# Szundi Szglab4-es tanácsai

Verzió: 1.17 – 2007-04-20_01

## Tartalomjegyzék

Ez a dokumentum (remélhetőleg) letöltető innen:

http://www.inf.bme.hu/~szundi/

# Fő gondolatok

## Be kell járni a konzira

Bármily fájdalmas is. Ugyanis különben honnan is tudnád, mit kell csinálni?!

## Hát az Info-site-on rajta van egy csomó minta!

Az info-site-os doksik sajnos nem jók. Igazából még ha azok is (nem nagyon néztem őket), akkor is mindenki a rosszat adja be, amit azokból ki lehet nyerni. Ha 7 csoportom van, akkor ebből 5 ugyanazt a marhaságot járja körbe valahogy, és ezt tuti szedik valahonnan. Az info-site-os doksikat **NE HASZNÁLD**! Az a fő, hogy TE alkosd meg, amit csinálsz, hiszen csak ezzel tanulsz, másolással nem. (főleg ezeknek a másolásával :) )

## De akkor hogyan lehet ezt megcsinálni, ha még minta sincs, amiből kiinduljak?!

Hát, meglepő, de egyszerűbb rossz minta nélkül jót alkotni, mint avval. :) A lényeg, hogy figyelj oda, mit pampog a konzulensed, **jegyzeteld le, mert elfelejted**, és akkor semmi gond nem lesz. Irányelvek lesznek, amiket majd mond, és egyik konzulens sem kegyetlen (na jó, csak kevesen) úgyhogy nem fog neked rossz pontot adni. Ha igen, vitatkozz, de készülj fel, hogy lesz válasza. Persze ha nem érted, miért kaptál rossz pontot, akkor hogyan is tanulhatnál az egészből? Kérdezd meg.

## Konzulens

Nem minden konzulens egyforma. Arra figyelj, amit ő mond el, és annak megfelelően dolgozz. Annyira nem lesz más, mint amit a másik mond, tehát nyugodtan beszélhetsz olyan csapatokkal is, akiknek más, jobbnak vélt konzulensük van.

## Az egész célja

Nem az az egész félév célja, hogy programot írj, hanem:

- hogy **megtervezz**, és megírj egy programot pár **más emberrel egy csapatban**, és
- hogy megtanuld **nagyvállalati szinten ledokumentálni**.

Ebből következik, hogy elvárnak majd olyan dolgokat is tőled, ami egy ilyen kis játéknál felesleges(nek tűnhet) számodra. Persze nem lesz majd igazad, nagy valószínűséggel, és ezt vedd tudomásul. :) Ez nem erről a játékról szól, hanem arról, hogy meg kell tanulnod másokkal és a főnökeiddel együtt dolgozni, és olyan programszerkezetet (modellt) alkotni, ami megállná a helyét akkor is, ha történetesen 10-100x ekkora programként fognánk fel (amilyenekkel majd remélhetőleg dolgozni fogsz).

## CVS, SVN

Az e-mailben egymásnak küldjük az új verziókat / floppy / usb-toll módszerek nagyon veszélyesek. Könnyen felülírhatjátok egymás munkáit, elveszhetnek verziók, szóval kész agyrém. Ja, és nehézkes is :)

Ha jót akartok, irány a sourceforge.net, és csináljatok egy project-et. Jár hozzá CVS támogatás is. A CVS (Concurrent Version System) egy olyan rendszer, ami tipikusan szöveges fájlokon tesz lehetővé párhuzamos munkát adatvesztés nélkül! Mindenkinek melegen tudom ajánlani. Kiváló windows-os kliens a TortoiseCVS, de az Eclipse még kiválóbb Java-s fejlesztőkörnyezet (és gondolom a többi is) beépített CVS pluginnel kényezteti használóit!

Mostanság tör utat magának elődje mellett az SVN (Subversion). Ajánlott windows-os kliens a TortoiseSVN. Nem kívánok sokat írni ezekről, ott a google. SVN plugin is letölthető az eclipsehez subclipse néven.

Ne feledjétek, CVS tárhelyet biztosít számotokra a sourceforge.net.

# Követelmény, projekt, funkcionalitás

## Követelmény definíció

Ez egy leírás arról, hogy mi a projekt célja, milyen lesz a felhasználói felület (pár mondatban), milyen rendszeren fog majd futni a program, és milyen minőségi követelményeket fogtok kielégíteni a fejlesztése közben.

## Projekt terv

Leírja a projekt kivitelezése közbeni nagy lépéseket (modell, szkeleton, proto, kész program), hogy ezekre miért van szükség, és hogyan fogjátok ezelet elkészíteni.

## Feladat magyar nyelvű leírása

Újra kell fogalmazni a feladatot. Láthatjátok, hogy a feladatkiírások minden félévben (direkt) kellőképp homályosak, akárcsak a való életben. :)

A rész lényege, hogy pontosítsátok a feladatot, és valamivel részletesebben leírjátok, de szakmai kifejezések mellőzésével, közérthető módon!

## Szótár

A dokumentációk során használt szavak jelentését előre le kell fektetni, hogy minden projektben résztvevő egyértelműen tudja olvasni, írni a dokumentumokat. Pl. játékos lehet gép előtt ülő ember, aki nyomkodja a gombokat, ill. a figura, akit irányít a játékban. A szótár az ilyen problémákat küszöböli ki.

## Essential use-case-ek

A program legalapvetőbb használati lehetőségei. Pl. egy mozipénztár jegykezelő programjának essential use-case-ei lehetnek: jegyet lefoglal,

lefoglalást megszüntet, jegyet elad. Ellenben nem use-case, hogy kilép a programból! Egy atomerőmű vezérlőprogramjának leállítása persze már use-case, hiszen azt szabályzott módon kell elvégezni.

A use-case diagram megrajzolása egy Actor figura lesz mondjuk játékos megnevezéssel, és a use-case-ek hozzákötve.

# Analízis modell

**EZ A LEGFONTOSABB AZ EGÉSZBEN! MINDENNEK AZ ALAPJA!! NEM PÁR ÓRÁS MUNKA, AMIBŐL JÓ MODELL LESZ.**

## Mi is az a modell?

A rendszereden belül az a komponens, ami azzal foglalkozik, ami magának a problémának a megoldása. Ebbe nem tartozik bele, hogy ezt hogy jeleníted meg (Menü, grafika, highscores <- ez mind ide nem kell!), vagy hogyan vezérled (MouseClick, keyevent <- ezek sem kellenek!). Ebbe az tartozik bele, hogy a valóságot reprezentáló osztályaid hogyan fognak együttműködni.

Az MVC-ről pár szó (nagyon alap):

http://en.wikipedia.org/wiki/Model_View_Controller

## Init szekvencia diagramok

Nem elég, hogy ha csak a <<create>> nyilakat rajzolod be! Az egész init lényege, hogy létrejönnek az objektumok **és** azok egymással kapcsolatba kerülnek! hogy ki miként kapcsolja össze őket milyen metódusokkal, az is lényeg! ( Valamiért mindenki csak <<create>>-eket csinál!)

## Szekvencia diagramok

Nem elég egy grandiózus, mindenre kiterjedő szekvencia diagram! Bármily hihetetlen, de a szekvencia diagramok *szekvenciákat* írnak le. Tehát pl. ha egy objektum megváltoztatja az irányát, akkor a 10 mp-el későbbi ütközés vagy akármilyen esemény **NEM** ebben a szekvencia diagramban fog megjelenni! Azért nem, mert az majd később fog történni **más esemény hatására**, így az nem ennek a szekvenciának a része. Ha nincs legalább 5 szekvencia diagramod egy tipikus szglab4-es feladatnál, akkor lusta vagy!

## Koordináták

Ez nem általános érvényű, de majdnem biztos, hogy igaz lesz a te programodra, amit itt az egyetemen adnak neked feladatként: a koordinátákat, ha lehet, nem a pályán rohangáló dolgokat reprezentáló objektumokba kell rakni. így is meg lehet oldani a dolgot, de máshogy még jobban. Általában van valami objektum, amin rohangálnak a szereplők, pl. "pálya" v. "űr", és általában az lesz a jó megoldás, hogy ha ez tartalmazza a többi objektumot, és az asszociáció attribútuma a koordináta. Az ütköztetés és az objektumok konkrét elmozdítását az őket tartalmazó objektum végzi az egyes objektumok kéréseinek megfelelően. Pl. van nekik irányvektoruk (irány+sebesség), és az alapján. Az ütközési esetekről értesíti az objektumokat (behívja az összeütköztél(ezzel) metódust pl.), aztán azok eldöntik, élnek/halnak-e.

Azt, hogy a koordináták az asszociáció attribútumai legyenek, legegyszerűbben és legtisztábban egy un. wrapper osztály segítségével oldhatjuk meg. Ennek lesz pár koordinátája stb., ami a térreprezentációval kapcsolatos, és ez fogja tárolni a térben szaladgáló objektumokat, bármilyenek is legyenek azok. A térnek így wrapperjei lesznek, és azoknak lesznek objektumaik. így bármilyen objektum lehet a térben, anélkül, hogy a tér milyenségéről tudna.

### Miért jó ez?

Azért, mert elválasztod vele az objektumok működését a világ tér-reprezentációjától. Nem feltétlenül kell az objektumnak tudnia a környezet tulajdonságairól. Így egy 2d-s játék könnyedén 3d-s játékká alakítható, csak az tároló osztályt kell átírni 3d-s valamivé pl. Persze ez csak akkor lehetséges, ha az objektumoknak nem kell tudniuk a világ dimenzióiról, nem használják a reprezentációt. Pl. így nem ők mondják meg, milyen irányba akarnak menni, ők csak azt mondhatják meg, hogy melyik objektum felé, és akkor a tároló objektum ad nekik vissza egy objektumot, ami az tároló felé az ő irányukat reprezentálja, vagy ilyesmi.

**Mikor nem jó ez?**

Látni kell persze, hogy sok esetben ez nem elegendő, ha az objektum saját maga akar tájékozódni, akkor már nem nagyon lehet megkerülni, hogy ismerje a reprezentációt, de meglepően sokszor nincs erre szükség. Továbbá vannak feladatok, amikor ez már értelmetlen, pl. folyamatos terű 3d-s űrhajós játékoknál, biliárd játéknál, ahol már a teljesítményen is sok múlik, itt egyszerűbb, és célravezetőbb, ha az objektumok maguk tárolják a koordinátájukat.

## Isten-objektumok

Kerüljük az olyan objektumokat, amik túl sok mindent csinálnak, és Manager, Controller vagy hasonló nevük van. Képzeljük el, a valóság hogy működik, és ha ebbe nem illik bele, amit csináltál, akkor valami nem stimmel! Ha esetleg nem értenéd, miről írok, képzeld el, vajon az Anyukád irányítja-e a mozdulataidat az iskolában, vagy akárhol. Hát, nem. Pedig sok olyan modellt láttam már, ahol mindenféle objektumok (Controller-ek) irányítgattak más objektumokat, azok meg zombiként léteztek. Nem biztos, hogy ezt az utat kellene járni.

## State-chartok

Figyeljünk oda, hogy az 1 állapotos statechart értelmetlen. Ebből viszont nem következik, hogy pl. a "Felszed", "Ütközik" vagy "Meghal" dolgok állapotok lennének! Nem kell baromságokat gyártani, csak hogy legyen 3 statechartod! Előfordulhat, hogy pl. nem is kell, ha egyszer nincs, csak mindennek 1 állapota. Ez van. (konzulenssel konzultáljatok!)

# Szkeleton

## Mi az a szkeleton

A szkeleton mint szó vázat jelent. Ez a modelledből készülő program váza lesz. Egy olyan kódot kell előállítani, ami gyakorlatilag semmit nem csinál, csak ennek segítségével a szekvencia és kollaborációs diagramokon felvázolt objektum-kommunikációkat (metódushívásokat stb.) végrehajtjuk, melynek során a kód nem csinál semmit, csak log-okat (pl. képernyőre írás) produkál, ami alapján a hívások sorrendje megállapítható. Ez annak ellenőrzésére szolgál, hogy a modell egyáltalán alapjaiban megfelel-e a célnak, meg lehet-e valósítani azokat a diagramokat, amiket felvázoltál a dokumentációban. Majd meglátjátok, hogy sok helyen már most hibára fogtok bukkanni!

## A szkeleton nem proto!

A szkeletonban megvalósított osztályoknak **nincs belső állapota**, belső változói, és **nincs** semmilyen **algoritmus implementálva**! Azt majd a proto-ban kell. Nem algoritmikusokat ellenőrzünk, csak a modell alapvető hibáit szűrjük ki.

## Mit kell tudni egy szkeletonnak?

Kitaláljátok, hogy mik a fontos tesztesetek, amiket le kell futtatni. Ez mondjuk nem lesz nehéz, mert (amennyiben rendesen dolgoztál) ezek az Analízis modell-ben beadott szekvenciadiagramokat fogják végrehajtani, hiszen pont ezek az érdekes részek, amiket a tesztekkel le kell fedni. Magyarul, a szkeleton eljátssza, hogy ezek a tesztesetek megtörténnek, és itt már kiderülhet pár olyan dolog, amire a modellalkotás során nem gondoltatok, és ha valami alapvető probléma van vele (kifelejtettetek valamit pl.), akkor az itt ki fog bukni, ugyanis a teszt lényege az, hogy a szkeleton logjai alapján ellenőrizzétek, a program tényleg képes a modell alapján működni. Magyarul, ha a szekvencia-diagrammokon és kollaborációs diagrammokon szereplő sorrendben jelennek meg a szkeleton kimenetén az objektumok üzenetei,

akkor minden rendben, ha nem, akkor lehet javítani vagy szkeletont vagy a modellt. Ne feledjétek az init szekvenciadiagramot sem (ami nem csak <<create>>-ekből áll, ugye!), ez az egyik legfontosabb része a rendszernek, ugyanis ez építi fel a modellt.

## Szkeleton valóságos use-case-ei

Először is az init use-case. Egyik use-case sem futhat le az init nélkül (tehát minden más use-case <<includes>> init use-case). A többi use-case-t úgy határozzátok meg, hogy azok lefedjék a szekvencia és kollaborációs diagramjaitokon is szereplő fontos modell-működéseket! (tipikis use-case-ek tehát: init, tárgyfelvétel/eldobás, ütközés, valaki teremt valakit stb.)

## Hogy készül egy tipikus szkeleton kódja?

1. Meg kell írni a modell osztályok vázát, az üres metódusokat létre kell hozni.
2. Ezekbe az üres metódusokba annyi kód kerül, hogy mindegyik metódus kiírja, hogy őt behívták (pl. "Urhajo: utkozes()")
3. Lesznek metódusok, ahol majd a futó program döntéseket fog hozni, ezeken a helyeken kiírunk egy kérdést, és a felhasználó válasza alapján fog a kód dönteni, merre tovább. Nem szabad objektumállapot v. belső változók alapján dönteni!
4. Egy új osztály (pl. Szkeleton) létrehozása egy public static void main metódussal-el, ami kiír egy menüt, hogy a doksiban szereplő melyik tesztesetet szeretnénk futtatni. Itt el lehet dönteni, hogy a fentebb említett kérdezgetős módszerrel járhatjuk be az összes tesztesetet kevesebb menüponttal, vagy a menüpontok előre meghatározzák a kérdésekre a választ, és akkor azokat fel sem kell tenni mondjuk. Mindkét megoldás jó. Amikor a felhasználó kiválasztotta a menüt, akkor kiírkálják a metódusok, hogy behívták őket, és egy pár kérdés kivételével lefut a progam, majd kilép, vagy megjelenik újra a menü.
5. Init use-case-t futtató metódus implementálása
6. Többi use-case-t futtató metódus implementálása, amelyek első

sora tipikusan az init use-case-t futtató metódus

## Architektúra, ütemezés

Itt körülbelül arról kell beszélni, hogy használtok-e szálakat, azok szinkronizációját mi biztosítja, milyen sorrendben és miként kezeli a rendszer az objektumait, azok interakcióit stb., és hogy ez miért jó.

Kell csinálni egy olyan ábrát is, amin az objektumok (konkrét példányok) találhatók, és minden objektumreferenciát jelöljetek egy nyilacskával. Magyarul, aki tud egy másik objektumról, az nyíllal mutat rá. Ebben az a jó, hogy a class-diagrammal ellentétben (ami statikus struktúrát mutat be) egy menet közbeni, objektumok kapcsolatairól szóló felállást mutat be.

# Proto

## Mi a proto célja

Képzeld el, hogy egy 50 fős projektben veszel részt, és 30 megás lesz a forráskód. Megvannak a tervek, hogy fog működni a rendszer, és az 50 programozó nekiáll, leprogramozza az egészet. Nyilván kell tesztelni a dolgokat, és mivel tudjuk, a hiba akkor javítható még ki olcsón, ha mindjárt a rendszerbe való bevitelekor meg is találjuk. Minél később találjuk meg, annál drágább lesz a kijavítása (mind időben, mind munkában, mind pénzben, talán presztízsveszteséget is okoz).

Aki már írt programot, bizonyára próbálta azt a módszert, hogy elindítja a programot, használja egy darabig, és ha minden jól működött, akkor valószínűleg jó is alapon késznek lett nyilvánítva. Azt senkinek nem kell elmagyarázni - gondolom -, hogy ez a módszer eleve nem lehet 100%-os, de tegyük fel, hogy az, tegyük fel, hogy mindenki tökéletesen ki tudja tesztelni fél óra (fél hónap) alatt a programját (a 30 mega forrásos rendszerét). Ekkor azonban az ügyfél kér pár új funkciót, módosításokat bevisszük, és lám, a program új verziójában olyan hibák is előkerülnek, amiket az előző verzió tesztelése során már kizártunk, hiszen akkor jó volt!

Hogy lehet ez? Bekövetkezhet-e ilyen? Mi az oka?

Igen! Sőt, nagyon valószínű, hogy be is fog! A legjobban megtervezett rendszereknél is előfordulhat! Minden program úgy épül fel, hogy egyes részei (komponensei) használnak más komponenseket is, nyilván. De hogy ha a használt komponenst megváltoztatjuk, az azt javító programozó nem tudhatja, hogy az ő komponensét ki, milyen módon használta. Igazából majdnem mindegy, hogy a dokumentáció szerint használták, vagy nem, az a lényeg, hogy előfordulhat, hogy az ezt használó komponensek ezután nem fognak helyesen működni az általuk használt komponens apró belső változásai miatt. Azt pedig tudjuk, ami előfordulhat, elő is fordul.

Most már mindannyian értjük, miért nem helyes az a módszer, hogy futtatom a programot, és ezzel tesztelem. Még a fejlesztés során is az utolsó

pillanatban felfedezett hiba javítása is okozhat újabb hibákat, minden alkalommal kezdhetnénk előről az egészet! A kézzel tesztelésnél ezt nem fogja senki megtenni, még ha kötelező is. Unalmas.

A megoldás az, hogy a programunk komponensei számára teszteseteket dolgozunk ki. Olyan programkódot írunk, ami egy tipikus helyzet elé állítja a komponenst, olyan helyzetek elé, amikkel a rendszer működése során fog találkozni, vagy találkozhat. Magyarul, írunk egy kódot, ami egy adott tesztet futtat, ezáltal kipróbálja a komponens egy funkcióját, vagy egy esetet rajta. Ezt hívjuk **teszteset**nek.

Az 50 fős vállalatnál a programozók a komponensük implementálásának végén feltöltik a Version Control System-re (pl. CVS, SVN vagy Arch vagy másra) a kódot, majd a rendszer éjjel automatikusan az implementálás előtt már elkészített teszteseteket futtatja, és másnap, amikor a programozó bejön, már emailt kapott az elkövetett hibáiról az automata rendszertől. Láthatja, hogy melyik tesztesete futott le hibásan, és meg is kapta a be és kimeneti fájlokat ilyesmiket, nekiállhat kitalálni, mi a baj.

A **ti feladatotok** az szglab4 kurzus 2. részének keretében a modell implementálása, és hozzá a tesztesetek kidolgozása, és implementálása, hogy hasonlóan jól biztosítsátok kódotok hibátlanságát.

## Hogy készül a proto

1. Kitaláljátok, hogy milyen teszteseteket fogtok kidolgozni. Lehet lesni a szkeleton use-case-eknél pl. ötletekért. A lényeg, hogy a tesztesetek ideális esetben fedjék a teljes forráskód minden végrehajtási lehetőségét. Írjátok le a teszteset nevét, és lényegét, leírását.
2. Meghatározzátok a tesztesetek bemenetét (a teszt végrehajtásához milyen objektumok hogyan kellenek, és a lépések, amiket ezekkel végrehajt a teszt)
3. Nagyon fontos, hogy leírjátok, mi az, amit vártok, hogyan lehet felismerni, hogy a kimenete a tesztnek sikerre utal-e, vagy hibára, ha hibára, akkor ezek várhatóan mik lesznek, hol kell majd javítani, ha így v. úgy történt.

4. Megtervezitek, hogy a tesztelőprogramotok a bemenetét milyen formában kapja meg, általános leírást készítetek erről. A kimenetről hasonlóképp.
5. Átnézitek az analízis modelleteket (amit már kijavítottatok persze a szkeleton tapasztalatai alapján), és meghatározzátok az egyes osztályok bonyolultabb metódusainak algoritmusait. Ezeket le is dokumentáljátok.
6. Részletesen megtervezitek a teszteket. Parancsszinten megtervezitek a bemeneteiket, üzenetjelleggel a kimeneteket.
7. A kimenetek felismerésére algoritmusokat adtok, programokat is lehet írni, ezek működését, felépítését, általános rendszerét le kell írni, implementálni.
8. Implementáljátok a kimenetek ellenőrző programjait
9. Implementáljátok a modell-t
10. Implementáljátok a futtatókörnyetetet, ami a bemeneti file-ok alapján meghajtja a modellt. (proto)

## Milyen parancsokat használjunk a bemeneten

Sok protonál láttam olyan parancsokat, hogy pl. az X űrhajó felveszi az Y dolgot. Ez nem jó akkor, ha ennek magától kell megtörténnie, amikor az űrhajó rámegy! A proto már kész játék, aminek nincs grafikus felülete, tehát ilyen parancs nem lehet, csak olyan, hogy menj erre v. arra, lőj, ilyesmik. A gép által irányított objektumok pedig lépkednek, és cselekednek maguktól.

Jó parancs pl. az, hogy új pálya, elem/objektum felrakása/levétele, elemek attribútumainak állítgatása(pl. irány beállítása), elemen akció kiváltása (pl. lövés), egy időegység léptetése v. x idő lejátszása. Látható, hogy a parancsok fele a proto futásához való előkészítést, pályaépítést stb-t végzi, a másik fele meg a játékos nem létező input perifériáját helyettesíti, amivel lehet irányítani a játékot, befolyásolni az objektumot, és léptetgetni az időt.

## Hogy nézzen ki a kimenet

A proto kimenetének egy célja van: minél pontosabb képet adjon arról, mi történt a modell belsejében. Nagyon fontos látni, hogy ha túl sok üzenet

van, és azt senkinek nem lesz kedve megnézni, mert túl sok, akkor két választás van, vagy csökkenteni kell az üzenetmennyiséget, vagy programra kell bízni az elemzést (utóbbi természetesen kiváló ötlet!).

Tipp: Érdemes az ilyen loggolásra esetleg megtanulni egy loggoló lib használatát, pl. a log4j nagyon egyszerű, és kiváló! Szépen formázott logkimenetünk lesz pontos időértékekkel, jelzi, melyik szál loggolt, és a loggolás melyik java fájl hányadik sorában történt, szóval minden, ami kell.

## Hogyan tesztelünk?

Minden tesztesethez készen áll a bemenet, és a kimenetet ellenőrző program (rossz esetben ember). Ekkor minden tesztre elindítjuk a proto-t, majd megvizsgáljuk a kimenetet, ideális esetben a kimenetet ellenőrző program emberek számára értelmezhető kimenetét (pl. Ok, vagy Hiba). Érdemes csinálni egy scriptet, ami minden tesztet elindít, és hiba esetén leáll. vagy ilyesmi. Ekkor javítunk, majd előről. Ha nincs hiba, akkor reménykedünk, hogy a teszteseteink tényleg helyesen lettek kitalálva, és elmegyünk szabadságra.

## Mit kell beírni a doksiba?

1. Interfész definíció. Leírjátok, miként fog működni a ki/bemenet, formátum, ezek célja, miért, hogy stb.
2. Use-case-ek. Mit lehet csinálni a proto-val. Kissé nagyvonalú lenne azt mondani, hogy ezek a parancsok, amik a bemeneti file-ból elérhetők, de kb. fedi a valóságot a közelítés.
3. Tesztelési terv. Hogyan fogjátok a bemenetet megadni, és a kimenetből eldönteni, mi a futás eredménye. Meg kell adni, milyen tesztesetek lesznek. A kimenetet feldolgozó programokat is le kell írni, miként elemzik a kimenetet.
4. Objektumok, metódusok tervei. Meg kell adni az objektumok állapotgépeit, az osztályok metódusainak működését röviden el kell magyarázni, a nem triviális algoritmusú metódusokat diagrammokkal (activity), részletes magyarázatokkal is meg kell tűzdelni.

5. Tesztek részletes tervei. Az egyes tesztesetekhez meg kell adni a bemenetet, meg kell adni, hogy a tesztesen mely funkciók tesztelésére való. Meg kell adni, mit várunk a kimenetben, és milyen feltételezett hibák esetén mire következtetünk.
6. A tesztelést támogató programok tervei. Ezek hogyan épülnek fel, működnek stb.

## Tipikus hibák

1. Tesztesetek és use-case-ek összekeverése. Ez nagyon kellemetlen, innentől kezdve hülyeséget lehet csak gyártani...
2. Sokszor előfordul, hogy minden metódust a csapatok triviálisnak bélyegeznek, hogy ne kelljen az algoritmusokat végiggondolni és leírni, diagramokat rajzolni. Haha!
3. A tesztek részletes terveinél tényleg le kell írni a bemeneti adatokat!
4. Ne feledjétek, hogy ez egy működő játék, csak nincs grafikus felülete! Így nem szabad olyan parancsokat implementálni, hogy pl. x valaki felveszi y valakit, ha a szabályok szerint automatikusan felvenné, ha pl. rálépne. Ilyenkor ilyen parancs nincs, hanem a rálépésével a kész modell megoldja a felvételét.
5. Hihetetlen számmal fordulnak elő csapatok, akik nem érzik szükségét említeni, hogy döntik el, a teszteset hibásan vagy jól futott-e le. Az értelmét kérdőjelezi meg a tesztelésnek, ha nincs megadva a sikeresség feltétele.

Általában elmondható, hogy ezeket a csapatok jól ellustulják. Nem véletlenül érnek a proto-s doksik 1.5-2x annyit, mint az eddigiek!

# Grafikus

## Alapgondolatok

A proto befejeztével kaptunk egy implementált, alaposan letesztelt modellt. Elméletileg most már a modell-osztályaink kódjában nem is fogunk javítani. Persze az élet nem habostorta, és valami úgyse fog klappolni (ha igen, akkor gratula), dehát semmi gond, hiszen minden proto-módosítás után csak újrafuttatjuk a teszteseteket lejátszó bat/shell script file-t, majd megvárjuk a "minden ok!" üzenetet. Csodás! Akkor kezdjünk neki a grafikus rendszer tervezésének! :)

Azért dolgoztunk ennyit a modellen külön, hogy egy önálló komponensét adja a rendszernek, ami független a megjelenítéstől és a vezérléstől! Tehát ne függjön a modell működése attól, hogy most 3D-s grafikával, AWT, Swing, Web-page-es megjelenítés lesz-e belőle, vagy joystick, egér, netán billentyűzetes irányítással fogjuk vezérelni a játék vagy a program menetét. Ebből következik, hogy logikusan a **grafikus rendszer fogja használni a modellt, nem pedig a modellbe írjuk bele a grafikával, rajzzal, megjelenítéssel foglalkozó kódot!** A helyes eljárás, hogy írunk egy olyan programot, ami

1. a felhasználó kívánalmai alapján inicializálásra szólítja fel a modellünket,
2. a játék során azt lépteti (pl. behívja a modell vilag.lép() metódusát)
3. a felhasználó parancsait a modell "nyelvére" lefordítva annak átadja (pl. urhajo1.fordul(-30 fok)),
4. figyelemmel kíséri a modellben történő változásokat
5. valamint azokat megjeleníti.

Látható tehát, hogy a modell és a grafikus programunk gyakorlatilag elkülönülnek. Mint ahogy te is, amikor megtanulsz a TV távirányítóval bánni, nem kell levágni a kezed, és beültetni a távszabályzót a csuklódba. Elég, ha megfogod! :)

## Részletesebben

A fenti magasröptű maszlagot akkor fordítsuk le gyakorlati elemekre.

A modell kódját nem változtatjuk meg! Nincs *valamimodellobjektum*.draw(...) metódus, meg hasonlók! Nem a modell dönti el, mit rajzol, és hogyan! A grafikus rendszer ismeri a modell-osztályokat, és pl. egy jó megoldás lehet, ha a modellben van Urhajo, Reketa és BorgKocka osztályunk, amit majd meg kell jeleníteni, akkor mindegyikhez csinálunk egy _modellobjektum_Rajzolo osztályt. Pl. az UrhajoRajzolo osztályba bele lesz programozva, hogy kérdezze le az űrhajó paramétereit, és azok alapján a kapott grafikus felületre rajzolja azt rá. Egy ilyen osztály tehát mondjuk 1 metódust tartalmaz, ami így néz ki pl: `public void rajzol(Graphics g, Urhajo u) { ... }`. A ... pedig a kódot takarja, ami lekérdezgeti u tulajdonságait és g-re rárajzol ebből valamit.

Ez eddig a minimum, amit el kell követni, de menjünk még tovább. Ezzel ugyanis van még prár probléma:

1. Honnan tudja a rajzoló, hogy az egész pályán belülre hova kell rajzolni az űrhajót, és milyen irányba nézzen, amikor ezt nem feltétlen ül az űrhajó tudja magáról?
2. Most akkor minden egyes osztályra írjunk egy if-et, hogy `if (obj instanceof Urhajo) UrhajoRajzolo.draw(g, (Urhajo)obj);`? Mert ugye ez csak Urhajo-t tud rajzolni!

Az első pontra a megoldás lehet a következő pár trükk. A java Graphics osztály (amire rajzolunk), tud pár tök jó dolgot. Mielőtt behívjuk az űrhajó rajzolóját, azelőtt lemásoljuk a Graphics g objektum állapotát egy Graphics g2 objektumba. `Graphics g2=g.create();` Ezzel lett egy másik Graphics-unk, ami ugyanolyan, mint az eredeti, de ha változtatjuk a beállításait, az eredetié nem változik. Ekkor azt mondjuk, hogy `g2.translate(`*urhajo-x-koordinata*, *urhajo-y-koordinata); Ezzel a koordinátarendszer origóját rámozgattuk az űrhajóra, az űrhajó ekkor ezen a graphicson már a (0,0) pontbon van. Ezután elforgatjuk a koordinátarendszert annyival, amennyire az űrhajó el van fordulva:* `g2.rotate(_urhajo-fok);` Ezzel az egésszel azt értük el, hogy a

rajzolónak nem kell tudnia az űrhajó hollétéről és irányáról, ő csak kirajzolja úgy, hogy az űrhajó 0 fokban áll a (0,0) pontban, és mi pedig a g2-t adjuk neki rajzolófelületnek, amit így előre kipreparáltunk. Az űrhajó így jó helyre fog kerülni, majd a g2-t eldobjuk, java eltakarítja. Ezt minden objektumra ismételjük.

A második pont is érdekes. Ha úgy döntöttünk, hogy minden objektum maga tárolta a koordinátáit és irányát stb.-t, akkor mielőtt a rajzolóját meghívjuk, az előző bekezdés alapján lekérdezzük az adott objektumtól, hogy hol van, merre néz, kipreparáljuk a g2-t, majd behívjuk a megfelelő rajzolót. Hopp! Pont ez az, amit meg akarunk most oldani, hogy ne kelljen sok `instanceof` operátort használni, mert ha a programunk tele van ilyen módszerrel, akkor egy új osztály felvétele után tuti lesz egy csomó hely, ahova elfelejtjük az `instanceof` operátorokat felvenni pl. Tényleg lesz ilyen, mindenki mindent elfelejt, főleg 3 évvel a kód "befejezése" utáni javításkor! :)

Szóval csinálhatunk egy Map-et, ami nyilvántartja azt, hogy melyik osztályunkhoz milyen rajzoló kell. A Map egy olyan dolog, ami valami kulcshoz valamit hozzárendel, ezt már a java készen adja. pl. `Map rajzolok=new HashMap();` Ekkor van egy Mapünk, ami üres. Tanítsuk meg, melyik osztályhoz melyik rajzoló kell: `rajzolok.put(Urhajo.class,new UrhajoRajzolo());` Amit látnunk kell még, az az, hogy a rajzolóknak ezek után csak egységesen kezelt objektumokat tudunk átadni, mivel egységesen kezeltük a problémát, tehát az UrhajoRajzolo nem Urhajo-t fog kapni, hanem Object-et, és majd ő alakítja (castolja) Urhajo-va! így már képesek leszünk adni a rajzoloknak egy közös interfészt: `Interface Rajzolo { public void rajzol(Graphics g, Object obj); }`. Amikor az objektumunk megvan, amit ki szeretnénk rajzolni, akkor előkapjuk a rajzolo-ját `Rajzolo rajzolo=rajzolok.get(obj);`, majd rajzoltatunk: `rajzolo.rajzol(g2,obj);`. Csodás! Van egy rajzrendszerünk, amibe új X osztály felvételekor csak a következőket kell tenni:

1. Rajzolo interface-t implementáló XRajzolo osztály megvalósítása
2. Rajzolo-kat tároló Map-be az XRajzolo regisztrálása az X osztályhoz.

```
rajzolok.put(X.class, new XRajzolo());
```

Kész! Ezután minden megy magától, nincs hibalehetőség, nem kell több helyen módosítanunk a programot! :)

Megjegyezném, hogy ez a rendszer így pofonegyszerű, de van vele némi probléma. Ez a rendszer a modell-elemeken lépked végig, azokat kezeli, nem pedig azoknak a grafikus rendszerben megjelenő nézeteit kezeli. Ha a megjelenítés során nem ennyire egyszerűek a követelmények, pl. csak bizonyos elemeket szeretnék megjeleníteni, vagy egyéb csak megjelenítéssel kapcsolatos dolgokat kell figyelembe vennem egyes objektumokra másképpen, mint másokra, akkor gondban leszek, mivel ez a class alapú rajzolás nem tesz különbséget objektumok között.

Ezt a problémát úgy küszöbölhetjük ki, hogy a modellünk működése során keletkező eseményekre reagálva a grafikus rendszer a megjelenítendő modellobjektumokhoz létrehoz egy őket megjelenítő view objektumot. Ennek az objektumnak a feladata, hogy a modellt összekösse a grafikus rendszerrel, a modell eseményeit a grafikus rendszer számára értelmes eseményekké konvertálja. (Pl. a dobozt elmozdítja a futószalag, amin rajta van -> grafikus objektuma erről értesül, és a grafikus koordinátáit módosítja) A rajzolást ekkor ezen objektumok segítségével végezzük, így ezeknek már lehetnek grafikus rendszer számára értelmes paramétereik is (a modell objektumoknak nem lehetnek ilyenek, hiszen függetlenek mindenféle megjelenítési rétegtől).

## A játéktér kirajzolása Java-ban

### A java (J)Component rendszer elve

A felületelemek java-ban gui komponensekként vannak csoportosítva. Egy komponens felel azért, hogy benne lévő dolgok (pl. Panel esetén) és saját maga ki legyen rajzolva. Ez úgy történik, hogy a **java megkéri a komponenst, hogy rajzolja ki magát**, nem pedig a program x időnként kirajzolja magát! Eseményvezérelt modern grafikus rendszerekben ez az eljárás, minden más csak hackelés, és ellenjavallt!

## Csináljunk komponenst!

A programban érdemes egy saját komponensimplementációként megjeleníteni tehát a játékteret. pl:

```
class Jatekter extends JComponent { public void paint(Graphics g) { ... } }
```

Ha frissíteni szeretnénk a játéktereden a rajzot, a következő a teendőnk:

1. `jatekter.repaint();` Ezzel megkéred a java-t, hogy kérjen meg téged, hogy rajzoltasson ki. Ez a helyes! :)
2. egyszer csak behívódik a `jatekter.paint(...)` metódus. Ekkor rajzolsz ki mindent a kapott Graphics-ra. JComponent esetén nem kell duplabufferelni.

## Időzítés

Erre vannak tutorialok a neten, a lényeg, hogy egy `javax.swing.Timer` segítségével a `jatekter.repaint()` metódust hívogatjuk megadott időközönként, és minden rendben lesz.

Figyeljünk oda rá, hogy elvileg egyes windows és X rendszerek az egymásratolódott paint események közül lehet, hogy csak 1-et adnak át, így nem biztos, hogy egy mp. alatt az előre beállított számú képkocka fog megjelenni, ha épp belassul a gép egy picit valami miatt. Ez persze nem feltétlenül érdekel minket, de egyes alkalmazásoknál fontos lehet!

## Konkurencia

Elvileg lehetséges volna egy olyan probléma, hogy miközben éppen a modellünket vizsgáljuk, a user kattint egyet, és az elrontja a modellt, miközben rajzoljuk, és így pl. 2 helyen is megjelenhetne az űrhajója, ahogy elmozdul, itt is és ott is megtalálja a rajzoló-rendszer, amikor bejárja a modellt. Szerencsére ettől nem kell félni, ugyanis az egérkattintások és a bill.-leütések is ugyanazon az eseménysoron érkeznek, ahol a paint esemény is, és amíg a paint metódusból nem térsz vissza, nem fognak megérkezni. Kiv. ha másik ablakról van szó, mert annak külön eseménysora van (lehet). Akkor

szinkronizálni kell a rajzolást és a modellt.

**Sok sikert a munkához!**

*Szabó  András (szundi)*

http://www.inf.bme.hu/~szundi/

szundi@inf.bme.hu

---