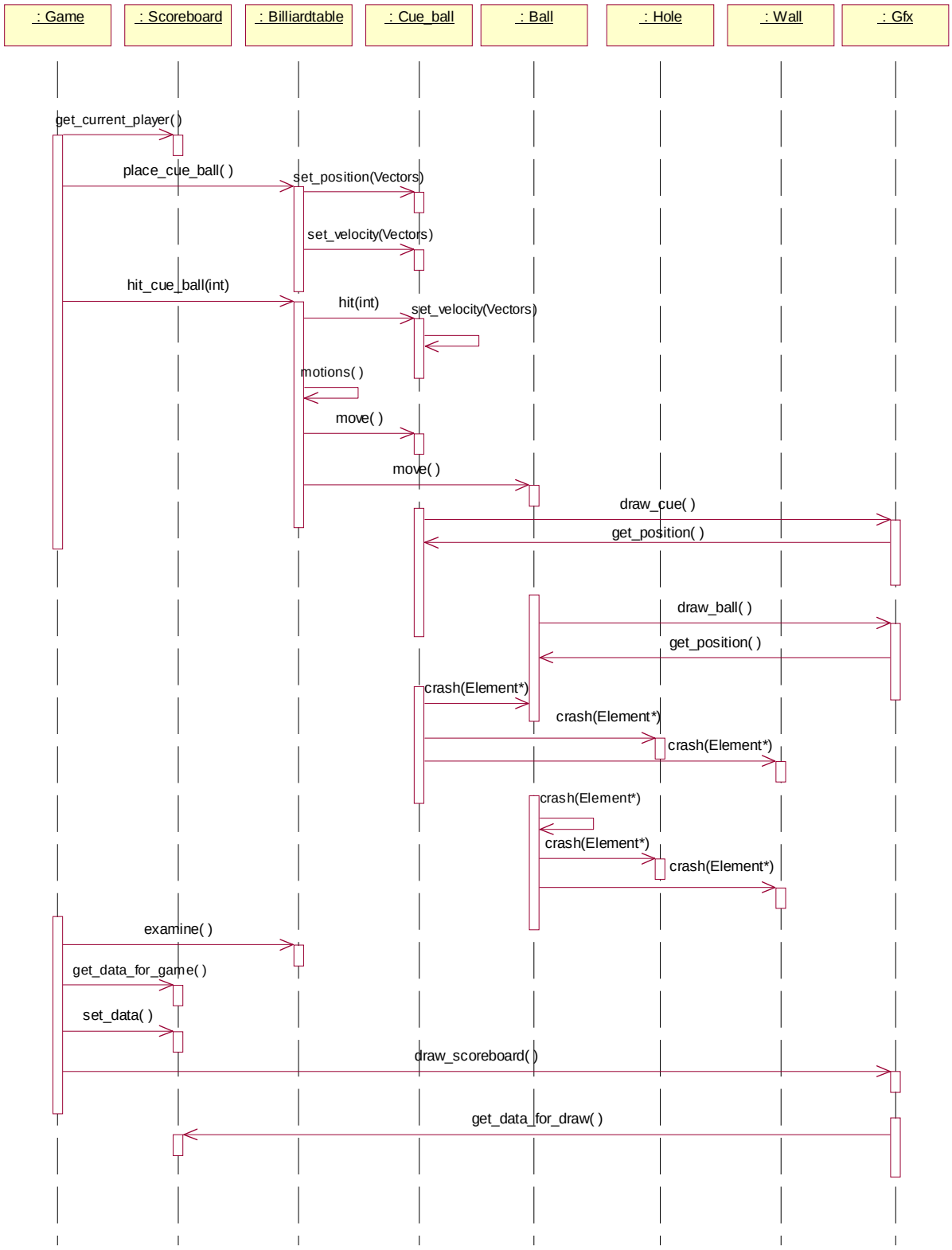
Az inicializálási folyamatot mutatja. Az Element alaptípusú objektumok ezen a diagramon nem szerepelnek, mert (ez az analízis modell óta, az implementáció alatt megváltozott) ezek nem itt inicializálódank, hiszen ha valaki elindítja a játékot pusztán azért, hogy kilépjen belőle vagy, hogy esetleg az alkótók adatait tekintse meg, felesleges annyi memóriát lefoglalni.

Tervezzük a kezdeti menüt kiegészítését a játék fájlból való betöltésével. Így, ha két játékos a játékot abbahagyni késznyszerül, az állást elmenthetik, majd később az elmentett állást betöltve ugyanonnan folytathatják. Ekkor természetesen másfajta inicializálásra van szükség.

A teljes program objektumdiagramja

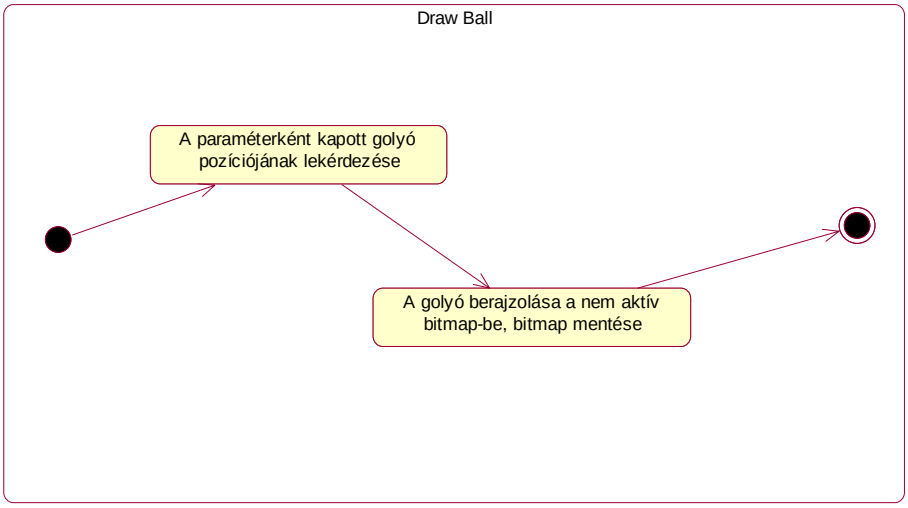


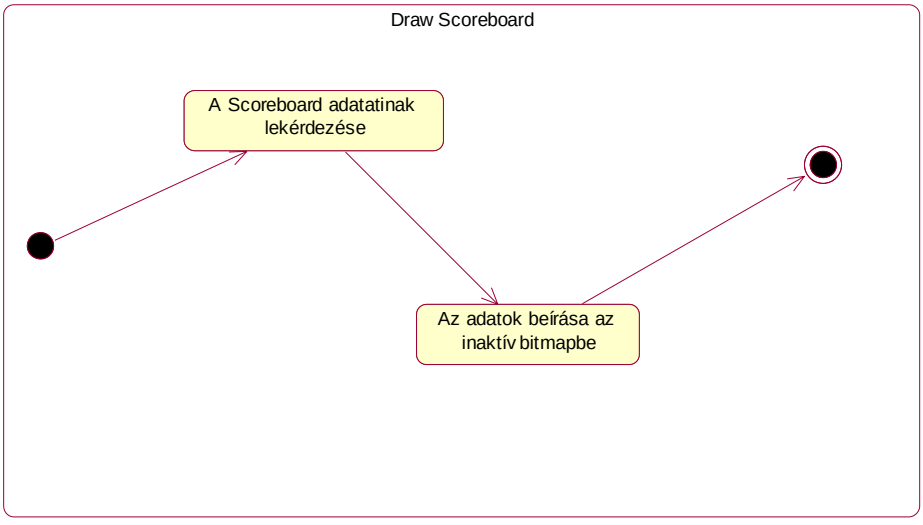
Szekvenia Diagram Fehér Golyó 1 Lökésére



A golyókból, lyukakból és falakból az egyszerűség kedvéért egy példány van, amikor a golyó saját maga crash() metódusát hívja meg, az azt jelenti, hogy az összes többi golyónak meghívaj a crash függvényét (a sajátját természetesen nem). A Cue_ball azért hívja meg maga a set_velocity metódusát, mert ő felelős a lökés erősségének beállításáért.

Statechart Diagramok
(Golyó és Scoreboard Rajzolására)





A végleges program

Fordítási útmutatás

A program készítése során azt az irányelvet követtük, hogy ameddig csak lehetséges megpróbáljuk fenntartani a viszonylagos platformfüggetlenséget. A platformfüggetlenség a prototípusnál még fenntartható volt, most azonban már elkerülhetetlen a specifikus kód írása. A specifikus kódnak Borland C++ 3.1–ben fordíthatónak kell lennie. Ezt szemelõtt tartva a prototípus készítése során is már a Borland C++ megvalósítások irányába mozdultunk el. Így például a modulok összeláncolását Borland C++ project file segítségével valósítottuk meg, mindazonáltal elkészíthetõ a programhoz Unix-os makefile melynek segítségével cc-vel is fordíthatóvá válik. A programhoz A Borland C++ BGI grafikai modulját használtuk.

A program modulokból épül fel. Minden egyes osztály egy külön modulban helyezkedik el. Az osztályok deklarációit header-file-okban (*.hpp) a definícióit C++ file-okban (*.cpp) helyeztük el. Az egyes modulok között a kapcsolatot a header file-ok tartják, az összes modult a project file fogja össze egy egésszé.

A fordítás elsõ lépéseként Borland C++ 3.1–ben meg kell nyitni a mellékelt project file-t (Magicb.prj). Ebben definiálva vannak a program moduljai, így az összeláncolásról az integrált fejlesztõkörnyezet gondoskodik. A mi feladatunk mindössze a megfelelõ környezet biztosítása. Ez pedig a következõ lépésekbõl áll:

- Állitsuk a fordítást large model-re: Options/Compiler/Code generation/Model
- Állítsuk be a megfelelő könyvtárakat: Options/Directories Fontos, hogy a Source directory megfelelően legyen beállítva, ugyanis a fordító itt fogja keresni a header-file-okat. Mindazonáltal a többi útvonal aktualizálása is elengedhetetlen a megfelelõ fordításhoz (pl. fontos, hogy a library útvonal is helyes legyen, mert a fordító itt keresi a könyvtári file-okat, melyek többek között például a large-modell-hez is kellenek).
- Állítsuk be a C++ értelmezést: Options/Compiler/Source/Borland C++
- Állitsuk be a C++ fordítást: Options/Compiler/C++ options/Use C++ compiler/ C++ always.
- Ezt követõen hozzáláthatunk a modulok befordításához. Ehhez válasszuk ki a Compile menü Build all parancsát. Ez elvégzi a fordítást.
- A program kiprobálásához nyomjon Ctrl+F9-et.

A végleges program header file-jai

header file neve	mérete[byte]	utolsó módosítás
Ball.hpp	5021	Máj 13 23:27
Btable.hpp	6000	Máj 14 12:32
Cue_ball.hpp	2943	Máj 13 23:14
Element.hpp	786	Máj 12 20:08
Game.hpp	3469	Ápr 23 06:33
Gfx.hpp	9581	Máj 12 11:36
Hole.hpp	3093	Máj 12 14:30
Main_pr.hpp	3730	Ápr 24 13:02
Scboard.hpp	6968	Máj 09 22:22
Use_mode.hpp	351	Máj 03 19:38
Vectors.hpp	4568	Ápr 23 06:34
Wall.hpp	4050	Máj 10 21:35

A végleges program osztály moduljai

modul neve	mérete	utolsó módosítás
Applicat.cpp	960	Ápr 24 13:20
Ball.cpp	8842	Máj 13 23:19
Btable.cpp	11866	Máj 14 12:32
Cue_ball.cpp	4917	Máj 14 12:32
Element.cpp	4524	Ápr 24 05:43
Game.cpp	6540	Máj 13 23:21
Gfx.cpp	28281	Máj 14 12:10
Hole.cpp	3591	Máj 12 03:10
Main_pr.cpp	5449	Máj 10 12:23
Scboard.cpp	8091	Máj 14 12:19
Vectors.cpp	8985	Ápr 23 06:34
Wall.cpp	9019	Máj 13 22:51

A végleges program egyéb moduljai

modul neve	mérete	utolsó módosítás
Bold.chr	14670	1992.06.10. 03:10
Egavga.bgi	5554	1992.06.10. 03:10
Euro.chr	8439	1992.06.10. 03:10
Goth.chr	18063	1992.06.10. 03:10
Lcom.chr	12083	1992.06.10. 03:10
Litt.cpp	5131	1992.06.10. 03:10
Magicb.dsk	2547	2000.05.13. 07:28
Magicb.prj	8169	2000.05.13. 07:28
Sans.chr	13596	1992.06.10. 03:10
Scri.chr	10987	1992.06.10. 03:10
Simp.chr	8437	1992.06.10. 03:10
Test_prg.cpp	21443	2000.05.02. 00:47
Text.prg	19353	2000.04.25. 23:22
Trip.chr	16677	1992.06.10. 03:10
Tscr.chr	17355	1992.06.10. 03:10

A Grafikus Felület Fejlesztése Közben Szükséges Változtatások

A grafikus felület illesztése közben alapvetően három okból kellett változtatásokat végrehajtani. Az egyik ok a PROTO-nál – a specifikált kimenet elérése érdekében – alkalmazott megoldások eltűntetésének szükségessége, a másik az előre – sajnálatos módon – nem kellően átgondolt lekérdezések megalkotásának elkerülhetetlen mivolta illetve néhány végeredményben szükségtelennek ítélt részlet törlésének elhatározása volt. A harmadik típusú problémát az okozta, hogy néhány számolt alapkoordináták sajnos kisebb hibával számoltunk…
Az első típusú problémák kiküszöbölése csupán kivágást jelentett, ami első megközelítésben a megfelelő DEFINE-nal megkreált makrók törlését avagy módosítását, végső soron pedig a tényleges kivágásukat jelentette. A második eset már kissé nehézkesebb, ugyanis itt mind az osztály deklarációját, mind a definícióját módosítani kellett. A szükséges újítások:

Új metódusok és azok rövid leírása

Billiardtable

`get_shape`	- az asztal alakjának kiadása
`get_starting point`	- a két kezdőpont kiadása
`get_color`	- szín lekérdezése
`last`	- az utolsó lement golyó helyének lekérdezése

Gfx

`clear`	- képernyő törlése
`swap_active_page`	- grafikus lapcsere

Hole

`get_radius`	- a rugár kiadása

Wall

`get_shape`	- alak kiadása

A módosítások zöme a *Gfx* osztályban következett be, mivel sok olyan elemet helyeztünk el benne eredetileg, amik ugyan az általános grafikus felület megteremtéséhez elengedhetetlenek, de most mégis úgy döntöttünk, hogy megszabadulunk tőlük, ne zavarják a kód érthetőségét jelenlétükkel. Ezek a változások a *text_type_array* és a *text_size_array* változók valamint (értelem szerűen) a hozzájuk tartozó beállító illetve lekérdező függvények, illetve a képernyő alapparamétereinek (magasság, szélesség, színmélység) kiadására szolgáló metódusok törlésében nyilvánultak meg. Ezeken kívül még kellett apróbb módosításokat eszközölni a *Cue_ball* osztályban, mivel a szögváltoztatásokhoz rendelt gombok fel voltak cserélve. A *Billiardtable*-ben, mert a hozzá tartozó header-file-ban található DEFINE-al létrehozott koordináták sántítottak

egy picit, és végül a *Main_program*-ban a menükezelés finomítása érdekében. A harmadik típusú találkozásokat már az imént megemlítettem, ezért azokat nem sorolnám fel megint. Bizonyos problémák már látszottak a PROTO elkészítésekor is, de akkor még csak gyanúsnak tűntek – hibás mivoltukra csak most derült félreérthetetlen módon fény…

Értékelés

A félévi munka során több-kevesebb nehézséggel kellett szembenéznünk, és ezeken a nehézségeken úgy-ahogy túltettük magunkat. A problémák zöme - természetesen - tervezési és szervezési nehézségek voltak.

Tervezés

Amikor az analízis modellt kidolgoztuk, még úgy gondoltuk, hogy felesleges ennyi diagramot rajzolgatni, meg tervezgetni, inkabb kezdjünk el kódolni, hogy minél hamarabb készen legyünk a programmal. Az analízis modellt egy csapattag dolgozta ki, és akkor úgy tűnt, hogy nemhogy részletes és tökéletes, de talán túlságosan is az. Aztán az idő előrehaladtával egyre több ptobléma derült ki.
A Szkeleton beadásakor (és a szekvencia diagramok felvételekor) derült ki, hogy amit olyan szépen láttunk, hogy hogyan történik, az csak nagyon t'volról olyan szép és jo. Még 3 nappal a beadás előtt nem tudtuk (kiv've az analízis modell kidolgozóját), hogy melyik metódus melyik oszt'lay melyik metódusát hívja, még az inicializálás szekvenciadiagramja sem volt teljesen tiszta Konzultálva az analízis modell kidolgozójával, úrrá tudtunk lenni ezen a problémán.
A modellben lévő lyukak, a következő feladat alkalmával, a protoipus elkészítésekor kerültek napvilágra. Kiderült, hogy a modell nem tartalmazza a játék megnyerésének illetve elvesztésének adminisztr'lásához szükséges attribútumokat é metódusokat, ezenkívül más hiányosságokra is fény derült. Végül egy 14 órás maratoni programozás, debuggolás és tesztelé után sikerült egy működő prototípus verziót időre elkészíteni.

Szervezés

Szervezési gondok között elsősorban a kommunikácós problémákat említenénk. Mivel ez egy kis csapat volt, kis munka, kis juttatásokkal (fizetés), ezért az értekezleteket és a feladatkiadásokat nem vettük olyan komolyan, mint ahogy ez szoftverfejlesztéssel foglalkozó cégeknél szokás. Jellemzőek voltak például az "én azt hittem, hogy te azt mondtad", meg az "én úgy emlékszem, hogy azt beszéltük meg" típusú mondatok. Mindezek abból adódtak, hogy a feladatokat nem írásban adtuk ki, amit annak, aki elvállalja alá kell írni. Így, maikor a mukáját készen visszahozza, az általa aláírt ískerződésben leírtak alapján lehet minősiteni a munkáját. Kis csapat lévén, de főleg azért, mert nem munkatársa, hanem barátok vagyunk, ezek a félreértések nem vezettek áthatolhatatlan akadályokhoz, de még így is előfordult, hogy néhány napig morosak voltunk. Erre a toleranciára persze nem csak barátok között lenne szükség, de láthat'-legalábbis mi azt tapasztaltuk ezalattt a félév alatt -, hogy még barátok között is néhétz ezt a toleranciát gyakorolni, így aztán egy nagyobb munka során, amelynek esetleg sokkal nagyobb tétje is van, hiába végezzük az barátokkal együtt, mégis meg kell hagyni bizonyos formalitást, méghozzá azt, hogy írásban kell a követelményeket és a munka leadásokat intézni.