SZAKDOLGOZAT

Zalavári Márk

2009
Debrecen

Debreceni Egyetem
Informatikai Kar

Szoftverfejlesztés JAVA nyelven:
MMORPG-Maker környezet fejlesztése

Témavezető:
Espák Miklós
Egyetemi tanársegéd

Készítette:
Zalavári Márk
Programtervező informatikus

2009
Debrecen

Tartalomjegyzék

Bevezetés

A számítógépes szerepjátékokban (RPG – Role-Playing Game) egy vagy több karaktert irányíthatunk egy elképzelt vagy valós, de általában igen mélyen kidolgozott világban. Ezen játékokban az első helyen a világ megismerése, a történet előremozdítása és a karakter(ek) fejlődése áll.

Azt, hogy egy-egy játékosnak milyen esélyei vannak az életben maradásra, elsősorban nem a gép előtt ülő ember fizikai reflexei vagy ügyessége dönti el, hanem játékbeli karakterének tulajdonságai, melyek e karakter életvitelének függvényében változhatnak. Így például ha harcról van szó, nem feltétlenül a gyorsabb, pontosabb játékos fog győzedelmeskedni, sokkal inkább az, aki karakterét már korábban felkészítette az ilyen helyzetekre. Ugyanígy a karakter tulajdonságaitól függenek az üzleti, diplomáciai, tudományos, vagy akár magánéleti sikerei. Egy karakter általában nem lehet mindenben elég jó, így a játékos döntésén múlik, hogy karakterét az élet mely területén teszi kiemelkedővé, s mely területeket hanyagolja el. Azt, hogy mi mivel hogyan függ össze, egy szabályrendszer fogalmazza meg, amely az adott világtól függően igen összetett is lehet.

Az MMORPG (Massive Multiplayer Online Role-Playing Game) az RPG interneten játszható változata, amelyben a világot benépesítő karakterek mögött nagyrészt hús-vér emberek állnak. Ez esetben bár a történet jóval kevesebb szerepet kap, a világ életszerűbbé válik: a diskurzusok, alkudozások valóságosak, az események kiszámíthatatlanabbak, a csaták a jóval nagyobb emberi részvétel miatt jóval hangulatosabbak. A világ sorsa nem determinisztikus többé, hanem a játékosok közösen alakítják.

Az általam AMMO névre keresztelt szoftvert néhány éve kezdtem el írni. A projekt célja egy olyan fejlesztői környezet, majd szerver létrehozása, amelynek segítségével a lehető legkevesebb programozói ismerettel viszonylag összetett RPG / MMORPG játék hozható létre.

A másik nagy cél a minél nagyobb kifejezőerő: A jelenleg forgalomban lévő ingyenes MMORPG-Maker-ek általános gyengesége, hogy az általuk létrehozható világok

meglehetősen szűk spektrumban mozognak, ami elsősorban a szabályrendszer változatosságának hiányában nyilvánul meg.

A szakdolgozat keretein belül az AMMO fejlesztői környezetének alapjaival foglalkozom. Ennek során bemutatom az eddig elkészült felületet, annak kezelését, majd ezt követően programozását, futás közben elérhető eszközeit, osztályait, működését.

Felhasználói dokumentáció

1. Általánosságok

A rendszer univerzumokra tagolódik, amelyek egymástól akár teljesen különbözőek lehetnek más-más szabályrendszerrel, történettel, helyszínekkel. Az univerzumok gyakorlatilag semmilyen kapcsolatban nincsenek egymással, mindegyik egy-egy külön MMORPG játéknak minősül.

Az univerzumok világokból állnak: Ezek a világok egy univerzumon belül azonos helyszínekkel rendelkeznek, s azonos szabályok szerint működnek. Céljuk a „túlnépesedés” elkerülése, számuk az univerzumban levő játékosok számától függ. (Minél több a játékos, annál több világ generálódik, hogy a népsűrűség ne haladjon meg egy bizonyos szintet.)

A világok felépítéséhez egy objektum-orientált eszköztár és szkriptnyelv áll rendelkezésre, melynek segítségével működés közben gyakorlatilag bármit megváltoztathatunk.

Jelenleg egy kliens van, amely egyaránt alkalmas a játékra, s a fejlesztésre is. Mivel még nincs szerver, jelenleg nincs lehetőség többjátékos módra, s a változtatások mentésére.

1.1. A játéktér

A játékterület a „Playground” fül alatt érhető el, ezen a képernyőn folyik maga a játék (lásd 1. ábra). A képen látható objektumok kiválasztása a bal illetve jobb egérgombbal történik, attól függően, hogy az elsődleges (jobb gomb), vagy a másodlagos (bal) kijelölésen kívánunk változtatni. Ezeknek a kijelöléseknek a parancsok végrehajtásánál van jelentősége: A parancsokat az elsődleges objektumnak adjuk ki és a másodlagosra irányulnak. Például a karaktereken értelmezett „Fire!” parancs kiadása esetén az elsődleges objektum – a karakter – a másodlagos objektumra fog tüzelni.

1. ábra: A játéktér

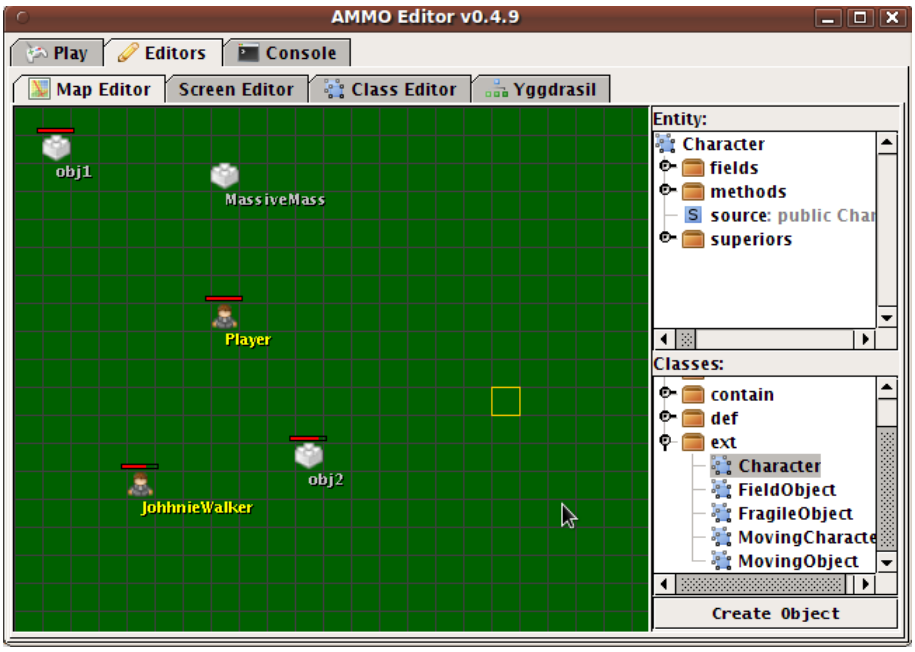
A jobb oldali panelen az elsődleges objektumon értelmezett parancsok láthatóak, ezek közül elhalványítva azok, amelyek az adott környezetben nem értelmezhetőek. (A példában: ha az elsődleges objektum egy karakter, akkor a „Fire” parancs szerepelni fog a listán, mivel a karakter képes erre a műveletre, ám ha a másodlagos objektum nem sérülékeny, akkor el lesz halványítva.)

Előfordulhat, hogy bizonyos gombok folyamatosan halványak, s a feliratuk mellett zárójelben egy rövid üzenet figyelmeztet különféle hibákra. Ez esetben a gomb mögött álló szkripttel vannak problémák (lásd a 2 fejezetet).

1.2. Szerkesztők

A rendszer, illetve az univerzumok/világok konfigurálására több szerkesztő áll rendelkezésre, melyek az „Editors” fül alatt bonyolultsági sorrendben követik egymást:

- A térkép szerkesztő („Map Editor”) a legegyszerűbb, mely használata nem igényel programozói háttértudást, ez szolgál a területek összeállítására, melyeken a játék zajlik.
- A képernyő szerkesztő („Screen Editor”) már egy fokkal bonyolultabb: itt adhatóak meg a parancsgombok beállításai, illetve a későbbiekben a képernyő megjelenését illető egyéb beállítások. Míg a területeket világonként is módosíthatjuk, addig a képernyő kinézete univerzum szinten egységes.
- Az osztály szerkesztőhöz („Class Editor”) már szükséges az objektum-orientált látásmód és némi programozói tapasztalat, – főleg amennyiben új viselkedést szeretnénk bevinni. Lényegében itt pakolhatjuk össze az univerzumot alkotó osztályokat.
- Az Yggdrasil szerkesztőhöz („Yggdrasil Editor”) pedig ezen felül még jól jön az AMMO működésének alaposabb ismerete, hiszen ennek segítségével rendszer szerte bármit közvetlenül módosíthatunk. (Az Yggdrasil-ról bővebben lásd a 2. fejezetet.)



2. ábra: A pálya szerkesztő

1.2.1 A pálya szerkesztő

A pálya szerkesztőn (lásd 2. ábra) a területek összeállítása történik. Az adott világban elérhető tereptárgy típusokat a „Classes” fül alatt találjuk. A megfelelő típust és a terület egy pontját kiválasztva a „Create Object” parancs kiadásával hozhatjuk létre a kívánt példányt. A már létrehozott példányok adatait kiválasztás után az „Entity” panelen láthatjuk. A tulajdonságaikat itt szerkeszthetjük is, ehhez duplán kell kattintani a módosítandó tulajdonságra.

1.2.2 A képernyő szerkesztő

Adott parancs beállításait a neve mellett levő „>>” feliratú gombbal hívhatjuk elő. Az előbukkanó panelen módosíthatjuk a parancs nevét, magát a parancsot, valamint az elsődleges, illetve másodlagos osztályt, melyen a parancsot értelmezzük. Minden parancs az elsődleges objektum egy metódusát hívja. Egy metódushívás a következőképpen néz ki:

```
metódus_neve( [pnév1:érték1 [, pnév2:érték2]...] )
```

A parancsokban használható változókat a 1. táblázat tartalmazza.

Integer, String, illetve Boolean értékek bekérése esetén a zárójelben szereplő felirat fog szerepelni az értéket bekérő párbeszédpanelen.

Alak:	Jelentés:	Osztály:
%1	Elsődleges objektum	Reference
%2	Másodlagos objektum	Reference
%I(String)	Felhasználótól bekért Integer	Integer
%S(String)	Felhasználótól bekért String	String
%B(String)	Felhasználótól bekért Boolean	Boolean
%x	A térképen kijelölt pont x koordinátája	Integer
%y	A térképen kijelölt pont y koordinátája	Integer

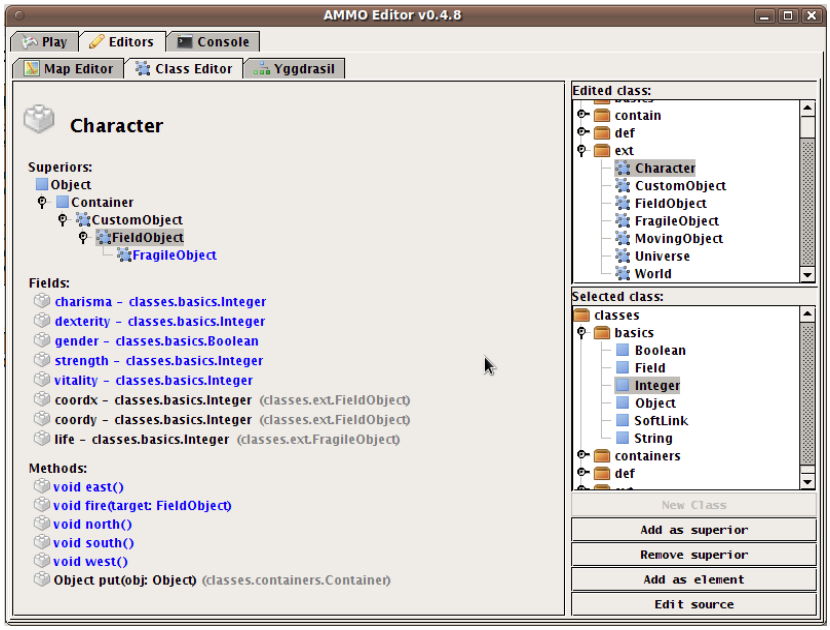
1. táblázat: A parancsokban használható változók

1.2.3 Az osztály szerkesztő

Amennyiben a meglévő osztályok nem elegendőek a megálmodott univerzum megvalósításához, akkor jön képbe az osztály szerkesztő (lásd 3. ábra). Itt hozhatunk létre új osztályokat, s módosíthatjuk a meglévőket.

A szerkesztendő osztályt a jobb felső panelről választhatjuk ki, melynek adatai ezt követően megjelennek az adatlapon. Az osztály neve alatt az ősöket látjuk fába rendezve. A kék színűek az explicit módon megadott ősök, a feketék pedig az ő őseik. További ősöket a jobb alsó panelről kereshetünk, majd kiválasztás után az „Add Superior” gombbal adhatunk hozzá. Eltávolításra az adatlapon való kijelölést követően a „Remove Superior” gomb szolgál.

Az adatlap következő eleme a tulajdonságok listája. A hozzáadás/eltávolítás az ősöknél leírtakhoz hasonlóan működik itt is.



3. ábra: Az osztály szerkesztő

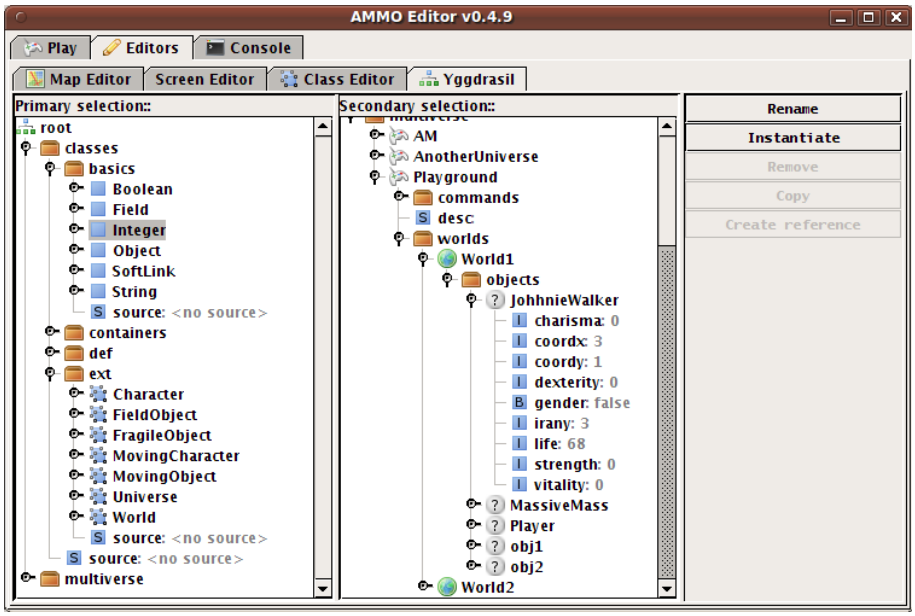
Metódusok hozzáadása az „Add Method” gomb megnyomásával történik. A létrehozást követően a tulajdonságlapon a metódus neve melletti nyílra kattintva megjelenő panelen végezhetjük el a beállításokat. Itt megadhatjuk a visszatérési értéket, a paraméterek neveit, típusait, alapértelmezett értéküket (ha van), valamint megírhatjuk a hozzájuk tarozó kódot. (Alapvető cél, hogy a felhasználónak a lehető legkevesebbet kelljen kódolnia, így a

későbbiekben kevésbé speciális rendszerek létrehozhatóak lesznek kód írása nélkül, pusztán a meglévő szabályrendszerek „keresztezésével”).

Új osztályt a csomag kiválasztását követően a „New Class” gombbal hozhatunk létre, mely az elnevezést követően alapértelmezetten a YComplexClass leszármazottja lesz.

1.2.4 Az Yggdrasil szerkesztő

Hozzáférést biztosít az Yggdrasil (lásd 2. fejezet) valamennyi objektumához és az azokhoz tartozó parancsokhoz. Általános esetben nincs szükség a használatára, mivel a felhasználás szempontjából lényeges funkciók jóval barátságosabb formában a többi szerkesztőben is megtalálhatóak, ám kísérletezésre, a fában való mélyebb vájkálásra, valamint az új funkciók kipróbálására kezdetben csak itt van lehetőség.



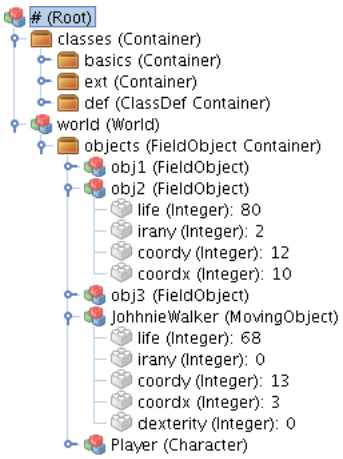
4. ábra: Az Yggdrasil szerkesztő

A bal oldali panelen választható ki az elsődleges, jobb oldalon a másodlagos elem: A parancsgombok a korábbiak szerint működnek itt is.

2. Az AMMO programozása

2.1. Áttekintés

A játék összes eleme – az osztályoktól a világok minden elemének tulajdonságaiig – egyetlen, Yggdrasil-nak nevezett fában tárolódik. (Az Yggdrasil nevét a germán és skandináv mitológiákból eredő, de a magyar népmesékből is ismert égig érő életfáról/világfáról kapta.)



5. ábra: Az Yggdrasil

Minden elem valamilyen osztályba tartozik, mely definíciója szintén megtalálható a fában. Az osztályok dinamikusan változtathatóak, azaz a definíció módosítása azonnal viszontlátható a már létrejött példányokon.

Az objektumok alapvetően két típusba tartozhatnak:

- Egyszerű objektumok: Jellegzetességük, hogy nem tartalmazhatnak további objektumokat, vagyis csak levélelemek lehetnek, s egyetlen értéket – önmagukét – képviselik. Ilyenek többek között az `Integer`, `String` és `Boolean` osztályok objektumai.
- Konténerek: Olyan csúcspontok, melyek további, előre megadott típusú, objektumokat tartalmazhatnak. (`StringContainer`, `YObjectContainer` `FieldObjectContainer`, `ContainerContainer`, stb.)

Van továbbá egy harmadik típus: az összetett objektumok halmaza. Ez a konténerek egy speciális változatát takarja, mely a bezárás megvalósításával még közelebb áll az objektum-orientált paradigma objektum-fogalmához. Továbbá a tartalmazott objektumok között lehetnek az adott osztályra jellemző kötelező objektumaik, melyek példányosításkor létrejönnek, s nem sem távolíthatóak el. (Ilyen osztályok: a `Character`, `FieldObject`, `World`, `Universe` stb.)

Az AMMO nyelvének szintaktikájának megalkotásához alapul a Java szolgált, ám van néhány lényeges eltérés: Először is megengedi a többszörös öröklődést (ez esetben az ősöket vesszővel elválasztva kell felsorolni), másrészt a paraméterlistákban nincs sorrendi kötés.

A Character osztály kódja:

```
public Character extends ext.FragileObject {
    public basics.Integer strength;
    public basics.Integer dexterity;
    public basics.Integer vitality;
    public basics.Integer charisma;
    public basics.Boolean gender;

    public basics.Void fire(ext.FieldObject target) {
         target.life = target.life - 10;
    }

    public basics.Void south() {
         coordy = coordy + 1;
    }

    public basics.Void east() {
         coordx = coordx + 1;
    }

    public basics.Void north() {
         coordy = coordy - 1;
    }

    public basics.Void west() {
         coordx = coordx - 1;
    }
}
```

A MovingObject osztály kódja:

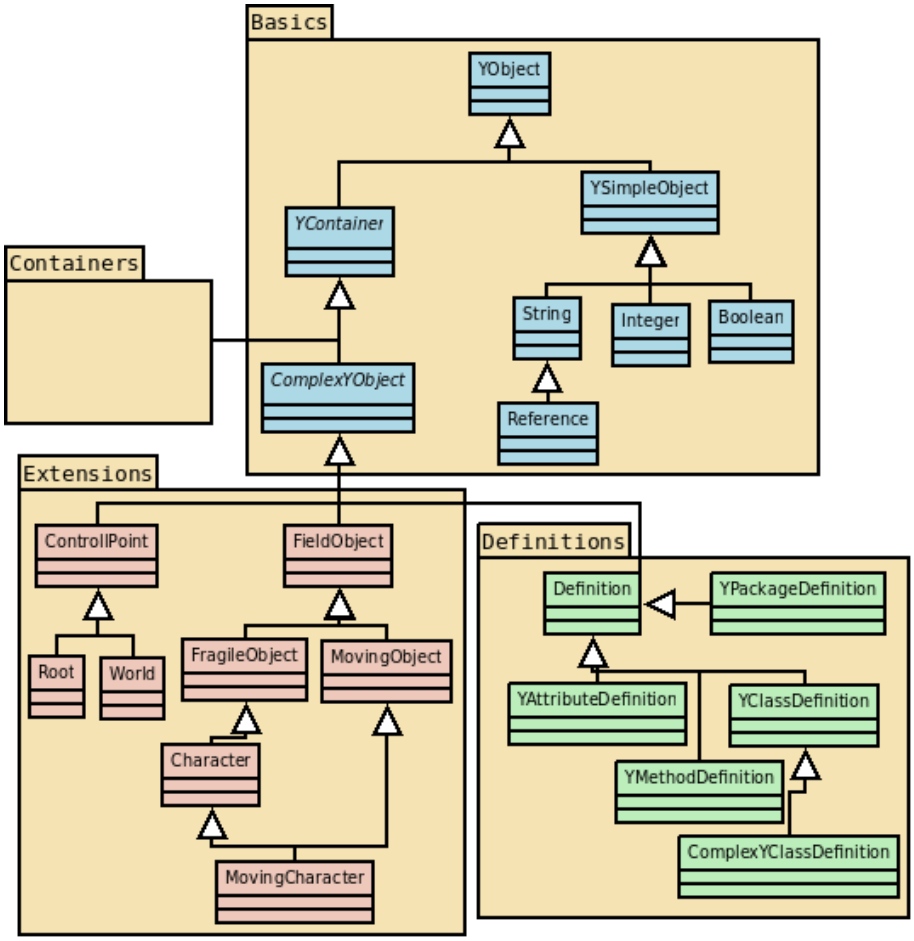
```
public MovingObject extends ext.FieldObject

    private basics.Integer irany;

    public basics.Void autoscript(){
         switch(irany) {
              case 0: {
                   south();
                   irany = 1;
              }
              case 1: {
                   west();
                   irany = 2;
              }
              case 2: {
                   north();
                   irany = 3;
              }
              case 3: {
                   east();
                   irany = 0;
              }
         }
    }
}
```

2.2. *Beépített osztályok*

Az AMMO beépített osztályinak nagy része nélkülözhetetlen az alapvető működéshez – Basics, illetve Definitions csomagok, valamint az Extensions csomag egy része –, míg mások az építéshez szükséges alapokat (FieldObject, Character), vagy egyéb kiegészítő funkciókat (ControlPoint, World) valósítanak meg.



6. ábra: Az AMMO beépített osztályai

2.2.1 A „Basics” csomag

Ez a csomag tartalmazza a legalapvetőbb osztályokat, mint például az `YObject`-et, ami az OO nyelvekhez hasonlóan itt is az ősosztály, továbbá itt találhatóak meg a primitív típusok megfelelői, a `String`, `Integer` és `Boolean` osztályok. A `Reference` egy relatív vagy abszolút útvonalat tartalmazó mező, mely az objektum-referenciát valósítja meg. A `Container` egy csak bizonyos osztályú elemeket tartalmazó objektum, melynek a

felhasználástól függően igen sokféle alosztályát lehet legenerálni, így az `Integer`-eket tartalmazó `IntegerContainer`-t, a `Container`-eket tartalmazó `ContainerContainer`-t és így tovább. A legenerált `Container`-ek közötti hierarchia a tartalmazott elemek típusára épül, tehát ha generálunk egy `ReferenceContainer`-t, legenerálódnak annak ősei is, a `StringContainer` és `FieldContainer`.

2.2.2 Az „Extensions” csomag

A `Root` az Yggdrasil gyökere, két kötelező elemmel rendelkezik: egy „multiverse” nevű UniverseContainer-rel, mely az univerzumokat tartalmazza, valamint egy „classes” nevű Container-rel, melyben a felhasználható osztálydefiníciók vannak csomagokba szervezve.

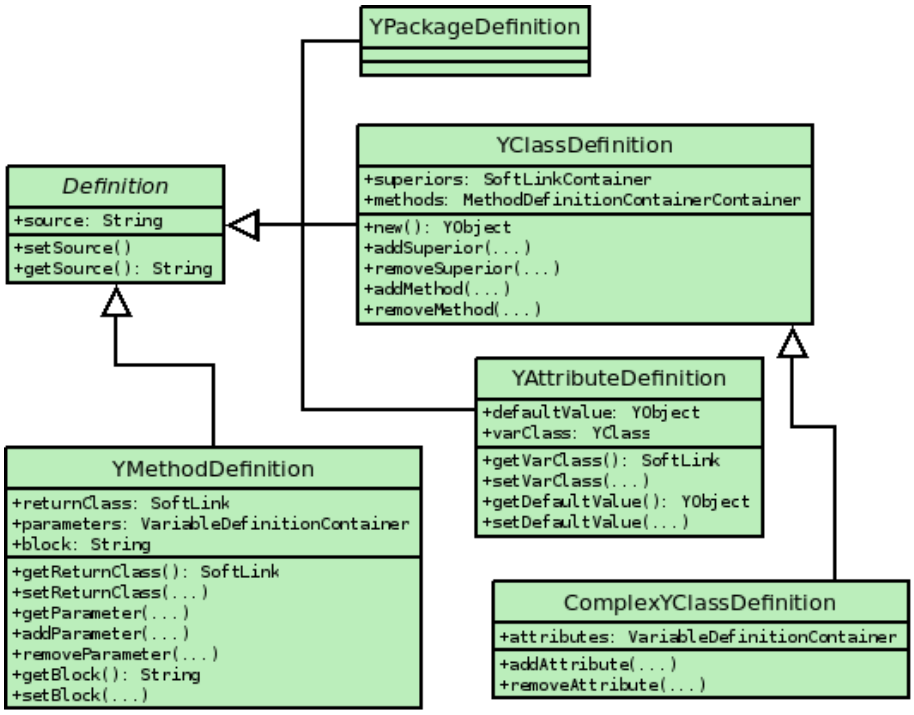
A `Universe`-nek két kötelező eleme van: Egy CommandContainer, melyben az alkalmazható parancsok, valamint egy WorldContainer, amiben az univerzumot alkotó világok vannak.

A `World` jelenlegi egyetlen kötelező eleme az „objects” nevű FieldObjectContainer, mely a játéktéren elhelyezett FieldObject-eket tartalmazza. A FieldObject már rendelkezik koordinátákkal, így elhelyezhető a játéktéren. Ennek leszármazottja FragileObject, mely a koordinátákon túl életerővel is rendelkezik, illetve a MovingObject, mely körbe-körbe mászkál a területen. A FragileObject leszármazottja a Character, mely többek közt képes FragileObject-ekre tüzelni.

Végül a MovingCharacter a Character és a MovingObject közös leszármazottja: egy körbe-körbe mászkáló karakter, amely lőni tud. (Ez utóbbi osztályok – a FieldObject leszármazottjai – jelenleg csak szemléltetésképp vannak a programban, ezen osztályok ugyanis már a konkrét világtól függenek.)

2.2.3 A „Definitions” csomag

A „Definitions” csomag tulajdonképpen a Java Reflection API végtelenül leegyszerűsített megfelelője. Osztályai egy-egy Y-Unit-ot csomagolnak be (YPackage, YClass, YComplexClass, YObject, YMethod), valamint tárolják a forráskódot, amelyből a becsomagolt elem létrejött, így az is elérhető, s módosítható a modellen belül.



7. ábra: A Definitions csomag

2.3. Láthatóság kezelés

Az AMMO láthatóságkezelése némileg eltér az OO nyelvekben megszokottól. A bezárást a ComplexYClass valósítja meg, az ettől primitívebb típusok tehát „átlátszóak”, vagyis a Konténerek tartalma korlátozás nélkül hozzáférhető. Jelenleg a `private` és `public` szint ismert, melyek az OO paradigmában megszokott fogalmakat takarja, de tervben van még a `protected` kategória.

Fejlesztői dokumentáció

3. Felépítés áttekintése

A rendszer négy kisebb projektre van osztva:

- „Common”: Ez a legterjedelmesebb. A kliens és szerver számára egyaránt szükséges, alapvető osztályokat tartalmazza. Ez elsősorban az Yggdrasilt és a szkriptnyelvet jelenti, de itt kapott helyet a naplózó is.
- „Client Engine”: tartalmazza azon osztályokat, melyre a szervernek nincs szüksége, ellenben nélkülözhetetlen alapja valamennyi Java alapú kliensnek. (Többek közt tartalmaz egy parancsértelmezőt, egy hálózati klienst, üdvözlőképernyőt és egy interfészt a konkrét felülethez, mely már nem része a csomagnak.)
- „Editor”: A felhasználói dokumentációban leírt jelenleg használt Swing alapú kliens.
- „Server”: Jelenlegi kezdetleges állapota miatt nem része a szakdolgozatnak.

4. Általános csomagok (Common Project)

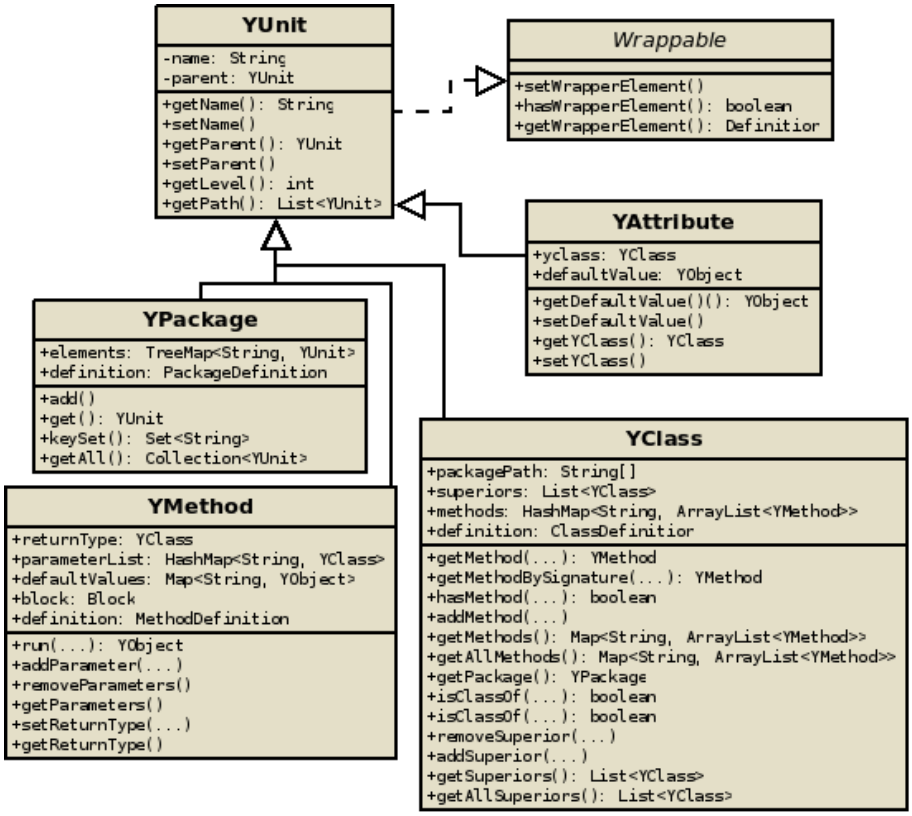
4.1. Vezérlés

A vezérlést mind kliens, mind szerver oldalon alapvetően két osztály végzi: Az általános vezérlő a kézi, illetve az ütemező az automatikus vezérlést.

Az általános vezérlő a kliens és szerver oldali vezérlők közös őse, mely itt még mindössze néhány alapvető funkciót valósít meg: Vezérli az indítást és leállítást, valamint elérhető általa az Yggdrasil Manager.

Az Yggdrasil Manager-en keresztül érhető el az Yggdrasil maga, valamint az ütemező. Az ütemező célja, hogy adott szkripteket megadott időnként vagy időpontokban lefuttasson. Jelenleg a fában megjelenő „autoscript” elnevezésű szkripteket futtatja fix időközönként.

4.2. A nyelv felépítése



8. ábra: A „yunits” csomag

4.2.1 Az YUnit-ok

Az Yggdrasil valamennyi eleme egy-egy YObject, melyek YClass-ok segítségével példányosíthatóak. Az YClass-ok YPackage-ekben helyezkednek el, s YMethod-okat, valamint összetett osztályok esetén YAttribute-okat tartalmazhatnak. Ezek valamennyien YUnit-ok, melyek egy fába szerveződnek. Ez fa a Definition-ök segítségével jelenhet majd meg az Yggdrasil-ban, így az a felhasználó szemszögéből egy önmagát leíró modellt fog alkotni. (lásd . fejezet). Az osztályok rövid leírása:

- YUnit: Valamennyi, a rendszeren belüli programstruktúrát alkotó elem YUnit (lásd a lista folytatását). A gyökérelemet kivéve valamennyi elemnek van pontosan egy szülője. Az YUnit-ok azonosítása a hozzájuk vezető útvonallal történik, így adott csomópontban minden YUnit neve egyedi kell legyen. Implementálja továbbá a

`Wrapper` interfészt, melynek segítségével Definition kapcsolható hozzá. (valamennyi YUnit-hoz pontosan egy darab Definition tartozhat, amennyiben elhelyezésre kerül az Yggdrasil-ban)

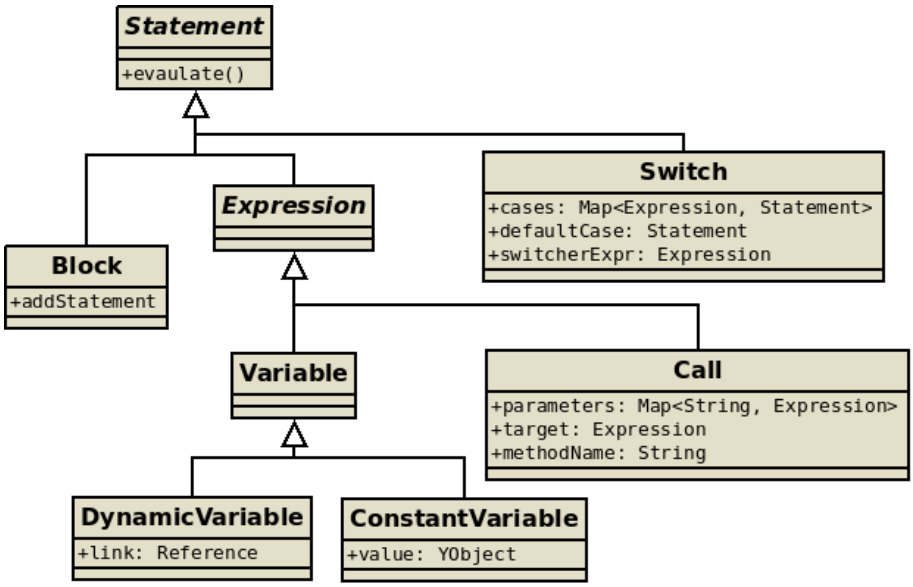
- YAttribute: Az OOP mező megfelelője, a fában csak levélelem lehet. Van típusa, neve és lehet alapértelmezett értéke. Nincs konkrét megfelelője a modellen belül, tartalmát lekérve egy YObject-hez jutunk.
- YMethod: A metódus megfelelője, szintén kizárólag levélelem lehet. Rendelkezik egy paraméterlistával (név-osztály párosokkal), valamint adott paraméterekhez rendelkezhet alapértelmezett értékkel. Tartalmaz továbbá egy blokkot (lásd a 4.2.2 fejezeben), melyet híváskor lefuttat a paraméterek által meghatározott környezetben és a megkapott objektumon.
- YClass: Absztrakt osztály, az OOP osztály megfelelője. Tartalmazhat YMethod-okat, melyeket szignatúra, vagy név és aktuális paraméterlista alapján lehet lekérni (ez esetben az adott nevű metódusok közül a megadott paraméterlistához legjobban illeszkető metódus kerül visszaadásra). Az YClass rendelkezhet ősökkel: Az YMethod-ok lekérése lehetséges csak az adott YClass-ra vonatkozólag, vagy az öröklődést figyelembe véve az ősök bevonásával is. Továbbá az `isClassOf(YObject)` metódussal megállapítható, hogy osztálya-e a megadott elemnek, illetve az `isSuperiorOf(YClass)`-al, hogy őse-e a megadott osztálynak.
- YPackage: Az OOP csomag megfelelője: YClass-okat és további YPackage-eket tartalmazhat, melyeket név alapján lehet tőle lekérni.
- YObject: Az OOP objektum-fogalom megfelelője. Egy objektum életciklusa a létrehozáskor kezdődik, s addig tart, míg eleme az Yggdrasil-nak (vagyis van szülője). Abban a pillanatban, hogy elveszti a szülőjét, eldobja a saját belső objektumait is, majd jelzi az osztályának, hogy nem része többé a fának: Ekkor kikerül a nyilvántartásból.

4.2.2 Utasítások

A szkriptekben használható utasítások implementációja a „Statements” csomagban található. A szkripteket elegendő egyszer – létrehozáskor – elemezni, s létrehozni belőlük a megfelelő utasítást. A `Statement` absztrakt osztálynak egyetlen publikus metódusa van, melyet a végrehajtáskor kell meghívni:

```
public abstract YObject evaulate(Map<String, YObject> environment, YObject thisElement);
```

Az environment-ben kell megadni az adott környezetben látható objektumokat, míg a thisElement-ben azt az objektumot, melyen az utasítás fut.



9. ábra: A "statements" csomag

Az utasítások közül a metódushívás (`Call`) érdemel néhány szót. Példányosításakor három dolgot kell megadni:

- A célobjektumot (`Expression`)
- A célobjektum egy metódusának nevét (`String`)
- A paramétereket (`Map<String, Expression>`)

A hívás menete:

- A példányosításkor megkapott paraméterek kiértékelése az aktuális környezetben, illetve objektumon

- Célobjektum megállapítása
- Hívandó metódus lekérése a név és a paraméterlista alapján (azonos nevű metódusok esetén hogy melyik választódik ki, futási időben derül ki – bővebben lásd az előző fejezetben)
- Amennyiben minden rendben ment, a megkapott YMethod futtatása a kiértékelt paraméterekkel

4.2.3 Fordítás

A megírt kódból az YUnit-okat és utasításokat a Synthetizer generálja le, mely egy az ANTLR[1] által generált elemzőt használ e célra (`AmmoLexer` és `AmmoParser` osztályok) Az elemző legenerálásához szükséges a szabályokat az `AmmoScript.g` fájl tartalmazza.

Amennyiben a kód szintaktikai hibákat tartalmaz, kivétel dobódik, mely tartalmazza a hibák leírását és keletkezésük helyét a kódban (sorindex, karakterindex) .

4.3. Beépített osztályok

4.3.1 Basics csomag

Itt helyezkednek el a rendszer alap YClass-ai. Elődlegesen a két alaposztály:

- SimpleYClass: Az egyetlen értéket képviselő YClass-ok őse. (pl. YStringClass, YIntegerClass, YBooleanClass).
- YContainerClass: Az YContainer osztályok őse. Factory metódusa gyártja le – vagy adja vissza a már meglevő – megadott osztályra specializált YContainerClass-t (olyan YContainer-eket hoz létre melyben csak olyan YObject helyezhető el, ami a megadott YClass-ba, vagy annak egy leszármazott YClass-ába tartozik).

Az YContainer-ekhez fűződik az Yggdrasil-ban bekövetkező események továbbítása is a fában felfele: Amennyiben új elem jön létre, régi törlődik, esetleg neve vagy értéke megváltozik, a megváltozott objektum értesíti az őt magában foglaló YContainer-t, ami aztán

az eseményt tovább küldi az ő szülőjének, egészen addig míg az fel nem jut a gyökérig. Így menet közben valamennyi releváns objektum értesül a változásról, a gyökér pedig továbbíthatja azt a vezérlő felé. Az értesülés elkapására a következő metódusok felülírásával nyílhat lehetőség:

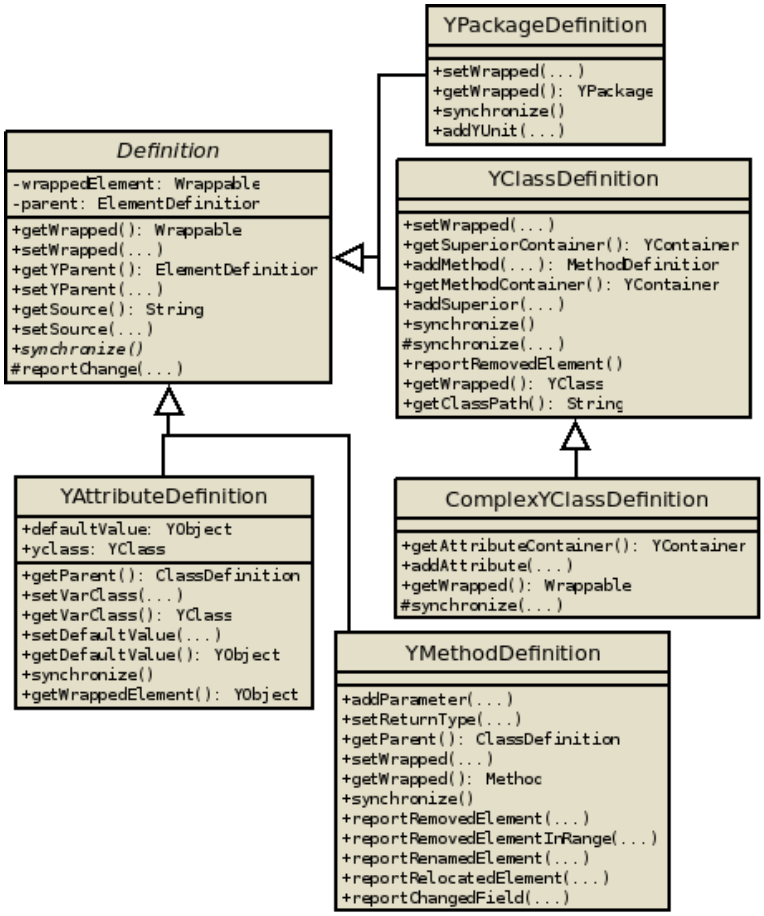
- `reportNewElement()`
- `reportNewElementInRange()`
- `reportRemovedElement()`
- `reportRemovedElementInRange()`
- `reportRenamedElement()`
- `reportRenamedElementInRange()`
- `reportRelocatedElement()`
- `reportRelocatedElementInRange()`
- `reportChangedElement()`
- `reportChangedElementInRange()`

Felülírás esetén figyelni kell rá, hogy meghívjuk az eredeti metódust is, mely majd továbbküldi az értesítést. Valamennyi értesítési típusnak két fajtája van, s minden esetben mindkét értesítés elkezd felgyűrűzni a gyökér felé, de az `InRange` típusú értesítések csak addig haladnak, amíg a megváltozott objektum látható (vagyis az első összetett objektumig).

Továbbá ebben a csomagban kapott helyet a ComplexYClass-ok őse is: Ez egy olyan YContainerClass, ami YAttribute-okat is tartalmazhat, melyek a kötelező elemeit definiálják. Elemei két metóduson keresztül kérhetőek le: Az `YContainer`-től öröklött `get(String)` metódussal, mellyel ez esetben csak a publikus elemeket adja vissza, s a `getPrivate(String)`-tel, mely bármely elem lekérésére használható. (megjegyzés: A felhasználó által futási időben létrehozott új osztályok mind az ComplexYClass-ok.)

Az alap osztálytípusokon kívül ebbe a csomagba került a Boolean, Integer, String és Reference egyszerű osztályok implementációja.

Definitions csomag



10. ábra: A Definitions csomag

A Definíciók elsődleges feladata, hogy a rajtuk végrehajtott változtatások az YUnite-okon is tükröződjenek, valamint hogy az YUnit-okbali változások megjelenjenek az Yggdrasilban (lásd 11. ábra).

Amennyiben egy YUnit változik meg, az automatikusan meghívja a hozzá tartozó Definíció – amennyiben van ilyen – `synchronize()` metódusát. A híváskor a Definition frissíti az elemeket, hogy azok szinkronban legyenek az YUnit aktuális állapotával.

Mivel a Definíciók nem csak az Yggdrasil részét képezhetik, hanem a becsomagolt elemek által az YUnit fának is részei, két szülővel is rendelkezhetnek: Az Yggdrasil-beli szülő az öröklött `getParent()`-tel, az YUnit szerintit pedig a Definíció által bevezetett `getYParent()` metódussal kérhető le.

A becsomagolt objektum a `getWrapped()` metódussal kérhető el. Az YAttribute kilóg a sorból, ugyanis míg a többi definíció tartalmából elég egyetlen példány, addig egy tulajdonságból nyilván akárhány különböző állapotú is létezhet egyszerre különböző YObject-ekben: Itt tehát a Prototype[2] tervezési mintát megvalósítva valemennyi `getWrapped()` híváskor készül egy másolat az alapértelmezett értékről, s ez adódik vissza.

11. ábra: A Definition osztályok használat közben

Nevükhöz méltóan a definíciók második lényeges feladata, hogy tárolják a kódot, amiből a becsomagolt elem létrejött, így megteremtve a lehetőséget, hogy az is elérhető, s módosítható legyen a modellen belül. A kód tárolása darabolva, hierarchikusan történik: minden egyes definíció csak a kód rá tartozó részével foglalkozik: a hierarchiában lejjebb elhelyezkedő elemek kódjainak csak a sajátján belüli helyét tudja. Mindig csak akkora rész áll össze, amekkorára felhasználónak szüksége van. (Amennyiben a reprezentáción keresztül kerül módosításra a csomagolt elem, úgy a kódnak bizonyos részei generáltak lesznek, míg az érintetlen részeknél megmarad a kézzel írott változat: Minden definícióhoz tartozik majd egy kódgenerátor, ami ezt a generálást elvégzi.)

4.3.2 Extensions csomag

Itt kapnak helyet azon osztályok, melyek nem nélkülözhetetlen részei az alap eszköztárnak, ám szükségesek lehetnek az univerzumok felépítéséhez. Jellemzően ComplexYClass-ok. Újak ide a már meglevők kiterjesztésével hozhatóak létre.

Egy általános ComplexYClass a következő módon van implementálva:

Az objektum:

```
public class World extends ComplexObject {

   protected World(String name, ComplexClass type) {
      super(name, type);
   }

   public YContainer getObjectsContainer(){
      return (YContainer) get(WorldClass.KEY_OBJECTS);
   }

  public static WorldClass yclass() {
      return WorldClass.singleton();
   }

   @Override
   public WorldClass getYClass(){
     return (WorldClass) super.getYClass();
   }
}
```

A z osztály:

```
public static class WorldClass extends ComplexClass {

   private static final String CLASS_NAME = "World";
   private static final String KEY_OBJECTS = "objects";
   private static WorldClass singleton;
   private static WorldClass singleton() {
      if (singleton == null){
         singleton = new WorldClass(CLASS_NAME, null);
         singleton.addMethod(Put.function);
         singleton.addElement(
            FieldDefinition.yclass().create(
            KEY_OBJECTS, YContainer.yclass(), null));
         placeClass(singleton);
      }
      return singleton;
   }

   protected WorldClass(String name, String[] path) {
      super(name, null);
   }
```

```
  @Override
  public World create(String name) {
     World newWorld = new World(name, this);
     init(newWorld);
     return newWorld;
  }

  private static class Put extends YMethod {

     private static final String FUNCTION_NAME = "put";
     private static final String KEY_OBJ = "obj";
     private static final String KEY_COORDX = "coordx";
     private static final String KEY_COORDY = "coordy";
     private static final YClass YCLASS_OBJ = ComplexObject.yclass();
     private static final YClass YCLASS_COORDX = IntegerField.yclass();
     private static final YClass YCLASS_COORDY = IntegerField.yclass();

     public static final Put function = new Put();

     protected Put() {
        super(FUNCTION_NAME);
        addParameter(KEY_OBJ, YCLASS_OBJ, null);
        addParameter(KEY_COORDX, YCLASS_COORDX, null);
        addParameter(KEY_COORDY, YCLASS_COORDY, null);
        setReturnType( YObject.yclass() );
     }

     @Override
     public YObject run(
        Map<String, YObject> params, Map<String, YObject> env,
        YObject thisElement) {

        World thisE = (World) thisElement;
        ComplexObject obj = (ComplexObject)params.get(KEY_OBJ);
        Integer coordx =
           ((IntegerField)params.get(KEY_COORDX)).getValue();
        Integer coordy =
           ((IntegerField)params.get(KEY_COORDY)).getValue();

        thisE.getObjectsContainer().put(obj);
        obj.set("coordx", coordx);
        obj.set("coordy", coordy);

        return obj;
     }
  }
}
```

A példa a rendszerben megtalálható World ComplexYClass implementációja. A World-nek jelenleg egyetlen eleme és egyetlen metódusa van: Az „objects” elem egy YContainer, melybe a térképen elhelyezett objektumok kerülnek. A put() metódus pedig akkor lesz segítségünkre, ha majd elemeket akarunk elhelyezni a térképen.

Fontos, hogy valamennyi YClass-t modellező objektum csak egyszer legyen példányosítva, mivel két osztályt csak akkor tekint a rendszer ekvivalensnek, ha őket azonos Java objektum reprezentálja. Az ilyen problémáknak könnyen elejét vehetjük, ha az `YContainer` és `YComplexClass` osztályok kivételével valamennyi YClass singleton lesz.

Az YClass példányosításakor adni kell neki egy nevet és egy elérési útját, ahova kerülni fog. Amennyiben ez utóbbi helyett null-t adunk meg – mint a példában is –, az új YClass az ősével megegyező YPackage-ben kerül majd elhelyezésre.

Ezt követően fel kell építeni új ComplexYClass-t az YClass-tól örökölt metódusok segítségével (`addSuperior`, `addAttribute` és `addMethod`), majd el kell helyezni a csomaghierarchiában a `placeClass()` metódussal.

Az YAttribute-ok megadása AttributeDefinition-ök megadásával történik: Ezek példányosítása során meg kell adni a tulajdonság nevét, típusát és alapértelmezett értékét (ennek hiányában a típus előre beállított alapértelmezett értéke fog beállítódni).

Az YMethod-ok definiálása valamivel összetettebb: Minden egyes metódushoz létre kell hoznunk egy új osztályt, melyet az `YMethod`-ból származtatunk: Ezek szintén singleton-ok lesznek. Példányosításkor meg kell adnunk az YMethod nevét, majd beállítani a paramétereket és a visszatérési értéket. (A paraméterek beállításánál a tulajdonságokhoz hasonlóan a nevet, a típust és az alapértelmezett értéket kell megadni. Az alapértelmezett érték elhagyása esetén a paraméternek az YMethod meghívásakor kötelező lesz értéket adni)

A következő lépés az új YMethod működésének megírása: ez a `run()` metódus felülírásával tehető meg. A metódusnak két paramétere van: A „params” a paraméterként megkapott név-érték párokat tartalmazza, a „thisElement” pedig azt az objektumot, amin az YMethod-ot meghívták. A paraméterlistát itt már szükségtelen ellenőrizni, ugyanis amennyiben valamelyik paraméter hiányzik, vagy típusa nem megfelelő, a vezérlés el sem jut idáig.

Egy másik fontos dolog, hogy az YObject-eket ne közvetlenül, hanem YClass-uk create() metódusával gyártsuk le, ugyanis ahhoz, hogy az YClass megváltozása esetén a változás az objektumaikon is végbemenjen, az YClass-oknak tudniuk kell a példányosított YObject-ekről.

A ComplexYClass-oknak továbbá szükséges a `create()` metódusban az `init()` meghívása, mely inicializálja az elemeket.

Végül megadjuk az esetleges getter/setter metódusokat, melyekre valószínűleg szükségünk lesz, amennyiben az objektum elemeit nem csak a modellen belül akarjuk elérni.

5. Kliens Motor (Client Engine Project)

5.1. Vezérlés

5.1.1 Kliens oldali vezérlés

A kliens vezérlő (`ClientController`) az általános vezérlő (`GeneralController`) leszármazottja, kiegészíti azt a csak kliens oldalon használatos funkciókkal: Számon tartja az Yggdrasilban kijelölt elsődleges és másodlagos elemet, a kiválasztott univerzumot és világot. A program egyéb részei számára hozzáférést biztosít hálózati klienshez, a parancsértelmezőhöz (`Commander`), valamint a felülethez az `IClientUI` interfészen keresztül.

A kliens vezérlőben regisztrálhatóak az `IClientControllerListener`-t implementáló objektumok, melyek a ezt követően értesítést kapnak a kiválasztások megváltozásáról (elsődleges/másodlagos elem, aktuális univerzum, illetve világ).

5.1.2 A parancsértelmező

A parancsértelmező feladata, hogy a – csak kliens oldalon értelmezett – parancsokat elemezze, majd visszajelezzen annak állapotáról vagy ha az a feladat, futtassa azt.

Egy parancs a következő részekől áll:

- megnevezés: Amennyiben a parancs megjelenik a felületen – például egy gomb formájában –, ezzel a névvel jelenik meg. (nem szolgál azonosításra, nem kell egyedi legyen)

- parancs kódja: A parancs végrahajtásakor lefuttatandó általános kód (csak kliens oldalon értelmezett: Annyiban különbözi az utasítástól, hogy tartalmazhatja a 1.2.2-es fejezetben leírt szimbólumokat)
- elsődleges Y-objektum osztálya: Azon Y-osztály, melyeken a parancs értelmezhető
- másodlagos Y-objektum osztálya: Azon Y-osztályok, melyeken a parancs végrehajtható

A végrehajtás lépései:

- Ellenőrzésre kerül, hogy a elsődleges és másodlagos Y-osztályok értelmezhetőek-e az adott környezetben: Amennyiben az általuk megadott osztályok valamelyike nem létezik, a parancs nem futtatható.
- Amennyiben az aktuális elsődleges vagy másodlagos elem nem tartozik az elvárt osztályba, a parancs nem futtatható.
- Amennyiben a korábbi feltételek fennállnak, megkezdődhet a behelyettesítés. Ha nem végrehajtás a cél, hanem csak a parancs állapotának megállapítása, úgy a parancsértelmező nem fog a felülethez fordulni a felhasználótól bekérendő adatokért, hanem a változókba alapértelmezett értékek fognak kerülni. Felhasználói interakció esetén nem biztos, hogy a behelyettesítés sikerrel jár, ez esetben a művelet megszakad.
- Ezt követően a Synthetizer segítségével kísérlet történik az utasítás legenerálására. Amennyiben ez nem sikerül, a Synthetizer visszadobja a szintaktikai problémákat, mely továbbmehet a felhasználó felé.
- Amennyiben az utasítás sikeresen legenerálódott, a parancs szintaktikailag rendben van és az aktuális környezetben futtatható is. Amennyiben futtatás volt a cél, meghívódik az utasítás evaulate() metódusa az aktuális környezettel és az elsődleges objektummal.

5.2. View

5.2.1 WelcomeScreen

Swing alapú üdvözlőképernyő, mely a felhasználónév és jelszó megadását követően elvégzi a bejelentkezést, illetve visszajelez, amennyiben problémák merülnének fel. A „Login” gomb megnyomásakor egy szálat indít, mely elküldi a szerver felé a kérést, maximum 5 máspodpercig várakozik a válaszra, majd az eredményről értesíti a WelcomeScreen-t. Az eredménytől függően vagy visszakerül az ablak a gomb megnyomását megelőző állapotba, vagy inicializálja a példányosításkor megkapott felületet (`IClientUI`).

Továbbá innen érhető el a kapcsolat nélküli mód is – mely jelen esetben az egyetlen választható alternatíva.

5.2.2 IClientUI

A Common csomag az üdvözlőképernyőn túl nem tartalmaz további felületi elemeket, hanem csak egy interfészt definiál, melyet a majdan létrehozott konkrét klienseknek implementálniuk kell. Az interfészen jelenleg definiált funkciók:

- Felület inicializálása
- Integer kérése a felhasználótól
- String kérése a felhasználótól
- Boolean kérése a felhasználótól
- Programkód kérése a felhasználótól
- Üzenet írása a konzolra
- Hibaüzenet jelentését jóváhagyó ablak megjelenítése

Lényeges, hogy a felület ne példányosításkor, hanem inicialiáláskor jelenjen meg, ugyanis a példányosítás már indításkor megtörténik, majd az inicializálatlan felület kerül tovább az üdvözlőképernyőhöz, mely azt csak a bejelentkezést követően fogja inicializálni.

6. Swing Kliens (Swing Editor Project)

6.1. Felület

A program felülete a Swing[3] eszköztárának felhasználásával készült, melynek leegyszerüsített kinézetét a DefaultMetalTheme felülírásával egy saját téma határozza meg.

6.1.1 CommandBar

Legenerálja az aktuális univerzumhoz tartozó parancsoknak megfelelő gombokat. Adott gombra kattintva egy SwingWorker objektum segítségével meghívja a háttérben a parancsértelmezőt, mely majd végrehajtja a megadott parancsot. Továbbá Implementálja az `IClientControllerListener` interfészt, aminek segítségével a kliens vezérlőhöz listener-ként hozzáadva folyamatosan értesülhet annak változásairól, így a parancs-lista mindig aktuális lesz.

A CommandBar-nak két üzemmódja létezik, melyek között példányosításkor lehet választani. Editor módban nem csak a parancsok kiadására, hanem szerkesztésükre is lehetőségünk nyílik. Ez esetben a Parancs Y-objektumok által közvetlenül az Yggdrasil-t módosítjuk.

6.1.2 MapView

Kettős feladata van: egyrészt megjeleníteni az aktuális világ objektumait, jelölve az épp kijelölt elsődleges és másodlagos elemet, – amennyiben az a megjelenített objektumok között van –, másrészt ha a felhasználó módosít a kijelölésen, továbbítani a módosítási kérést a vezérlő felé.

6.1.3 SelectorTree

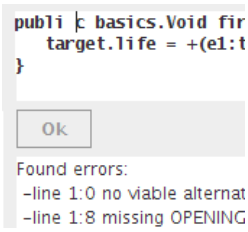
Az Yggdrasil különböző aspektusainak megjelenítésére, s az adott nézetben megjelenő elemek kiválasztására szolgál. Két alaptípusa van, melyek factory metódusokkal hozhatóak létre: A `createYggdrasilSelector()` az Yggdrasil alapértelmezett nézetét adja (egy YggdrasilTreeModel áll mögötte). Ez esetben a megjelenített fa ekvivalens az Yggdrasil egy részfájával. A `createClassSelector()` által létrejövő komponens viszont csak a csomagokat és osztályokat jeleníti meg (ClassTreeModel), melyek elrendezése nem feltétlenül egyezik meg az elemek Yggdrasil-ban elfoglalt helyével.

Mindkét esetben megadható, hogy a komponensben kijelölt elem az elsődleges, vagy a másodlagos kijelölést reprezentálja, illetve az YggdrasilSelector esetében megadható, hogy az Yggdrasil-nak pontosan melyik részét jelenítse meg.

6.1.4 MapEditor

A térkép szerkesztésére szolgáló eszköz Egyrészt tartalmaz egy MapView-t, mely a módosítandó objektumok kijelölésére, valamint az új objektumok helyének kiválasztására szolgál, valamint két SelectorTree-t. Az egyik egy elsődleges szelektor, amin a módosítás alatt álló objektum elemei láthatóak, a másik egy másodlagos, ami mögött egy ClassTreeModel áll: itt választhatjuk ki az újonnan létrehozandó objektum típusát.

6.1.5 ScriptEditor

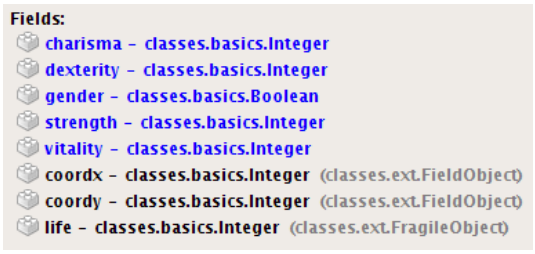


A forráskód menet közbeni módosítását lehetővé tevő felületi elem. Példányosításakor egy ElementDefinition-t kell megadni, melytől a szerkesztő lekéri a kódot. A kód szintaktikai elemzése folyamatos, így a felhasználó mindig látja az épp aktuális hibákat az ablak alsó részén. Amennyiben a kód helyes, a változtatások elmenthetőek. (A ScriptEditor

hatáskörébe csak az új, ellenőrzött kód elmentése tartozik, a változások végrehajtásáért nem felel.)

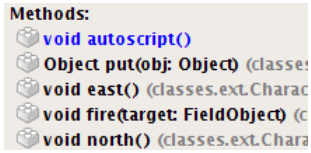
6.1.6 AttributeList

A `JList` osztályt terjeszti ki: Az elsődleges objektum osztályának elemeinek nevét és típusát jeleníti meg. Az örökölt elemek fekete, az osztály saját elemei kék színnel kerülnek kiírásra. Az adott elemen történő kattintásra a elemekhez tartozó AttributeDefinition másodlagos elemként kijelölésre kerül. A háttérben egy AttributeListModel áll.



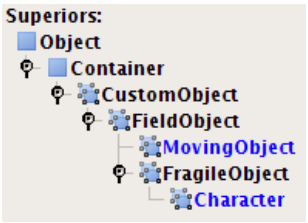
6.1.7 MethodList

Működése a AttributeList-hez hasonló, azzal a különbséggel hogy a metódusok megjelenítésére és kiválasztására hivatott. A metódusok származási helyük szerint kerülnek csoportosításra, kezdve az osztály saját metódusaival, s haladva felfele az öröklődési fában.



6.1.8 SuperiorTree

A `JTree` osztály kiterjesztése, az elsődleges objektum osztályának őseit jeleníti meg fába rendezve (a gyökér az Object, a levélelemek pedig a közvetlen ősök). A színezés – kék/fekete – itt is a korábbiak szerint történik. Amennyiben a többszörös öröklődésből kifolyólag egy osztály több párhuzamos öröklődési ágon is megjelenik, úgy a fában is többször fog szerepelni.



6.1.9 ClassEditor

Tartalmaz egy SuperiorTree-t, egy AttributeList-et, valamint egy MethodList-et, továbbá két SelectorTree-t, melyek az Yggdrasil „classes” csomópontját jelenítik meg. A felsőn választható ki a szerkesztendő objektum (elsődleges szelektor), a másodikon pedig a kiválasztott, vagy a létrehozandó elem osztálya.

7. Továbbfejlesztési lehetőségek

Mivel a program jelenleg is fejlesztés alatt áll, nyilván rengeteg hiányossággal rendelkezik, ami jelenleg elsősorban az öröklődés és elrejtés területén jelentkezik, de van mit javítani a bolondbiztosságon és a hibajelzés minőségén is. Jelenleg ezen hiányosságok pótlása az elsődleges cél.

Ezt követné a jogosultságkezelés bevezetése, melynek segítségével különböző felhasználóknak eltérő jogaik lehetnek a különböző objektumokon, majd amint az alapvető működés megvan, sor kerülhet a szerver fejlesztésének megkezdésére.

8. Köszönetnyilvánítás

Köszönetemet szeretném kifejezni elsőként tanáraimnak, akik megismertettek a programozás rejtelmeivel, köztük kiemelten Juhász Istvánnak, aki helyretette a fejlesztésről és tervezésről alkotott elképzeléseimet.

Szeretném megköszönni Espák Miklósnak – témavezetőmnek – a kezdeti nehézségekben nyújtott segítségét, s későbbi hasznos tanácsait, valamint türelmét.

Továbbá köszönöm családomnak a munkához biztosított hátteret, helyet, időt, s megértést, valamint Varga Nórának s Balog Péternek a morális segítséget, akik a már a kezdetektől figyelemmel kísérték a fejlesztést, s érdeklődésükkel, ötleteikkel, kritikáikkal ösztönzően hatottak a projekt haladására.

9. Irodalomjegyzék

[1] Terence Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages, 2007
[2] Gamma, Helm, Johnson & Vlissides, Design Patterns, 1997
[3] Brian Cole, Robert Eckstein, James Elliot, Marc Loy, David Wood, Java Swing 2nd Edition, 2002