Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Angeli Róbert, Fekete András

BIOFEEDBACK ALAPÚ KERETRENDSZER ADAPTÍV MOBIL OKTATÓJÁTÉKOKHOZ

KONZULENS

Dr. Forstner Bertalan

BUDAPEST, 2013

Tartalomjegyzék

Összefoglalás

A projekt célkitűzése készségfejlesztő- és oktatójátékok megalkotása, valamint az oktatójátékokhoz tartozó keretrendszer fejlesztése, mely a kognitív állapot becslésén és neurális visszacsatoláson alapszik, és kellőképpen általános különböző biológiai visszacsatoláson alapuló eszközök illesztéséhez. Cél a tanulás hatékonyságának javítása a felhasználó tanulási képességeihez alkalmazkodva. A szívritmus mérő, valamint az EEG szenzor használata lehetővé teszi, hogy a rendszer a nehézségi szint és a jutalom adaptív módosításán keresztül hozzásegítse a felhasználót a legjobb tanulási eredmények elérésében, hiszen az érzelmi és kognitív állapotok vizsgálata által növelhető a tanulás eredményessége, hatékonysága.

Kulcsszavak: készség fejlesztés, oktatójáték, biofeedback

1 Bevezetés

A dolgozat témája az Automatizálási és Alkalmazott Informatika Tanszéken immár 5. féléve futó InnoLearn projekt, melyen Nagy Richárddal, és Szita Ádámmal együtt dolgozunk. Richárd feladata az EEG eszköz illesztése volt, Ádámé pedig a felügyelő alkalmazás, valamint a szerver megvalósítása. További oktatójátékok is készülnek a keretrendszerünkre, például Simon Endre Andrásnak a hangszeres zenét oktató szoftvere. Jelen dolgozat a szerzők projekthez kapcsolódó eredményeit mutatja be. A mi feladatunk a keretrendszer megtervezése és megvalósítása mellett két, a keretrendszert használó játék elkészítése volt.

Az információs technológia és a mobil eszközök gyors fejlődése lehetővé tette, hogy a tabletek piaca egyre népszerűbbé váljon az okostelefonok terjedése mellett. A tabletek megtartják az okostelefonok összes előnyét, mint a kötelező érintőképernyő, megfelelően erős számítási kapacitás és mobilitás, továbbá sokkal nagyobb kijelzőt biztosítanak telefonhívásra is alkalmas elődeiknél. A nagy képernyő képessé teszi őket az okostelefonok szokásos funkcióinak kiterjesztésére. Többek között emiatt, valamint az intuitív felhasználói felület megjelenítésének képessége miatt megfelelő alapul szolgálhatnak komoly oktatójátékok fejlesztéséhez, melyek segítik a tanulást, és a tanulási készségek fejlesztését.

A fentieket megerősíti, hogy az oktatás több területén használnak tableteket. Szándékunk egy olyan összetett alkalmazás létrehozása, amely kiterjeszti ezt, a tanulási folyamat hatékonyságának maximalizálásával. Hogy ezt megvalósítsuk, az alkalmazás rugalmas módon alkalmazkodik a felhasználó tanulási képességeihez. Ez azért fontos, mert az említett tanulási folyamat minden ember számára különböző lehet. A tanulási fázisok folyamatos monitorozása több szenzor felhasználásával visszacsatolásként működik, ez pedig lehetővé teszi, hogy a rendszer befolyásolja az aktuális feladat nehézségi szintjét, továbbá a játék által adott jutalom különböző attribútumait, ami segít a legjobb tanulási eredmény elérésében.

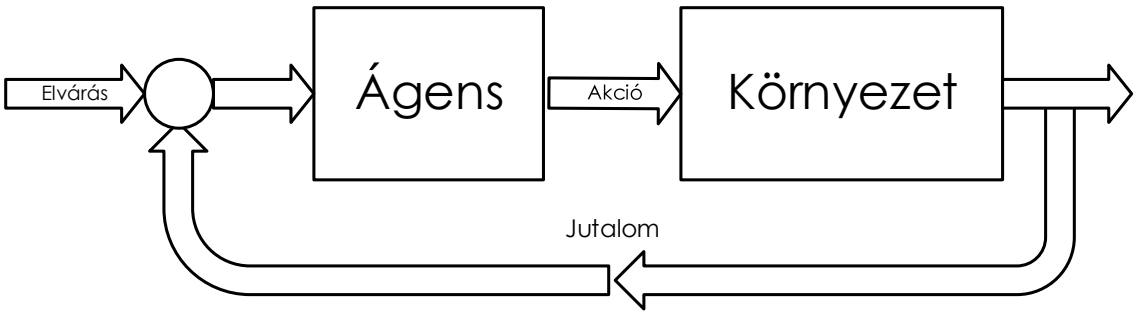
A legfontosabb eszköz a fent említett szenzorok közül az EEG, mely a visszacsatolás alapjául szolgál. A 14 csatorna és 2 giroszkóp használatával folyamatosan küldi a mért adatokat az agyban keletkező mikrovolt nagyságrendű amplitúdóval rendelkező agyhullámokról.

A dolgozat további része a következőképpen épül fel. A 2. fejezetben bemutatjuk az idegtudományi területek kapcsolódó eredményeit, melyek a keretrendszer működési elvét magyarázzák. A 3. fejezet a felhasznált eszközöket veszi sorba, továbbá bemutatja az alkalmazás egy tipikus felhasználási esetét. Ezt követi a 4. fejezet, mely a rendszer felépítésének részleteit ismerteti meg az olvasóval, és bemutatja az alkalmazás készítése során felmerülő problémákat, és az ezeket megoldó tervezői döntéseket. Az 5. fejezet a keretrendszer felhasználásával általunk készített játékokról szól. Végül a 6. fejezet összefoglalásával zárul a dolgozat.

2 Elméleti háttér

A számítógépes játékok hatásának vizsgálata széles körben elterjedt téma a kutatók között. Korábbi tanulmányok például megmutatták, hogy az akciójátékok képesek fejleszteni a figyelő képességet [1]. Továbbá korábbi tanulmányok bevezettek különböző biológiai visszacsatoláson alapuló metódusokat. Céljuk az alany érzelmi állapotának irányítása volt a szívritmus, a vérnyomás, vagy az anyagcsere monitorozásával [2][3]. A dolgozatunk a sokféle különböző megközelítést összegzi, és egy új megközelítést mutat be.

A kutatásunk mögött álló elmélet a megerősítéses tanuláson alapszik [4]. A megerősítéses tanulás egy jól ismert algoritmus a gépi tanulás területén melyet az elmúlt évtizedekben a kognitív idegtudomány is adaptált. Az elmélet alapvető célja, hogy megmagyarázza, a jutalomkereső viselkedés hogyan befolyásolja az ágens döntéshozó mechanizmusát. Egy adott környezetben több akció közül lehet választani, és az ágens elsődleges célja, hogy a lehető legnagyobb globális jutalmat érje el a környezetben. Az ágens ennek elérése érdekében lép interakcióba a környezetével.



2.1. ábra Megerősítéses tanulás

Ebből adódóan az ágens a végrehajtandó akciót a várható jutalom nagysága szerint választja ki. Azonban ahogy az az **Hiba! A hivatkozási forrás nem található.**. ábrán is jól látható, tanulás előtt ez a becsült jutalom eltér a valóstól, és ez a hiba jelzi a tanulási folyamat eredményét [5][6][7][8].

A kognitív idegtudományban neurotransmitterek felelősek az elvárt és a kapott jutalom közti különbségért [9][10]. Ha a kapott jutalom nagyobb a becsültnél, az dopaminszint növekedést eredményez, mely boldogság érzethez vezet. A videó játékokkal történő játéknak is hasonló hatásai vannak [11]. A legújabb tanulmányok fizikai magyarázatot kerestek a számítógépes játékok hatására [12]. Játék közben ugyanaz a

folyamat megy végbe a háttérben, mint a tanulás során. A számítógépes játék során a játékos hasonló örömöt és jutalomérzetet tapasztal meg, mint a tanulás során.

A dolgozatban meg fogjuk mutatni, hogy oktatójátékokban a megerősítéses tanulásra épített, a keretrendszerünktől származó visszacsatoláson alapuló jutalom manipuláció jobb tanulási képességhez vezet.

A projekt célja egy olyan keretrendszer megalkotása, mely támogatja oktatójátékok fejlesztését, és megnöveli azok teljesítményét a keretrendszer által gyűjtött adatok szerint adaptívan módosított jutalom által.

3 A rendszer bemutatása

Ebben a fejezetben a rendszert alkotó eszközök mellett a felhasználás egy tipikus esetét mutatjuk be.

3.1 A felhasznált eszközök

Az alkalmazás megvalósításához az Android platformot választottuk, mely az okostelefonok és tabletek piacának egyik legjelentősebb szereplője. Rendszerünk megvalósítására a jó minőségű, részletes fejlesztői dokumentációk mellett az teszi alkalmassá, hogy támogatást nyújt USB eszközök illesztéséhez, amire a szenzorok csatlakoztatásához lehet szükség, valamint lehetővé teszi natív, C illetve C++ nyelvű kódrészek beágyazását az alkalmazásokba, mellyel az olyan teljesítménykritikus részek megfelelő sebessége biztosítható, mint amilyen esetünkben a szenzor jelek feldolgozása. Jelenleg egy mobil EEG eszköz illesztése megoldott, és egy szívritmus mérő eszköz illesztése van folyamatban.

3.1. ábra Emotiv Epoc EEG

Ahogy azt említettük, a rendszerhez már sikeresen illesztettünk egy mobil EEG-t. Ez a fenti képen is látható *Emotiv Epoc* típusú eszköz, mely nagy felbontású mérést biztosít 14 csatornán, továbbá vezeték nélküli csatlakozást tesz lehetővé, mely a mobilitást tovább fokozza. Ezen kívül giroszkópot is tartalmaz, mely az optimális pozícionálást teszi lehetővé, így lehetővé válik a számunkra fontos jelek érzékelése. Ilyen a *P300*, ami egy eseményhez kapcsolódó potenciál, mely korábbi tanulmányok alapján a jutalom feldolgozásához kapcsolódik, és érzékeny a jutalom mértékére [13]. Továbbá ide sorolható

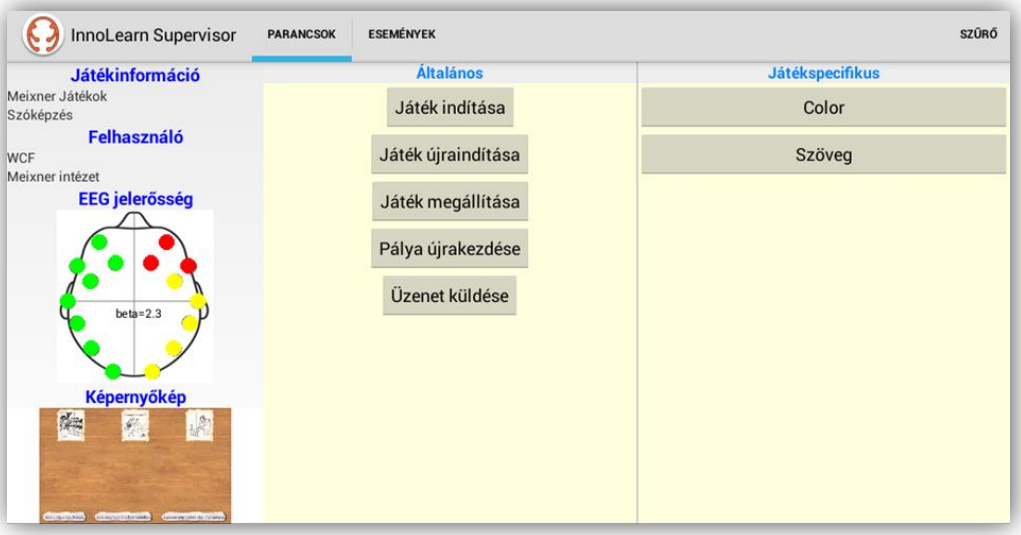
az agyhullámok különböző típusainak (alfa, béta, théta) megkülönböztetése, és ezeknek az egyes tartományokban leadott teljesítményének mérése (például középfrekvenciás béta), mely a koncentráció mértékéről nyújt információt. Az eszköz az Android rendszer USB port támogatására építve csatlakoztatható tabletekhez is, melyet felhasználva a keretrendszerhez történő illesztését Nagy Richárd végezte, amit egy másik TDK dolgozatban mutat be.

A szívritmus mérők közül a *Zephyr HxM BT Heart Rate Monitor* típusú eszközére esett a választás, mivel Bluetoothon keresztül egy nyílt API-t használva kommunikál, és Java SDK-t is tartalmaz. Ezen kívül a hagyományos *Bluetooth* szabványon alapul, így kompatibilitási problémák sem merülhetnek fel, szemben a *Bluetooth Low Energy*-vel, melynek támogatása rendszerszinten csak az Android legújabb, 4.3-as verziójától biztosított, ám sok esetben ez sem garantálja a zökkenőmentes működést. Az eszköz megérkezése csak a dolgozat leadási határideje után várható, ezért csak az illesztés előkészítésének lépéseit mutatja be egy későbbi fejezet.

3.2 A rendszer működése

Az alkalmazás felhasználásának tipikus módja a következő:

1. A mérés iskolai környezetben, egy tanteremben folyik.
2. A résztvevők tipikusan egy felügyelő pedagógus, aki a mérés koordinálásáért felelős, és egy vagy több gyermek, akik az oktatójátékkal játszanak.
3. A mérés megkezdése előtt a gyermekekre a pedagógus biofeedback eszközöket csatol.
4. A pedagógusnál lévő tableten a mérés felügyeletét elősegítő alkalmazás fut, mellyel a játék annak indulásakor automatikusan felépíti a kapcsolatot. Ennek felhasználói felülete látható az alábbi képen.



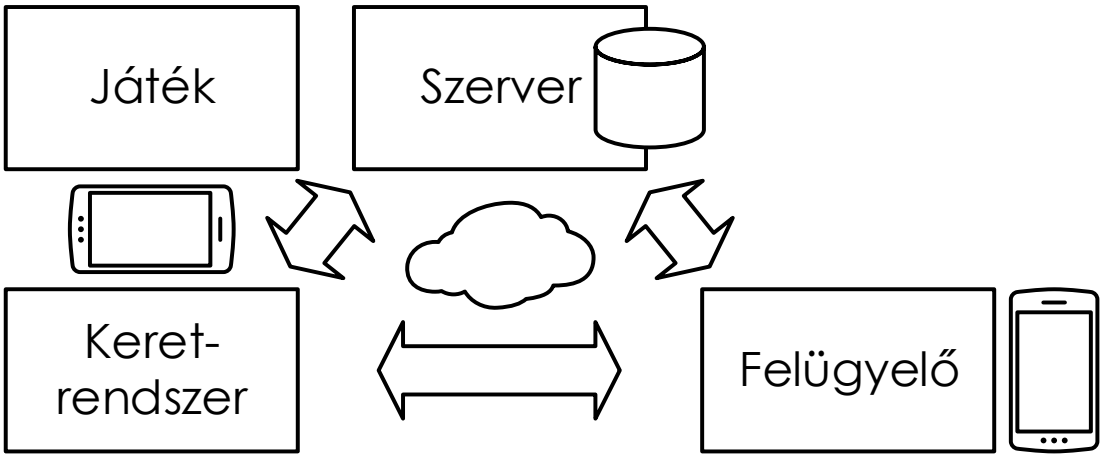
3.2. ábra A felügyelő alkalmazás felhasználói felülete

5. Innentől kezdve folyamatos a kommunikáció a felügyelő alkalmazás és a keretrendszer között, ezáltal a pedagógusnak lehetősége nyílik többek között a játék elindítására, megállítására, és a nehézségi szint módosítására, miközben néhány másodpercenként frissülő képernyőképen kísérheti nyomon a tanulók játékmenetét.
6. A szenzorok a játék folyamat közben adatokat gyűjtenek, melyből a rendszer a mentális aktivitás mértékére és a felhasználó érzelmi állapotára következtet, mint például a játékos unott, mivel túl könnyű számára a feladat, vagy éppen frusztrált annak nehézsége miatt.
7. Ennek függvényében a következő pálya könnyebb vagy nehezebb lesz, például változik a szavak száma, melyek közül a szinonímákat kell megtalálni, vagy esetleg egy másik feladattípusra vált a rendszer, például párosítás helyett egy szöveg kiegészítését kell megoldani.
8. A mérés befejezésekor a játékos tabletje a mérés során összegyűlt adatokat automatikusan egy szervernek küldi. A szerveren található adatok a következő ábrán látható felületen keresztül érhetőek el.

3.3. ábra A szerver webes felülete

4 A rendszer felépítése

Az alkalmazást tekintve három architektúrálisan jól elkülönülő szerepről beszélhetünk. Az alkalmazás különböző részei különböző eszközökön futnak, de a mérés során folyamatos kommunikációban állnak egymással. A **Hiba! A hivatkozási forrás nem található.**. ábrának megfelelően ez a három szerep a következő: a játékos, a felügyelő és a szerver. A szerveren folyamatosan szinkronizálva vannak a jelenlegi játékos eredményei, játékban végrehajtott akciói, továbbá a szenzorok által gyűjtött adatok. Ezek az adatok egy későbbi adatbányászati kutatásnak is alapjául szolgálhatnak, melyek a szenzorok által mért különböző biológiai folyamatokra utaló jelek értékei, és a felhasználó teljesítménye közti összefüggéseket vizsgálhatják, ezzel nyújtva egy újabb perspektívát az általunk megalkotott rendszer felhasználásának.



4.1. ábra Az alkalmazás architektúrája

A felügyelő alkalmazást az oktatás adott területén járatos pedagógus szakember fogja kezelni. Ő felelős a mérés koordinálásáért, továbbá a jelezheti a lehetséges zavaró körülményeket, melyek elvonhatják a játékos figyelmét. Ez azért szükséges, mert a mért jeleket a külső zavaró körülmény az oktatójátéktól függetlenül befolyásolja. Továbbá lehetőség van a keretrendszer által javasolt nehézségi szint, vagy jutalom felülbírálására is, hiszen egy a gyermeket ismerő, több éves tapasztalattal rendelkező pedagógus esetleg jobban tudja, hogy mi a gyermek igényeinek leginkább megfelelő, amit a keretrendszer később akár figyelembe is vehet az otthoni gyakorlás során.

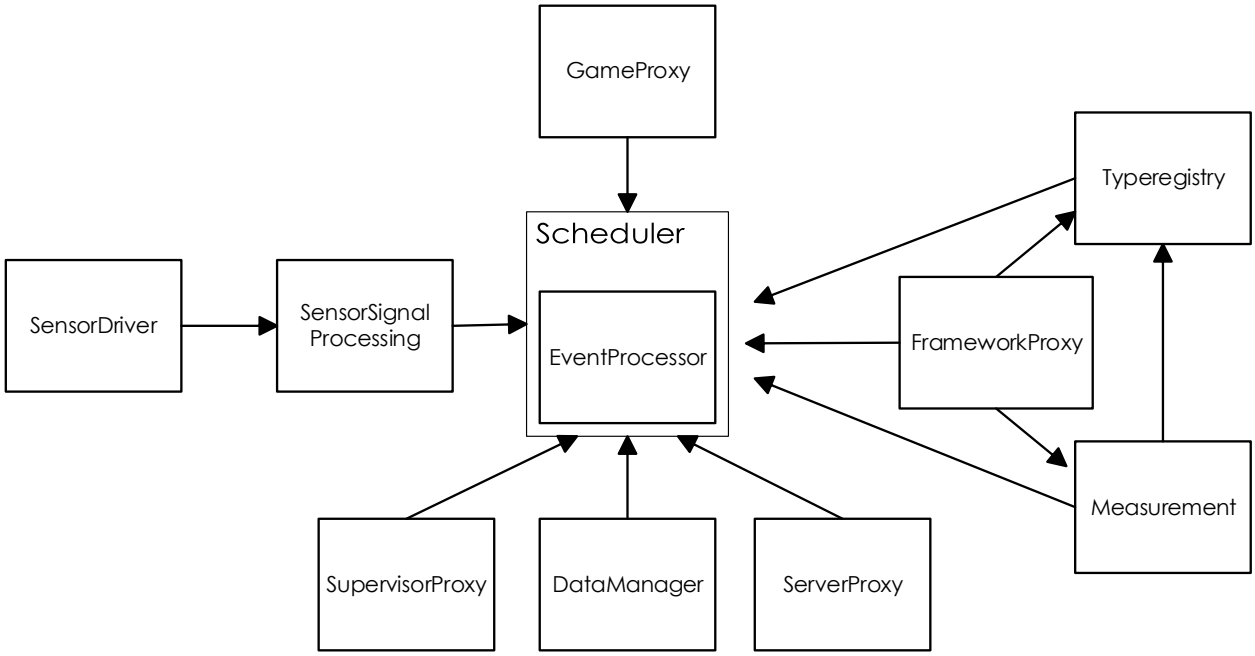
A játékos készülékén fut a játék mellett a keretrendszer, azaz az irányító modul, mely a rendszer többi funkciójának megvalósításáért felelős. Ide tartoznak a javasolt

nehézségi szint és jutalom meghatározásán kívül a szenzoroktól érkező jelek feldolgozása és kiértékelése, a játékoktól érkező adatok feldolgozása, mint például a játék által értelmezett és a felügyelő által kiadható parancsok, vagy a játékban végrehajtott felhasználói események, továbbá a szerverrel történő szinkronizáció, és a felügyelő alkalmazással folytatott kommunikáció megvalósítása is.

Az irányító rendszerkomponens ugyanazon az eszközön fut, mint a játék. Komplexitása miatt több elkülönülő, lazán csatolt komponensre osztottuk, melyek a különböző részfeladatok elvégzéséért felelősek.

4.1 A keretrendszer felépítése

A keretrendszer felépítésének alapvető építőelemeit szolgáltató, különböző funkciókat ellátó modulok meghatározott módon kapcsolódnak egymáshoz.



4.2. ábra: Modulok kapcsolódási hálója

4.1.1 Modulokra bontás

A fent látható modulok dinamikusan, akár futási időben be- vagy kiregisztrálhatók, ezáltal még inkább adaptívvá téve a keretrendszer működését. Ezek a komponensek jól definiált interfészeken keresztül kommunikálnak egymással, így a belső implementáció rejtve marad. A tervezés során fontos szempont volt, hogy a különböző modulok egy általános, közös interfészt valósítsanak meg, melynek köszönhetően a működésük a

konkrét implementációtól függetlenül befolyásolhatóvá válik. Ez lehetővé teszi, hogy egy új rendszerkomponenst könnyen hozzá lehessen adni a keretrendszerhez, ami szükséges lehet új szenzortípus illesztésekor.

Minden egyes modulnak a közös őse az *AbstractModule*, melyen keresztül a fent említett közös funkcionalitást az egyes modulok igénybe vehetik. A közös ős konstruktorában paraméterként adódik át az aktuális ütemező (*Scheduler*) referenciája. A konstruktorban ezen az ütemezőn keresztül történik meg magának a modulnak a beregisztrálása. A modulok beregisztrálása következtében értesülhetnek a keretrendszer által fogadott események halmazáról.

Itt fontos megemlíteni, hogy a moduloknak a fenti folyamat ellentéteként lehetőségük van dinamikusan kiregisztrálni magukat az említett ütemezőnél. Ennek hasznosságára egy példa, mikor a *SuperVisorProxyModule*-nak nem sikerül kapcsolódnia felügyelő alkalmazáshoz. Ekkor kiregisztrálhatja magát, hiszen így számára már nem szükséges, hogy a különböző eseményekről értesüljön.

4.1.2 Belső kommunikáció

Mivel több forrásból származó adat feldolgozása szükséges, ezért az esemény alapú kommunikáció mellett döntöttünk. Esemény keletkezik a játékban végrehajtott cselekvéskor, amikor jutalmat kap a játékos, amikor a felügyelő küld parancsot, amikor egy szenzor jelének feldolgozása befejeződött, stb. A központi esemény-feldolgozás lehetővé teszi a bejövő események ütemezését a prioritásuk alapján. Szinte az összes modul a központi eseménykezelőn keresztül kommunikál egymással, ami tovább csökkenti a közöttük fennálló függőségeket. Minden modulnak van egy dedikált eseménysora, így csak a számukra szükséges eseményeket kapják meg. Események alkalmazásával elkerülhető az aktív várakozás, mert ha egy modul eseménysora kiürült, a feldolgozó szála blokkolódik. A *Visitor* tervezési mintát alkalmaztuk az események feldolgozásához.

Az *AbstractModule* deklarál két változót melyek segítségével a modulok meghatározhatják, hogy mely eseménytípusok azok melyek számukra relevánsak.

E két változó a *Visitor* minta analógiáját követve egy-egy *Visitor* interfész, melynek implementálása a konkrét modul leszármazottakra van bízva. Ez a két interfész az *EventMatcherVisitor*, valamint az *EventProcessorVisitor*.

Az *EventMatcherVisitor* felelőssége, hogy megmondja, hogy egy adott modul mely események keletkezésében érdekelt. Az interfész implementációjában a különböző esemény típusokat paraméterül kapó, túlterhelt függvényeket (*function overload*) kell a megfelelő módon megvalósítani. Jelen esetben ez annyit jelent, hogy az adott implementációnak *true* illetve *false* értékkel kell visszatérnie, ha az adott esemény irányában érdekelt, illetve ha nem.

```
@Override
     public boolean matchesEvent(ConcreteEventA event) {
     return true;
}
```

Ha visszatérési érték igaz, akkor az adott esemény bekerül az adott modul feldolgozási sorába.

Mivel az egyes eseménytípusok száma a fejlesztés során folyamatosan bővült, ezért a könnyebb kezelhetőség érdekében létrehoztunk két könnyebben felhasználható implementációt az említett interfészek helyettesítésére (*AbstractFalseEventMatcherVisitor, AbstractTrueEventMatcherVisitor*), melyek implementálják a fenti metódus összes változatát, ugyanakkor egységesen vagy csak *true*, illetve vagy csak *false* értékkel térnek vissza. Így saját leszármazott megvalósításakor csak azokat a metódusokat kell felüldefiniálni, melyek eltérnek az előre definiált értéktől.

A másik lehetőségként adódhatna, ha egy *if-else* struktúrájú függvény mindegyik modulra vonatkozóan megmondaná, hogy az adott esemény bekerüljön-e a feldolgozási sorába, vagy sem. A *Visitor* minta implementációja révén, a hasonló kérdés eldöntésére *function overload*-ot használunk. Ennek ebben az esetben az az előnye mutatkozik meg, hogy *function overload* esetén már fordítási időben eldől, hogy melyik metódus fog meghívódni az adott paraméter változó statikus típusa alapján, így nincs szükség futási idejű kiértékelésre. Ez az *if-else* szerkezetről nem mondható el. Ha sok esemény érkezik, akkor a *function overload*-nak könnyen kihasználhatjuk teljesítménybeli előnyeit.

A másik említett *Visitor* interfész az *EventMatcherVisitor*, melynek megvalósítása révén az általunk feliratkozott eseményeket kaphatjuk meg az egyes megvalósított eseménykezelő függvényeken keresztül. Ezek az események már az egyes modulokhoz tartozó eseménykezelő sorból kerülnek ki.

```
@Override
public void visitEvent(ConcreteEventA event) {
     // ...
}
```

A modulok az inicializálásuk végeztével elindítják a számukra dedikált eseménykezelő szálat.

A szál folyamatosan figyeli az adott modul eseménykezelő sorát, és ha abban esemény szerepel, akkor meghívja vele a modul saját, már korábban említett EventProcessorVisitor implementációját.

```
eventHandlerThread = new Thread() {
     public void run() {
          try {
               while (true) {
                    Event nextEvent = AbstractModule.this.scheduler
                         .getNextEventForModule(AbstractModule.this)
               nextEvent.acceptEventProcessor(eventProcessorVisitor);
               }
          } catch (InterruptedException e) {
               // ...
          } catch (NoSuchModuleException e) {
               // ...
          }
     };
};
```

A fenti kódot tekintve fontos megemlíteni, hogy az egyes modulokhoz tartozó eseménykezelő sorok implementálást tekintve az olvasás művelet során blokkolódnak, ha nincs visszaadható eredményük. Ennek megvalósítására a Java generikus osztályát, a *LinkedBlockingDeque* osztályát használtuk fel, mely funkcionalitását tekintve kielégíti a fent leírt elvárásokat.

A fent leírtakon túl az *AbstractModule* szolgáltat még egy *onEvent* eseménykezelő függvényt leszármazottai számára, melyen keresztül maguk a modulok tudnak eseményt generálni, és azt átadni a feldolgozási sornak. Ennek feldolgozása pedig már a fent leírtak alapján veszi kezdetét.

4.1.3 Az ütemező

Az ütemező használatával megakadályozható, hogy a kevésbé fontos események előbb kerüljenek feldolgozásra, mint a fontosabbak. A trigger esemény akkor fordul elő, amikor a játékos jutalmat kap a játéktól. A trigger esemény miatt dinamikusan kell emelnünk a szenzorok feldolgozási rátáját, hogy biztosan detektálhassuk a jelet. Érthető,

hogy ha a trigger esemény feldolgozása nem történik meg azonnal, akkor a jel elkapásának az esélye jelentősen csökken.

Hasonlóképpen a felügyelő alkalmazás által generált események magasabb prioritással bírnak. Ezek az események jelzik, ha külső események befolyásolták a mérési folyamatot. Ezt az eseményt is időkritikusnak tekintjük, mert szükséges és elvárt, hogy a hibás információ a lehető leghamarabb kikerüljön a számítási folyamatból, és ne befolyásolja a mérési folyamatot a továbbiakban. Az alkalmazás úgy lett tervezve, hogy lehetővé tegye az ütemezési algoritmusok közötti könnyű váltást a Strategy tervezési minta segítségével, fenntartva a lehető legmegfelelőbb ütemezési stratégia kiválasztását.

4.2 Mérő modul

A mérő modul felelős a nehézségi szint kiszámításáért és a jutalom tulajdonságainak megválasztásáért esemény típusonként. A számítások alapjául három forrásból származó események szolgálnak. A játék a felhasználó teljesítményéről küld eseményeket, azaz a felhasználó által végrehajtott akciók milyen közel vannak a tökéletes megoldáshoz. A szenzoroktól származó adatokból a jutalom által kiváltott öröm nagyságára következtethetünk, ami lehetővé teszi a rendszernek a felhasználó számára legmotiválóbb jutalom megtalálását. A rendszerünk gondoskodik az EEG eszköz és a tablet összekapcsolásáról, az EEG jelek feldolgozásáról és értelmezéséről, továbbá a markerként említett agyi aktivitások (mint a P3A, theta hullámok) sikeresen azonosításra kerültek. A felügyelő érvényteleníthet bizonyos szenzori méréseket, ha külső tényezők befolyásolták a játékost, így ezek nem fogják megváltoztatni az eredményeket.

4.3 A keretrendszer kommunikációja

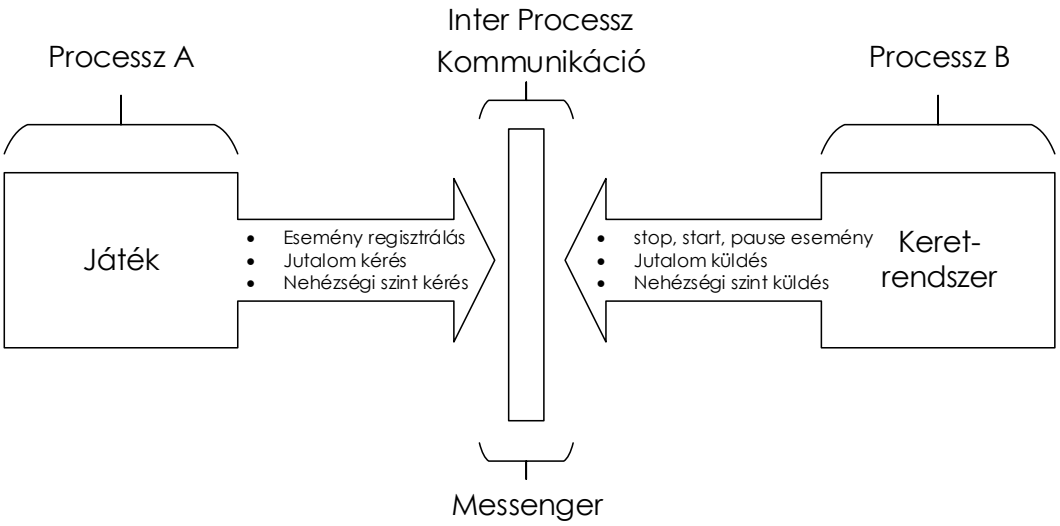
A keretrendszernek több különálló résszel kell aktívan fenntartani a kapcsolatot, melynek köszönhetően a kommunikáció csatornáknak stabil működést kell megvalósítaniuk minden irányba.

4.3.1 Keretrendszer és kliens közötti kommunikáció

A keretrendszer valamint a kiajánlott funkcionalitását használó oktató alkalmazás külön processzben futnak egy adott eszközön. Ezt az implementációs döntést az tette indokolttá, hogy ha egy adott eszközön több oktatóalkalmazás is telepítve lenne (például

külön oktatási célra dedikált eszköz), akkor ezek közösen ugyanahhoz a szolgáltatáshoz kapcsolódjanak.

4.3.1.1 *Inter processz* kommunikáció megvalósítása



4.3. ábra: Processzek közötti kommunikáció

Mivel maga a keretrendszer külön processzben fut, ezért szükséges, hogy megfelelő módot találjunk az *Inter Process* kommunikáció kialakítására és megvalósítására. A különböző folyamatok közötti kommunikáció miatt különösen fontos a hatékony megvalósítás. A könnyű felhasználhatóságot szem előtt tartva a Wrapper tervezési mintát alkalmaztuk, így a keretrendszerrel történő kommunikáció megvalósításának részletei teljesen rejtve maradnak.

Erre többféle lehetőség is kínálkozik *Android* platformon. Ezek közül az egyik lehetőség, az *Intent*-ekkel való kommunikáció[1].

Ebben az esetben az egyes processzek *Intent*-eken keresztül lennének képesek kommunikálni egymással, mely lényegében egy aszinkron üzenetküldési fogadási és válaszadási mechanizmusnak lenne betudható. Az egyes processzek rendelkeznének olyan modulokkal melyek megfelelő *IntentFilter*-ek segítségével az előre definiált *Action*-okre lennének feliratkozva, így esemény alapon értesülnének új üzenet érkezéséről. Mivel az

[1] Az *Intent*-ek az *Android*-on használatos kommunikációs elemek, melyek absztrakt leírását tartalmazzák egy végrehajtani kívánt műveletnek ezáltal futási idejű kapcsolat megvalósításának lehetőségét kínálva a különböző alkalmazás komponensek között.

Intent-ekbe saját adatot is bepakolhatunk, így könnyen azonosíthatóvá tehetnénk az üzenetfolyamokat megfelelően generált *id* segítségével, így az egyes üzenetekre kapott válasz azonosítása sem jelentene gondot.

Az egyedüli probléma a fenti megoldási ötlettel, hogy az *Intent*-ekkel való kommunikáció nem tekinthető gyorsnak, főleg nem akkor, amikor sok üzenet érkezik egyszerre.

A fenti probléma áthidalására, *ServiceConnection* objektumokat használunk.

A folyamat úgy épül fel, hogy az *Activity* az inicializációs szakaszában *bindService()* metódusa segítségével kapcsolódik a *Service*-hez. Fontos, hogy itt a megfelelő *flag*-et használva tegyük ezt meg, hiszen ha a *Serivce* még nem létezik, akkor hozzuk is létre egyúttal. A fenti metódusnak ezenfelül átadunk még egy interfész implementációt melyben a kapcsolódás folyamatához tartozóan lekezeljük a különböző állapotokat (kapcsolt megszületése, kapcsolat megszakadása)

A kapcsolat létrejöttekor kapunk egy *IBinder* interfész implementációt, melynek felhasználásával inicializálhatjuk saját *Messenger* objektumunkat.

A *Messenger* egy *Handler* implementációt vár paraméterül, melynek *handleMessage()* függvénye segítségével tudjuk lekezelni a kapott üzenetet. *Android*-on egy processzen belül többek között hasonló mechanizmus alapján történik a *UI* szál és a háttér szálak közötti kommunikáció megvalósítása is. Legtöbbször akkor szokták használni, mikor egy *UI* elemet szeretnénk módosítani a háttérszálból, ezt ugyanis közvetlenül nem tehetjük meg, hiszen az adott *View* elemet csak az őt létrehozó szálról engedélyezett módosítani.

Miután létrejött a kapcsolat, küldünk egy „*HELO*” üzenetet a *Service*-nek azzal a célzattal, hogy a kétirányú kommunikációt biztosítsuk. Eddig a pontig ugyanis a játék oldalon csak üzenetet tudunk fogadni a *Service*-től. Szükséges azonban, hogy a kommunikáció mindkét irányba biztosított legyen.

A fent említett inicializálási üzenetben a fenti okból kiindulva fontos, hogy beállítsuk a *replyTo* mező értékét saját Messenger implementációnkra, melyen keresztül a Service képes lesz az ellenkező irányba üzenetet küldeni.

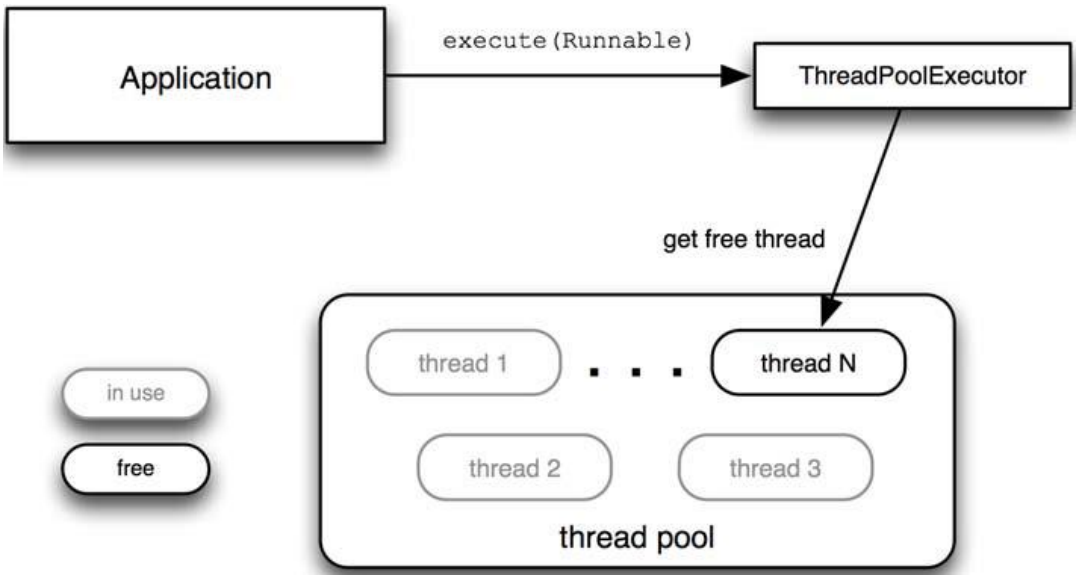
Ha kliens oldal inicializálási üzenetére válasz érkezett, akkor ez egyenértékű a keretrendszer inicializálási folyamatának végével, így megkezdődhet a típusok

beregisztrálásának folyamata, valamint ekkor már rendelkezésünkre áll egy kétirányú kommunikációt biztosító csatorna.

4.3.1.2 Üzenetek fogadása keretrendszer oldalon

A korábban említett indokokat és igényeket figyelembe véve és szem előtt tartva az üzenetek fogadását illetően megfelelő irányvonalnak bizonyult az üzenetek feldolgozásához külön szálak rendelése, így párhuzamosítva a feldolgozás folyamatát. Ennek kivitelezéséhez a *thread pool* használatát találtuk a legmegfelelőbb választásnak. A megfelelő implementáció megvalósítása következtében a beérkező üzeneteket gyorsan tudjuk lekezelni, elkerülve ezzel az esetleges feldolgozási / üzenet sorba (*message queue*) kerülést és az esetleges ottani feltorlódást.

A tervezés és kivitelezés során többféle *thread pool* implementációt is számításba vettünk és megvizsgáltunk a megfelelő és leginkább alkalmas megvalósítása kiválasztása érdekében.



4.4. ábra: A *thread pool* működése a gyakorlatban. Az *Application* átad egy *Runnable*-t az *Executor*-nak végrehajtásra. Ezt követően az *Executor* a *pool*-ból allokál egy szálat a feladat végrehajtására.[19]

A *FixedThreadPool* előre meghatározott számú szálat hoz létre a *pool*-ban, melyeket felhasznál a kapott *task*-ok végrehajtására. Ha mindegyik szál foglalt és éppen használatban van, és eközben új *task* érkezik be, akkor a beérkező feladatok egy várakozási sorba kerülnek, egészen addig, míg fel nem szabadul egy őket kiszolgálni nem képes szál.

A *ScheduledThreadPool* implementációját tekintve hasonló alapokon nyugszik, mint a fent említett megvalósítás, azzal a különbséggel, hogy a végrehajtást megadott késleltetéssel ütemezni tudjuk.

A *SingleThreadPool* lényegében megegyezik azzal a megvalósítással, amit akkor kapnánk, amikor a *FixedThreadPool* inicializálása során 1-et adnánk meg a *pool* méretnek.

Jelen helyzetben a fenti implementációk közül egyik sem kínált kielégítő megoldást a fentebb vázolt igényeinkre. A *CachedThreadPool* ellenben előre nem definiált számú szállal inicializálódik, így pontosan akkor hoz létre szálakat, amikor azokra szükség van. Azok a szálak, melyek végeztek a rájuk bízott feladat végrehajtásával, visszakerülnek a pool-ba, és újabb beérkező task-ok kiszolgálására válnak így alkalmassá[2]. Ezzel a tulajdonsággal jelentős erőforrás megtakarítást érhetünk el (mobil eszközökön ez egyáltalán nem nevezhető utolsó szempontnak), hiszen megspóroljuk egy adott szál létrehozásával és megszüntetésével járó, amúgy sem erőforrás takarékos műveleteket.[19]

A *CachedThreadPool* használata során, a tulajdonságaiból fakadóan sok, nagy mennyiségű, rövid ideig tartó, aszinkron feladat elvégzése esetén érhetünk el jelentős teljesítménybeli előrelépést. Nekünk pont erre van szükségünk.

A keretrendszer inicializálása során a fent említett implementációt használjuk, valamint minden egyes beérkezett üzenet feldolgozását egy külön szálra bízzuk, melyeket az említett pool-ból allokálunk.

```
ownMessenger = new Messenger(new Handler() {
     @Override
     public void handleMessage(Message msg) {
          executor.execute(new MessageReceiver(msg));
     }
});
```

A *MessageReceiver* osztály egy saját osztály mely implementálja a *Runnable* interfészt, melynek következtében átadható az *execute()* függvénynek, mint végrehajtható feladat.

A *MessageReceiver* implementációját tekintve fontos említést tenni a konstruktorban paraméterül kapott *Message* objektum kezelésének miként jéről. Itt ahelyett,

[2] Itt érdemes megjegyezni, hogy a *pool*-ba visszakerült szálak 60 másodperces inaktivitás utána terminálódnak, elkerülve ezzel a *pool* túlságosan nagyra hízását.

hogy csupán a referenciát eltárolnánk az említett objektumra vonatkozóan, érdemesebb az egyes minket érintő paraméterekről másolatot készíteni, és azokat elraktározni erre a célra deklarált változók formájában.

Erre azért van szüksége, mivel az *API* dokumentációra támaszkodva a *Message* objektumok, drága erőforrás lévén, egy erre célra kialakított *object pool*-ból allokálódnak. Mivel mi több szálon dolgozzuk fel a beérkezett üzeneteket, ezért előfordulhat, hogy amikor a *Message* objektumból ki szeretnénk olvasni a számunkra releváns adatokat, addigra az adott objektum már újra felhasználásra került, így inkonzisztens adatokhoz juttatva minket. Ez a probléma a fent említett módon áthidalható.

A fentiekből adódóan annak ellenére, hogy a *Message* osztály rendelkezik publikus konstruktorral, érdemes, és ajánlott ahelyett a statikus obtain() függvényét használni, mely az említett *object pool*-t használja fel a kért objektum szolgáltatására.[16]

Másik fontos implementációs probléma, hogy az inter proecess kommunikációból adódóan ha az üzenetek valamilyen szerializált objektumot tartalmaznak, akkor annak betöltésért felelős *ClassLoader*-t explicit meg kell adnunk, ellenkező esetben *ClassNotFound* kivételt kaphatunk.

4.4 Adatkezelő modul

Az adatkezelő modul felelős a hálózati kommunikáció lebonyolításáért a felügyelő alkalmazással és a szerverrel, továbbá utóbbi esetében az adatok cache-eléséért is. Erre azért van szükség, mert azt szeretnénk, ha az eszköz erőforrásait a keretrendszer algoritmusai használnák, és nem pedig a szerverrel történő szinkronizáció, hiszen a szerveren nem szükséges valós idejű mérési adatok tárolása. Ezen felül ennek a modulnak a feladata annak eldöntése, hogy melyik eseményt kell a szervernek, és melyiket a felügyelőnek elküldeni.

4.4.1 Lokális cache

Ennek megvalósításához az Android platformon elérhető SQLite adatbázist választottuk. Segítségével könnyedén megvalósítható a cache-elés implementálásához szükséges funkcionalitás. Mivel a szerver és a keretrendszer közötti kommunikáció tervezésekor fontos szempont volt, hogy az adatok reprezentálása platformfüggetlen módon történjen a minél lazább csatolás elérése érdekében, ezért a JSON alapú adatreprezentáció mellett döntöttünk. Ahhoz tehát, hogy a keretrendszer által használt

objektumokat az SQLite adatbázisba menthessük, szükséges lenne valamilyen objektum-relációs leképezés alkalmazása, majd visszaolvasáskor ennek inverzét alkalmazva tudnánk előállítani az objektum JSON-né konvertált reprezentációját. Ez feleslegesen bonyolítaná a cache-elési folyamatot és növelné a komplexitást, ezért ennek a folyamatnak a leegyszerűsítése érdekében azt a megoldást választottuk, hogy az SQLite adatbázisba már eleve a JSON-né konvertált objektumot mentjük, melynek kulcsa esemény típusú adatok esetében egy integer ID, típusleírók esetében pedig a könnyű beazonosíthatóság érdekében az adott típusleíróhoz tartozó típus neve. A text típusú elsődleges kulcs használata nem okoz jelentős teljesítményromlást, hiszen ennek a táblának a tartalma csak a játék által használt eseménytípusok és jutalomtípusok leíróit tartalmazza, melyek száma nem jelentős.

Az adatok szerverre történő feltöltését egy külön Android Service végzi, mely akkor kerül elindításra, ha a keretrendszert futtató Service befejezte a működését. Ha ekkor valami miatt nem sikerülne az adatok feltöltése a központi adattárba, például megszakad az internet kapcsolat, vagy lemerül a készülék, akkor a későbbiekben újra próbálkozik bizonyos feltételek bekövetkezésekor. Az egyik ilyen alkalom a készülék bootolása, valamint Wi-Fi hálózathoz történő sikeres kapcsolódás, továbbá ha a készülék bekapcsolt állapotban van, akkor néhány óránként mindenképpen megpróbálja feltölteni a cache-ben tárolt adatokat. Ha minden adat feltöltésre került, akkor a lokális cache tartalmát törli a rendszer, ezzel biztosítva, hogy feleslegesen ne foglalja a mobil eszközök esetén sokszor szűkös tárhelyet.

A fenti működést megvizsgálva viszont jogosan adódik a kérdés, hogy hogyan biztosított az adatbázis konzisztenciája, ha egyszerre futna az adatfeltöltő és a keretrendszert futtató Service. Ezt a problémát orvosolandó mindkét Service futásának indulásakor megvizsgálja, hogy nem fut-e a másik. Ha mindkettő fut, akkor természetesen a keretrendszernek van elsőbbsége, ilyenkor az adatfeltöltő service az általa beregisztrált BroadcastReceiver-en keresztül értesül a keretrendszer indulásáról, majd leállását követően szintén egy broadcast üzenet küldésével értesíti a keretrendszert, hogy elkezdheti az adatbázis használatát.

4.4.2 Keretrendszer és a felügyelő alkalmazás közötti kommunikáció

A keretrendszer és a felügyelő alkalmazás közötti együttműködés érdekében szükséges, hogy valamilyen módon egymással képesek legyenek kommunikációt folytatni.

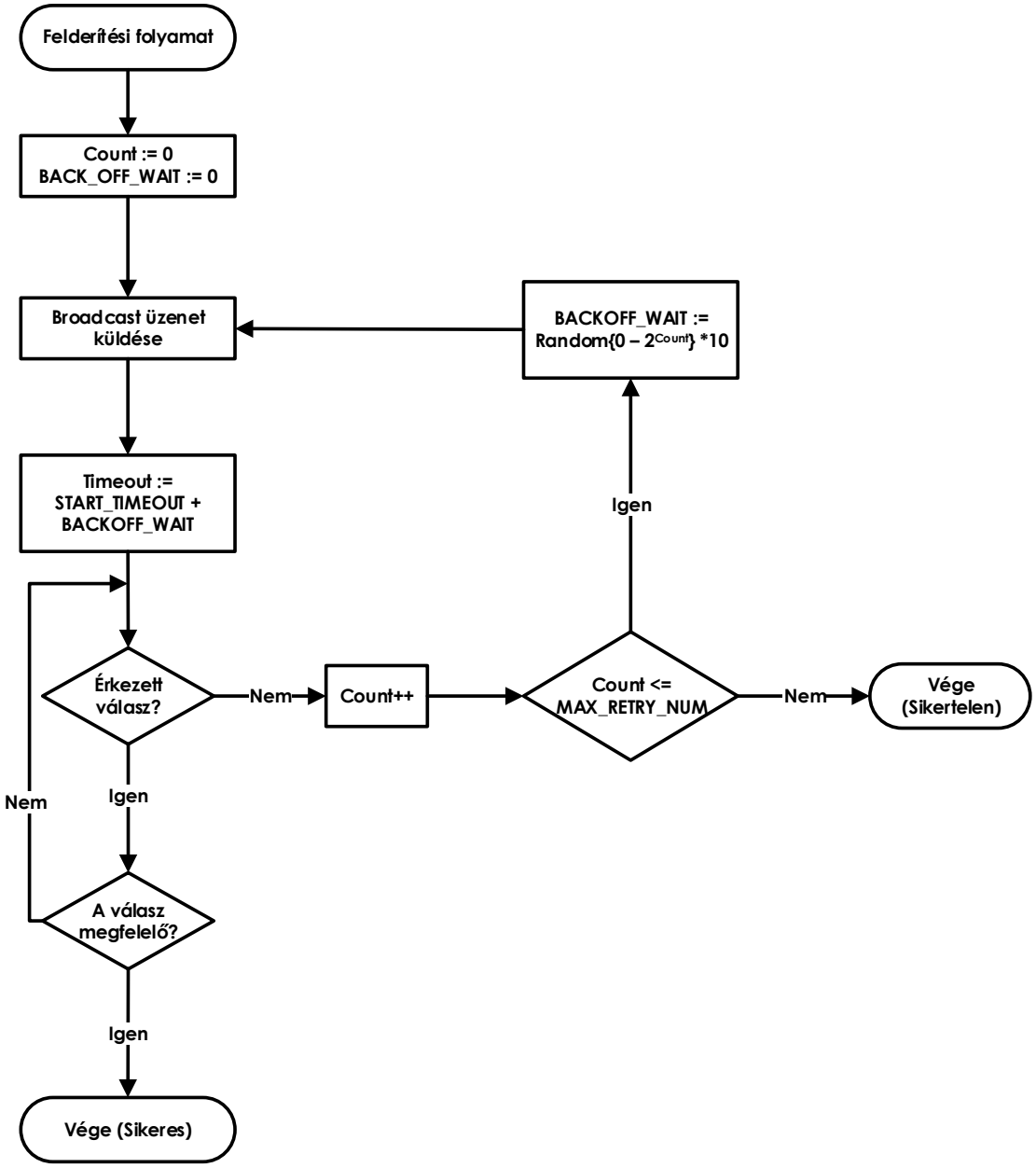
Első lépésként a kapcsolat-felépítés módját szükséges definiálnunk a két kommunikációs végpont között. Annak érdekében, hogy felesleges konfigurációs beállításokat ne kelljen kieszközölni a keretrendszer oldalon, valamint a keretrendszer egyszerű használatát szem előtt tartva, a kapcsolat felépítés módjául a hálózat automatikus felderítését választottuk.

A felderítés kezdeményezője a keretrendszert futtató eszköz. A keretrendszer a felügyelő eszköz megtalálására létrehoz egy *Datagram Socket*-et, melyen keresztül a későbbiekben az *UDP* csomagok kerülnek kiküldésre a felderítés véghezvitele érdekében.

Az említett *socket broadcast* üzenetek formájában küldi ki a *datagram* csomagokat az előre definiált *port*-ra, az előre egyeztetett felderítési üzenettel. A tervezés során definiáltak alapján a felügyelő rendszer, ha *listening* állapotban van, akkor ezekre a felderítő csomagokra vár, majd ezekre válaszol.

A felderítés folyamatáról, az újrapróbálkozások számáról, illetve a *timeout*-ok változtatásáról egy állapotgép gondoskodik, hogy a kiküldött üzenetek száma és a vakozási idő optimális értékek között mozogjon.

Az állapotgép alapját egy *exponenciális backoff* stratégia adja. (Felépítésé tekintve hasonló az *Ehternet MAC* protokolljához)



4.5. ábra: Felderítési folyamat állapotgépe

Miután sikeresen megkaptuk az *ACK* válasz üzenetet a felderítő csomagra, akkor kezdetét veszi a kapcsolatkialakítás második fázisa. Ennek elején felszabadítjuk az UDP *socket*-et, mivel már a második fázisban nem lesz rá szükség, hiszen itt már TCP alapú *socket* kommunikációt valósítunk meg.

Az első fázisban kihasználtuk az UDP jellemzőit, azaz hogy segítségével könnyen tudtunk *broadcast* üzeneteket küldeni, valamint a nem garantált csomag célbaérkezés illetve esetleges csomagelvesztés nem jelentkezett zavaró tényezőként. A második szakaszban, mikor már a kommunikáció résztvevői ismertek, már a megbízhatóbb kapcsolat orientált TCP megvalósításra támaszkodtunk.

Az inicializáció részeként két *stream*-et hoztunk létre, egy *object* output és egy *object* input *stream*-et, melyek a *tcp socket* input és output *stream*-jeivel inicializálódtak, és ennek segítségével biztosítják a kommunikáció be illetve kimenő ágát.

Ha minden helyesen lezajlott, akkor a következőkben egy külön szál felelősségévé válik, hogy az *object input stream*-et figyelje. (A *stream readObject()* függvénye blokkolódik így elkerülhetjük az aktív várakozást.)

4.4.2.1 Üzenetek fogadása

A hálózaton való kommunikáció során szerializált objektumokat küldünk át a kommunikáló felek között. A megoldandó probléma ebben az esetben az, hogy egy szerializált objektum deszerializálása után detektáljuk annak konkrét típusát.

Az objektum szerialiálás során az osztály meta-adat információk is szerializálódnak, így a lehetséges objektum típusok ismeretében valamint az *instanceOf* operátor segítségével kikövetkeztethetjük a kapott objektum típusát, és ezt követően az adott típusra kasztolás után vissza is kaphatjuk azt.

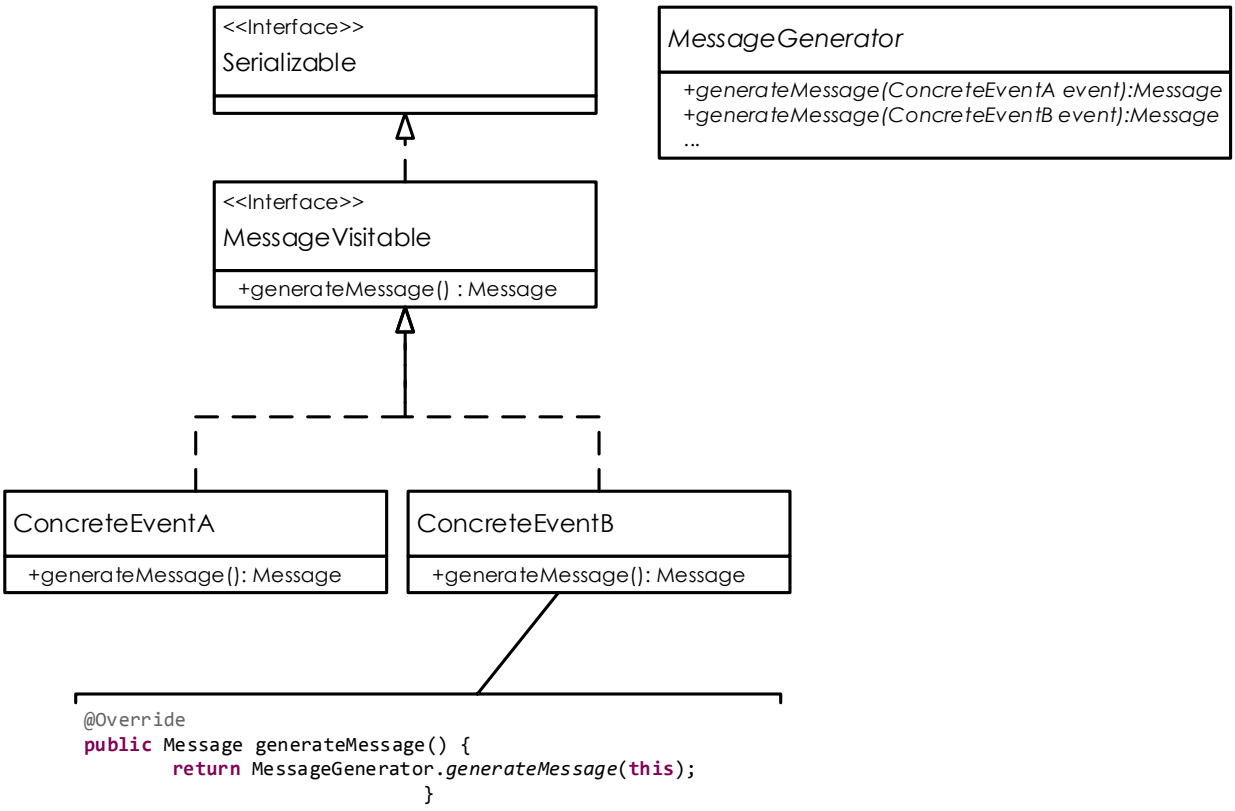
A hálózaton átküldött *Message* objektum két fontos paraméterrel rendelkezik. Ezek egyike egy *Object* változó, maga az üzenet tartalma, a másik pedig egy *byte*, mely felhasználását tekintve akkor válik hasznossá, ha csak magának az üzenet típusának van információ közlő szerepe.

A *Message* osztályt még elláttuk egy *generateObject()* nevű függvénnyel, mely a paraméterül kapott *Message* objektum alapján visszaadja a konkrét tartalmazott objektumot a megfelelő típusra kasztolva. Ehhez szükséges a második paraméterként átadott *listener* objektum, melynek az adott típusú paraméterrel túlterhelt metódusa fog meghívódni (*onReceiveObject(...)*).

Az így kézhez kapott objektum segítségével már meghívhatjuk az *AbstractModule*-tól örökölt *onEvent()* eseménykezelő metódusunkat és átadhatjuk neki a kapott objektumot.

4.4.2.2 Üzenetek küldése

A kommunikáció kétirányúságának biztosítása végett szükséges, hogy a felügyelő eszköz irányába is tudjunk üzeneteket / *object*-eket küldeni. A különböző típusú objektumok kezelésére és belőlük üzenet generálása a *Visitor* tervezési mintát használtuk fel.



4.6. ábra: Üzenetküldés során alkalmazott Visitor minta adaptációja

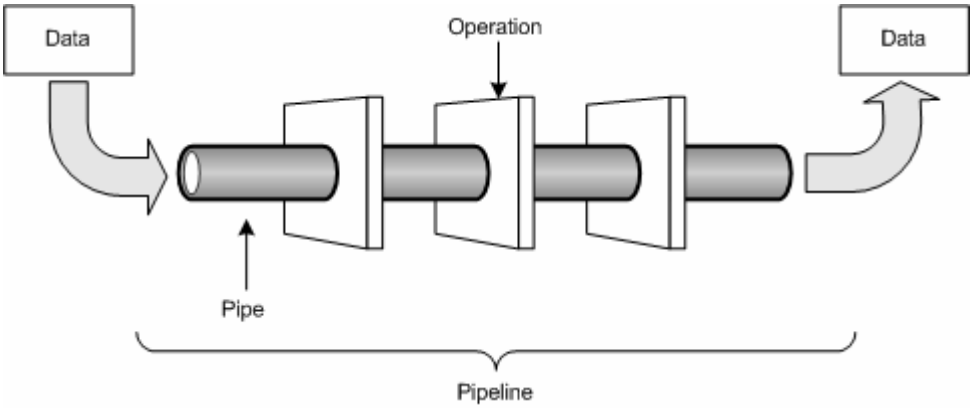
Minden eseménytípus, mely átküldhető üzenet formájában meg kell, hogy valósítsa a *MessageVisitable* interfészt. Ezzel együtt magát a *Serializble* interfészt is megvalósítja, mivel a *MessageVisitable* interfész ezt kiterjeszti. Az egyes konkrét esemény típusok a *generateMessage* függvényükön belül meghívják önmagukat átadva paraméterül a *MessageGenerator*-nak, ahol a *function overload*-nak köszönhetően a megfelelő típust lekezelni képes *generateMessage()* függvény fog meghívódni. A *Visitor* mintával analógiában a *MessageGenerator* feleltethető meg magának a *Visitor*-nak.

A későbbi bővíthetőséget szem előtt tartva, valamint a kommunikációs rétegtől való függetlenséget előnyben részesítve a *MessageGenerator* függvényeit egy megfelelő *interface*-be helyeztük el, így azt konkrét *MessagGenerator*-ok implementáció függően valósíthatják meg, így lehetőséget biztosítva különböző kommunikációs csatornától függően, különböző *Message* objektumok generálására.

4.5 Szenzorkezelő modul

A szenzorkezelő modul esetében a többlépéses jelfeldolgozás ötlete merült fel. Ennek a megvalósításához a Pipes and Filters tervezési minta került alkalmazásra. Így

képesek vagyunk a különböző szűrőket egymás után alkalmazni, ezzel biztosítva a jelfeldolgozás lépéseinek testreszabhatóságát. Ez azért is fontos, mert az algoritmusnak konfigurálhatónak kell lennie egy trigger esemény által, így amikor az adott jel előfordulási valószínűsége nagyobb, könnyebb lesz felismerni.



4.7. ábra: Pipes and Filters absztrakt ábrázolása[20]

Még mindig probléma lehet, hogy az alkalmazásnak időkritikus eseményekre is válaszolnia kell. Ez azt jelenti, hogy az agy a jutalom megkapása után 300 ms-al reagál, ami a P300 agyhullám kibocsátását jelenti. Mivel a rendszerben egyszerre relatív nagyszámú esemény van, ezért a first-come-first-served metódus nem alkalmazható, hiszen így az időkorlátok miatt könnyen lemaradhatnánk releváns események feldolgozásáról. A probléma kiküszöbölése érdekében, bevezettük az események priorizálását, és ennek eredményeképpen az ütemezőt.

4.5.1 Szívritmus jelek

Az ergonómiában a szoftver könnyű használhatósága a szoftver minőségének egyik dimenziója. Minél könnyebben használható egy szoftver, annál kisebb mentális terheltség jelentkezik a szoftver használata során. Ennek mérésére a szubjektív kérdőívek mellett objektív módszereket is bevezettek, melyek közül az egyik a szív ritmus variabilitás – angolul Heart Rate Variability (HRV) – teljesítmény spektrumának a vizsgálata [14]. Mivel mentális terheltség nem csak egy nehezen használható szoftver esetében jelentkezik, hanem az oktató játékok tanító feladatainak megoldása során is, ezért ezeket az eredményeket felhasználva a keretrendszer által a felhasználó kognitív állapotára adott becslések is tovább pontosíthatóak. Így jobban becsülhető a játékos fejlődését elősegítő optimális nehézségi szint is, mely folyamatos kihívást jelent.

Az EEG-vel szemben a szívritmus mérő eszköz előnye hogy olcsóbb, egyszerűbb felhelyezni, hiszen nincs szükség semmilyen gélre, csak egy pántot kell a mellkasra csatolni. Nehezebben is tud elmozdulni, hiszen az EEG-nél már egy fejrázás vagy egy bólintás is elég lehet, hogy az érzékelők elmozduljanak a helyükről. Természetesen az EEG által szolgáltatott adatok sokkal több információt hordoznak a felhasználó kognitív állapotáról, mint egy szívritmus mérő, ezért mindkét eszköz keretrendszerhez illesztése indokolt. A keretrendszer működéséhez nem szükséges mindkettő (sőt egyik) jelenléte sem, azonban ezek szolgálnak a visszacsatolás alapjául, tehát ezek nélkül pontatlanabbak a becslések.

Néha önmagában a pulzusszámot is vizsgálják, de ennek váltakozása jobban mutatja a mentális terheltséget. Azonban számunkra maga a pulzusszám is érdekes lehet, hiszen stresszhelyzetben emelkedik a pulzus, és könnyen lehet, hogy a tanulási folyamat sikertelenségét éppen a nyugtalanság okozza. Ilyenkor a felügyelő alkalmazást használva a pedagógus a keretrendszer által küldött adatok alapján erről is értesülhet, így pontosabb képet kapva a gyermek állapotáról, és néhány nyugtató szóval vagy a körülményeken változtatva a gyermek számára megfelelőbb környezetet alakíthat ki, ahol kevesebb a számára nyugtalanító tényező, ezzel elgördítve a hatékony tanulást útjában álló akadályokat.

A feladatok okozta nehézséget viszont jobban mutatja a HRV, vagy az abból származtatott és még kifejezőbb szívdobbanások között eltelt idő váltakozása – angolul Heart Period Variance (HPV). Számos tanulmány megmutatta, hogy a mentális terhelés növekedése csökkenést okoz a HPV teljesítményspektrumának középfrekvenciás (0.07 – 0.15 Hz) tartományában. A 0.15 – 0.45 Hz tartomány a légzésgyakorisággal van összefüggésben, a 0.04 – 0.07 Hz pedig a hőszabályozási ingadozással korrelál. A középfrekvenciás tartomány teljesítményspektrumát befolyásolhatják a nagyobb mozdulatok, mint például a nevetés, vagy nyújtózkodás. Ilyen esetekben jön jól a felügyelő alkalmazás által biztosított funkció, mely lehetővé teszi, hogy a pedagógus felülbírálja a keretrendszert, így a rendszer által nem érzékelhető és a mérés eredményét befolyásoló tényezők kiszűrhetőek.

A vizsgálathoz 30 másodperces csúszóablakot választottunk, melyet egy másodpercenként léptettünk. Ez már kellő mennyiségű mintát tartalmaz, hogy ablakfüggvény használata mellett a spektrális elemzés pontos eredményt adjon. Ablakfüggvénynek a Hamming-függvényt választottuk, a frekvenciatartományba történő

transzformáláshoz pedig az FFT-t. Ebből több ingyenesen felhasználható C könyvtár is található az interneten, melyek az Android Native Development Kit (NDK) segítségével könnyedén integrálhatóak a keretrendszerbe, a C nyelv pedig lehetővé teszi az algoritmus gyors futását. Mivel a szívritmus mérő szenzort illesztő modul csak egy jól meghatározott interfészen keresztül használja a jelfeldolgozó komponenst, ezért az a későbbiekben még hatékonyabb algoritmusra cserélhető, például wavelet-eken alapulóra, anélkül, hogy a konkrét algoritmus implementációján kívül bármit módosítani kellene.

A keretrendszer architektúrájának helyességét bizonyítja, hogy az eszköz illesztéséhez mindössze az eszközzel történő kommunikációt lebonyolító, és az eszköz által előállított jeleket feldolgozó komponensek hozzáadására volt szükség, a keretrendszer többi részét nem volt szükséges módosítani. Ezt az tette lehetővé, hogy a szenzor illesztő modulnak elég a központi eseménysort ismernie, melybe jelfeldolgozás eredményeképpen a magas szintű eseményeket teheti, mint például mentális terheltség alacsony, magas, stb. Ebből az eseményből az ajánlást készítő modul tudni fogja, hogy csökkenteni vagy növelni kell a nehézséget, és adaptívan állapítja meg az esemény által kiváltott változás nagyságát.

4.6 A keretrendszer rugalmassága adaptív környezetben

A keretrendszer tervezésekor igen fontos hangsúlyt kellett fektetni a rugalmasságra, és a különböző külső alkalmazásokkal való kompatibilitás könnyű megvalósíthatóságára és fenntarthatóságára. Ezt a tervezésbeli elvárást arra alapoztuk, hogy a keretrendszernek előre nem ismert oktatójátékokkal kell tudnia együttműködni, mindezt az együttműködést a keretrendszer alapvető működésbeli módosítása nélkül megvalósítva. Ez igen fontos szempontnak tekinthető, mivel a keretrendszer használata igazából akkor nyer értelmet, ha annak funkció bázisát több független fejlesztő is fel tudja használni saját biofeedback technikán alapuló oktatójáték fejlesztése során.

A fenti analógiát folytatva, az adaptív bővíthetőségen kívül fontos, hogy figyelembe vegyük a fejlesztés során a keretrendszer könnyű használhatóságát. Ebben a tekintetben az elsődleges fejlesztési irányvonal az volt, hogy a keretrendszert felhasználó fejlesztőnek minél kisebb mértékben kössük meg a kezét, és a lehető legtöbb szabadságot biztosítsuk számára a fejlesztés során. A célkitűzések között szerepelt még továbbá a keretrendszer *API*-jának könnyű, kényelmes, és magától értetődő használata, mely tulajdonságok tovább növelik annak alkalmazhatóságát.

A tervezéskor meg kellett vizsgálnunk, hogy tulajdonképpen, mik is azok az előre nem ismert paraméterek illetve tulajdonságok, melyek tekintetében a keretrendszert dinamikussá illetve adaptívvá szeretnénk tenni a támogatási lehetőségek fenntartása végett.

Az előre nem ismert paraméterek között négy fontos tényezőt érdemes kiemelni az előzőek alapján. Ezek közül az egyik a játék által használt különböző eseménytípusok halmaza, melyek a játék során keletkezhetnek, befolyásolva magát a játékmenetet. Abban a tekintetben nyílván nem akarjuk korlátozni az oktatójáték fejlesztőit, hogy milyen játékeseményeket használhatnak az általuk fejlesztett alkalmazásban, valamint azokat milyen paraméterekkel látják el. Itt a felmerülő probléma, hogy a fentiek ellenére, valamilyen információt mégis szükséges, hogy kapjunk az említett eseményekről annak érdekében, hogy a keretrendszer számára egy megfelelő input-ként szolgáljon.

Hasonló problémával állunk szemben a játék események kezelésén túl a különböző pályák végén kapható jutalmak generálásakor is. Itt is szeretnénk követni azt az irányvonalat, hogy teljes mértékben a játékra bízzuk, hogy milyen jutalmakat kíván adni egy adott pálya végén, és ennek lehetőségeit tekintve nem szeretnénk korlátok közé szorítani a fejlesztőt, valamint magát az elkészült alkalmazást.

A keretrendszer szolgáltat bizonyos eseménykezelő függvényeket, melyek megvalósítása révén a játék kliens hasznos információkhoz juthat a játék jelenlegi állapotát illetően. A fent említett előre definiált eseményeken túl azonban biztosítani szerettük volna a lehetőséget, hogy saját, a kliens által definiált eseményekre is reagálhasson a játék, ezzel is biztosítva a keretrendszer könnyű bővíthetőségét.

Végül a negyedik tényező az előző ponthoz hasonlóan a kliens és a keretrendszer valamint a felügyelő alkalmazás között kívánja dinamikussá tenni a kommunikációt, viszont a kommunikáció irányát tekintve a fordított irányban. Ennek értelmében biztosítani szerettük volna, hogy a kliens aktívan kezdeményezhessen üzenetküldést a felügyelő alkalmazás irányába, mindezt előre nem ismert paraméterek segítségével.

Ezek miatt az adat réteg tervezésekor általános struktúrákat kellett bevezetnünk. Az alapötlet a következő: az előre nem ismert játékesemények numerikus adatait és az adat típusának leírását külön táblában tároljuk úgy, hogy az érték tábla egy idegen kulccsal hivatkozik a leíró táblára.

Minden játék, az első induláskor egy inicializációs folyamaton megy keresztül, amely során a játék beregisztrálja a saját típusait a keretrendszernél. Ezek a típusok a

keretrendszer oldalán egy típus tárolóban tárolódnak, és dinamikusan linkelődnek a bejövő eseményekhez. Ezzel a módszerrel a memória használat minimalizálható, mivel minden típus leíróból csak egy van jelen a memóriában, függetlenül az ebből az adott típusból fogadott események számától. Egy másik előnye ennek a megközelítésnek, hogy serializálással járó overhead csökkenthető, mert a komponensek közötti kommunikáció során elegendő a típusazonosító elküldése, ami azonosítja a típust a keretrendszer oldalán.

4.6.1 Felhasznált tervezési minták

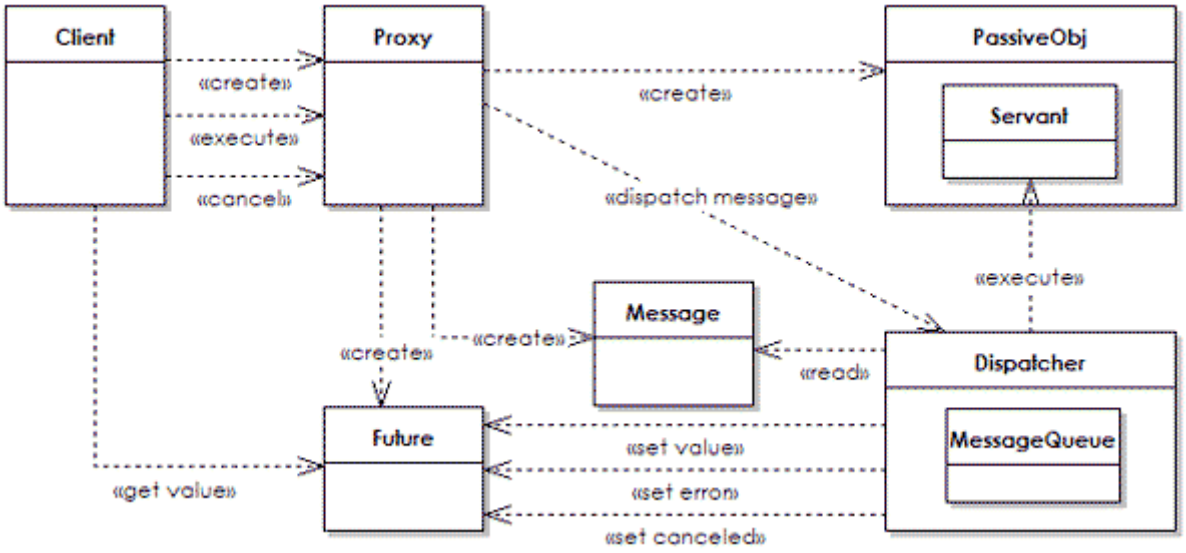
A fent említett problémák leküzdéséhez a tervezési fázisban hatékonyan alkalmaztunk többféle szoftver tervezési mintát. Ezek között szerepel *Proxy* tervezési minta*, Active Object, Singleton, Factory* stb. A következőkben a tervezés szempontjából két hangsúlyosabb mintákat részletezem.

4.6.1.1 Active Object

Az *Active Object* tervezési minta lényege, hogy különválasztja az eljáráshívást az eljárás végrehajtásától. Ha a mindennapi életből szeretnék analógiát találni a minta lényegére, akkor talán a legmegfelelőbb példa a párhuzam szemléltetésére a „pincér az étteremben” hasonlat. A pincérek felveszik a rendeléseket, azonban maga a kiszolgálás ettől függetlenül történik meg, időben akár később, sőt elképzelhető, hogy eltérő sorrendben is, mint ahogy a feladatok felvételre kerültek. Fontos szempont, hogy míg az rendelés el nem készül, addig a vendégek foglalkozhatnak mással, ugyanúgy ahogy a pincérek is. Ezzel szemléltetve a dolog aszinkron mivoltát.[18]

Szoftveres oldalon a fenti problémát röviden úgy lehetne jellemezni, hogy elképzelünk egy többszálú alkalmazást, ahol a termelő és fogyasztó folyamatok kommunikálnak egymással, azonban mindezt úgy megoldva, hogy se a fogyasztó folyamatok, sem pedig a termelő folyamatok ne blokkolják egymás kommunikációját.

Ezt a problémát hivatott megoldani az Active Object tervezési minta.



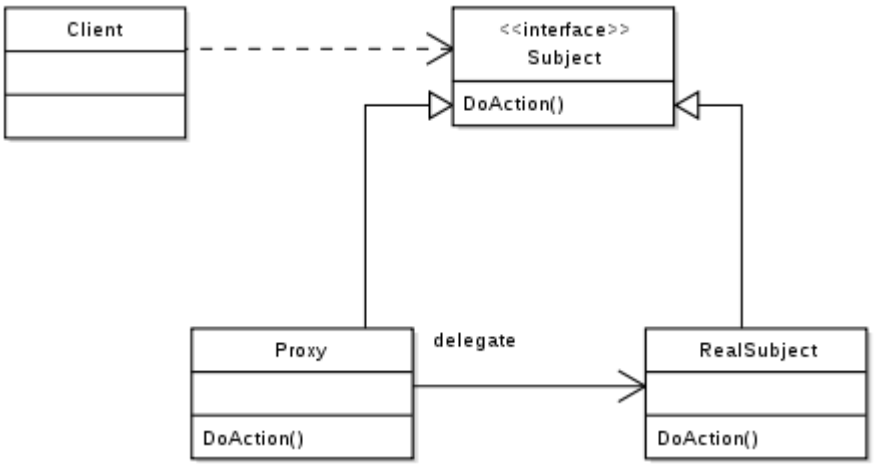
4.8. ábra: Az Active Object tervezési minta[21]

Működését tekintve a minta lényegében a következőképpen működik. A kliens meghívja a proxy egy adott függvényét, melynek hatására a proxy egy *Message*-t generál, mely magát a függvényhívást és annak a paramétereit, kontextusát hivatott reprezentálni. A kliens visszakap egy *Future* objektumot a *proxy*-tól, melyen keresztül lekérdezheti az eredmény esetleges aktuális állapot, vagy értesülhet annak megérkezéséről. A keletkezett *Message* objektum, miután bekerült egy várakozási sorba, végrehajtását tekintve egy külön szálon történik annak kiszolgálása. Az ennek során keletkezett eredmény visszaíródik a *Future* objektumba, ahonnan a kliens a megfelelő módon kiolvashatja azt, ezzel befejezve az adott folyamatot.[15]

A minta előnyei közé sorolható, hogy a konkurencia kezelés egyszerűsítése mellett külön választja az eljárás meghívását annak végrehajtásától, ezáltal biztosítva az aszinkron működést.

4.6.1.2 Proxy

A *Proxy* tervezési minta lényege, hogy egy konkrét objektumhoz való hozzáférést szabályozza, mindezt egy, a konkrét objektumot helyettesítő objektum segítségével teszi. Általában a helyettesítő objektum által publikált interfész megegyezik (részben vagy egészben) az elfedett objektum interfésszel.



4.9. ábra: A Proxy tervezési minta [22]

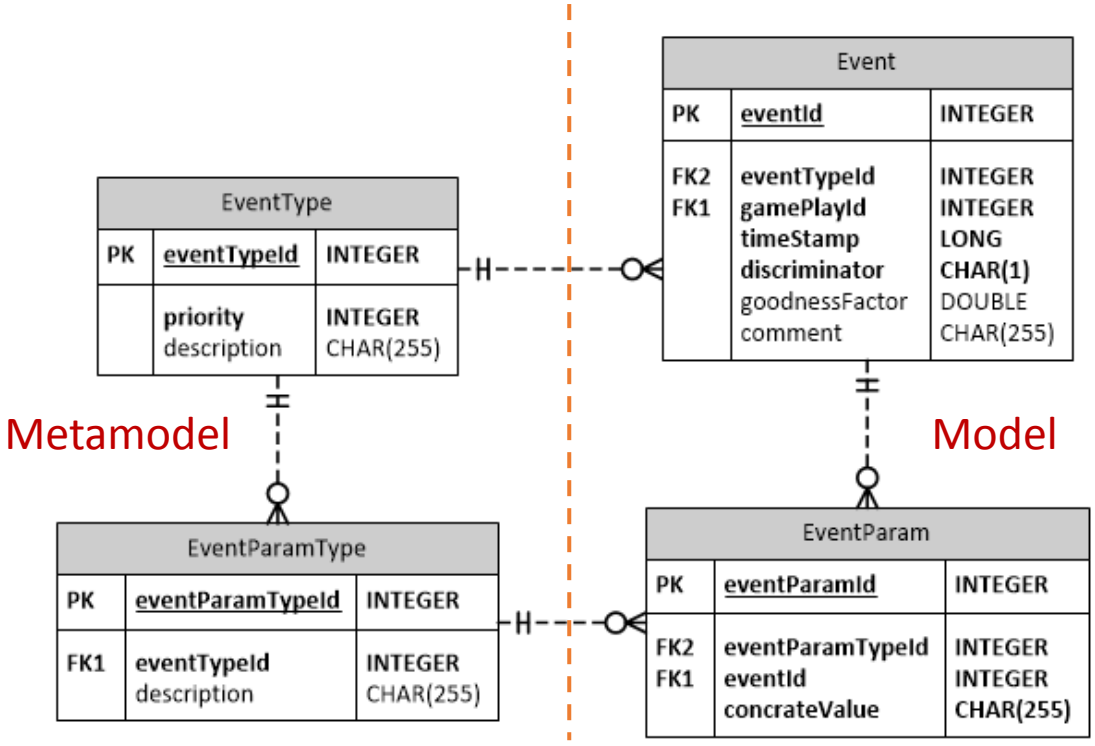
A proxy tervezési mintának több alváltozata terjedt el a gyakorlatban. Ezek közül az ismertebbek: *access proxy, remote proxy, virtual proxy, facade*. Ezek közül az általunk használt változat a *remote proxy*, mely lényegében a kliens elől elrejtik, hogy a kliens által látott objektumon hívott függvények egy távoli objektumhoz továbbítódnak és ott hajtódnak végre.

4.6.2 Dinamikus adatkezelés

Az előre nem ismert, de ugyanakkor támogatni kívánt adatok kezelhetőségére egy saját adatkezelési módszert kellett bevezetnünk, mely kellően általános ahhoz, hogy az imént vázolt problémát áthidalja.

A megoldást saját típusleíró osztály definiálása jelentette. Lényegében ezzel a megoldásunkkal egy újabb absztrakciós szintet vezettünk be, melynek köszönhetően az adatszerkezet kellően rugalmassá vált előre nem ismert típusok tárolásához és továbbításához.

A megvalósításunk adatbázis oldalon is támogatást kívánt, így az itt kialakított általános struktúra a különböző eseményekre vetítve a következőképpen lett megtervezve:



4.10. ábra: Adatábrázolás általánosságának kiterjesztése adatbázis oldalon

A típusleíró (*TypeDescriptor*) generálásakor tulajdonképpen összeállítunk egy listát, hogy az adott osztályra vonatkoztatva az milyen változókat tartalmaz, ezeknek a változóknak mi a típusa, valamint milyen intervallumon belül vehetnek fel értékeket. Ezen fent említett tulajdonságok tárolásáért felelős a *ParamType* osztály. A könnyű szerializálhatóság, és típusfüggetlenség és általánosság megőrzése érdekében az említett osztály csak String változókat tartalmaz. Ezek a paraméter neve, paraméter típusa, és a megengedett intervallum.

Az így legenerált típus leírókat egy listában tároljuk, mely lista egy típusleíró osztályhoz tartozik. Ezek a típusleírók sorosítódnak a beregisztrálási folyamat során, és kerülnek átküldésre a keretrendszer számára.

A sorosíthatóság megvalósíthatóságát a *Parcellable* interfész implementációjával érjük el, szemben a *Java*-ban megszokott *Serializable* helyett. A döntés meghozatalát a teljesítmény és időbeli különbségek indokolják, szemben a kényelmi megfontolásokkal. Míg a *Serializable* interfész implementálása nem okoz különösebb nehézséget és extra kódolást, addig a *Parcelable* implementálása plusz implementációt is maga után von, mindezt a karbantarthatóság, valamint az átláthatóság rovására. Ettől eltekintve a

Parcelable bevezetésének pont a *Serializable*-lel szemben mutatott sebességbeli javulás volt a fő mozgatórugója.

4.6.3 Típusok dinamikus beregisztrálása

Mivel a támogatott oktatójátékok számára biztosítani szeretnénk a lehetőséget saját típusok deklarálására, valamilyen módon a keretrendszer számára is szükséges, hogy ismerettel rendelkezzen ezen típusok létezéséről.

Ennek érdekében a keretrendszer inicializálási fázisában meghatározott részt dedikálunk ezen dinamikus típusok beregisztrálására.

A fejlesztés különböző szakaszaiban több lehetőség is kipróbálásra került a fenti probléma áthidalását illetően.

A kezdeti fázisban a kliens feladata volt, hogy egy számára biztosított interfészen keresztül végrehajtsa az általa definiált típusok beregisztrálását. Ez a működést tekintve jól elkülönített fázist jelentett a keretrendszer inicializálását illetően, ugyanakkor plusz terhet rótt a kliens oldalra, ezáltal növelve az esetleges hibalehetőségek előfordulását, valamint növelte a keretrendszer kliens felé mutatott komplexitását.

Második iterációban nem volt dedikált beregisztrálási folyamat. Ekkor ez dinamikusan, futási időben történt akkor, ha a keretrendszer olyan eseményt kapott, melynek típusa még számára ismeretlen volt. Ekkor magának az események a továbbításán túl magát a típus beregisztrálását is elvégezte. Ez a megoldás különböző konkurenciával kapcsolatos kérdéseket vetett fel, hiszen az üzenetek feldolgozása külön szálakon történik, így biztosítani kellett, hogy az esemény beregisztrálására vonatkozó üzenet mindenképpen előbb dolgozódjon fel, mint maga az esemény.

A végső megoldásban szintén külön beregisztrálási folyamat biztosította az ismeretlen típusok ismertetését a keretrendszerrel, ugyanakkor ez nem jelentett plusz függvényhívási kényszert a kliens oldal számára.

Ennek a megoldásnak az alapját a megfelelő típusok dinamikus felderítése jelentette. A feladatot ebben az esetben az jelentette, hogy futási időben megtaláljuk egy adott osztály leszármazottait. Ez esetünkben a *GameEvent* valamint a *Reward* leszármazottainak dinamikus felderítését jelentette.

Nem magától éretetődő, hogy ezt megfelelő módon miként tudjuk kivitelezni. A *Google* a fenti probléma feloldására hozta létre, és bocsájtotta a fejlesztők rendelkezésére a

Reflections[23] nevű osztálykönyvtárat. A könyvtár különböző futási idejű meta-adat analízis alapokon nyugvó lekérdezési lehetőségeket biztosít a felhasználó számára. Többek között lekérdezhetjük egy adott annotációval ellátott osztályok listáját vagy egy adott osztály összes leszármazottját.

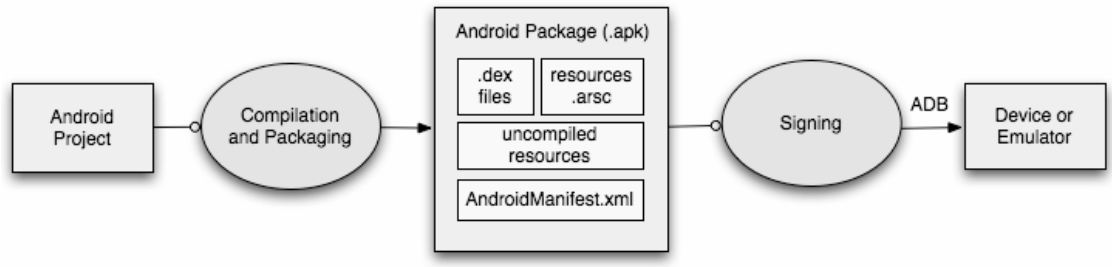
Sajnos magának a könyvtárnak a használata *Android* platformon bizonyos korlátokba ütközik.

A probléma feloldásához közelebbről meg kellett vizsgálni az *Android* által alkalmazott osztály betöltési technikát, illetve ennek folyamatát.

Az *Android* operációs rendszer által használt virtuális gép működését tekintve különböző területeken eltéréseket mutat a hagyományos standard *Java*-ban megszokott *JVM*-től. Jelen esetben a releváns működésbeli eltérés az osztályok *byte code*-ban való tárolására vonatkozik.

Java-ban a *compile* folyamat során a különböző java osztályok *byte code*-ra fordulnak, minden egyes osztály a nevének megfelelő file-ba kerül *.class* kiterjesztéssel. Futási időben ebből a *byte-code*-ból fordul a gépi kód (*JIT Just In Time*).

Az *Android* által alkalmazott *Dalvik Virtual Machine* nem futtat Java *byte-code*-ot. Az *Android* alkalmazások *dex* kiterjesztésű (*Dalvik Executable*) fájlokra fordulnak, majd ezek összetömörítése során keletkezik maga az *apk* fájl. Lényegében a *.dex* fájlok a *.class* fájlokhoz hasonlíthatók leginkább, azzal a különbséggel, hogy az előbbiek *Dalvik* virtuális gépen futnak. Egy *.dex* fájl több osztályt is magában foglalhat ellentétben a *class* fájlokkal. Ennek az az előnye többek között, hogy a több osztály által használt konstansok, *string*-ek így nem fordulnak elő duplikáltan a byte *code*-ban, ezzel a mobil eszközökön oly fontos helytakarékoskodási szempontoknak eleget téve.[24]



4.11. ábra: *Android* alkalmazás *build*-elési folyamata[24]

A fentiekben leírtak miatt a *Reflections* könyvtárban alkalmazott eljárás *Android*-on nem működőképes az osztályok eltérő reprezentációja miatt.

Első kísérlet a probléma leküzdésére igen bizonytalan megoldásnak ígérkezett. A bizonytalansága abból eredt, hogy reflexiót használva igen erősen támaszkodott arra, hogy osztályváltozó egy adott névvel szerepel egy osztály definíciójában.

Ebben a megoldásban azt használtam ki, hogy az *Android* forráskódban szereplő *PathClassLoader* osztály rendelkezik egy *mDexs* elnevezésű privát változóval, amin keresztül elérhető az alkalmazásunk *dex* fájljainak a listája. Ezektől a *dexFile*-októl lekérdezhető az általa tartalmazott osztály-definíciókhoz tartozó osztályok nevei. Amint rendelkezésünkre állnak az osztálynevek már a *ClassLoader*-t felhasználva betölthetjük az osztályokat.

Sajnos a fenti megoldás gyenge pontja miatt nem alkalmazható a gyakorlatban, mivel míg a *PathClassLoader* osztály korábbi *Android* verziókban a *ClassLoader* leszármazottjaként még tartalmazta a fent említett változót, addig az újabb *Android* verziókban már ez nem mondható el. (Itt már a *BaseDexClassLoader* leszármazottja).

A megoldást a fentebb vázolt folyamatból a *PathClassLoader* eliminálása jelentette. Az *Android* API rendelkezésünkre bocsájt egy *ApplicationInfo* elnevezésű osztályt, melytől lekérdezhetőek az alkalmazásunkkal kapcsolatos különböző információk. Ezen információk nagy része az *AndroidManifest* fájlban deklarált értékek megfeleltetését jelenti. Az említett információk között megtalálható többek között a *sourceDir* nevű változó melynek segítségével megkaphatjuk a *package*-ünk teljes elérési útvonalát. Erre a jelen helyzetben azért van szükség. mert az említet helyen található meg alkalmazásunk *dex* fájlja. Az említett elérési útvonalat paraméterül adva a *DexFile* konstruktorának máris lekérdezhetővé válik a file által tartalmazott bejegyzések, melyek megfeleltethetőek maguknak a tárolt osztályoknak.

Ezen a ponton már hozzáférhetünk a megfelelő osztályok neveihez. Azonban itt fontos észrevennünk, hogy a teljesítményi mutatókon valamint a végrehajtási időn sok javulás érhető el, ha nem töltünk be minden osztályt, csak a számunkra szükségeseket.

Fontos, hogy minél inkább takarékoskodjunk a virtuális gép *PermGen* tárhelyével. A *Permament Generation* rész a *JVM* memóriakezelésében egy külön memória részként definiálható, ahol a java virtuális gép a statikus változókat, valamint az osztálybetöltő által betöltött *class*-okat tárolja. A *ClassLoader* referenciát tart a már általa betöltött *class*-okra,

így a GC nem fogja tudni őket felszabadítani, tehát jelen esetben szükségszerű, hogy takarékoskodjunk az előbb említett hellyel, ezzel elkerülve váratlan *OutOfMemory kivételek*-ok előfordulását.

A fentiek alapján nem töltjük be az összes osztályt, hanem csak azokat, melyek egy, a játék *package*-ben helyezkednek el.

A játék *package*-nek meghatározását illetően a *getClass.getPackage.getName()* helyes értékkel fog szolgálni, köszönhetően annak, hogy *Java*-ban minden (nem *final* kulcsszóval ellátott) metódus virtuális, így megkaphatjuk a keretrendszert használó osztályok *package* nevét, mely segítségével már szűrni tudjuk a betölteni kívánt osztályok halmazát.

Az alkalmazott megoldás továbbfejlesztési lehetőségei közé sorolható annak megoldása, hogy magának osztálynak a betöltése nélkül lehessen meta adat információkhoz jutni az adott osztályt illetően *Android* platformon. Ezzel egyrészt teljesítményi javulást érhetünk el, hiszen magának egy osztálynak a betöltése viszonylag időigényes műveletnek tekinthető. Hasonló megoldást alkalmaz Java SE környezetben például a *Javassist* osztálykönyvtár [25] mely magából a *bytecode*-ból nyeri ki a szükséges meta-adat információkat, magának az osztálynak a betöltése nélkül.

Következő lépésként adódik, hogy futási időben el tudjuk dönteni, hogy egy megfelelő osztály egy másik adott osztálynak a leszármazottja-e, vagy sem. Ennek eldöntésére több lehetőség közül van módunk választani.

Ezek közül az egyik az *instanceOf* operátor használata, melynek első argumentuma egy példányváltozó, jobb operandusa pedig egy osztály neve. Az operátor igaz értéket szolgáltat, ha példány a megadott típus példánya. Itt fontos megjegyezni, hogy *instanceOf* operátor fordítási időben értékelődik ki, így szükséges, hogy már ekkor ismerjük az adott típust, mely jelen esetben nem áll rendelkezésünkre[3].

A futási idejű dinamikus kiértékelést részesítve előnyben a számunkra leginkább megfelelő metódus az *isAssignableFrom()*. Ez a függvény paraméterül egy *Class object*-et vár, ellentétben az előzőekben említett metódussal. A függvény akkor ad vissza igaz

[3] A végrehajtási időket figyelembe véve az *instanceOf* operátor használata a gyorsabb. Ez magyarázható azzal, hogy az *instanceOf* operátorhoz külön *bytecode* tartozik.

értéket, ha az osztály, amin a metódust meghívtuk, megegyezik, vagy őse a paraméterül átadott osztálynak.

```
List<Class<? extends Parent>> subClasses = ListFactory.newArrayList();

// ...

Class<?> clazz = (Class<?>) getClassLoader().loadClass(className);
if (parent.isAssignableFrom(clazz)) {
     @SuppressWarnings("unchecked")
     Class<? extends Parent> subclazz = (Class<? extends Parent>) clazz;
          subClasses.add(subclazz);
}[4]
```

A fenti kódrészletből látszik, hogy a kódolási folyamat megkönnyítésére létrehoztunk generikus, statikus *factory* metódusokat a különböző generikus kollekciók létrehozására. (pl.: *ListFactory*).

A *Java*-ban sajnos egy generikus típussal rendelkező osztály példányosításánál szükséges, hogy a konstruktor meghívásakor a generikus típusokat is deklaráljuk. Ellenben ha egy függvényt hívunk meg mely generikus típussal rendelkezik valamint egyértelműen kitalálható a paraméterek típusa, akkor ezt a *compiler* megteszi helyettünk.

Ezt a két tényt ötvözve készítettük el a különböző kollekciókhoz tartozó *Factory* osztályokat. Például egy lista létrehozását kiszolgáló metódus:

```
public static <E> ArrayList<E> newArrayList(){
     return new ArrayList<E>();
}
```

Ennek köszönhetően elkerülhető redundáns módon a generikus paraméterek megismétlése a kollekciók létrehozásakor.

---

[4] Látható, hogy a *compiler warning*-ot adni a *cast*olásra. Fontos, hogy a fejlesztés során figyeltünk arra, hogy *@SuppressWarnings("unchecked")* annotációt csak bizonyítható esetekben raktuk ki, ezzel elkerülve, hogy hamis biztonságérzetet keltsünk magunkbanban, valamint a keretrendszert később továbbfejlesztőkben.

Törekedtünk továbbá arra is, hogy ha a fenti annotációt kellett használnunk, akkor minél inkább leredukáljuk az annotált *scope*-t. Ennek érdekében sokszor vezettünk be lokális változókat ahelyett, hogy inkább egy egész függvény annotáljunk *unchecked cast* annotációval.

4.6.4 Dinamikus kommunikáció megvalósítása

A keretrendszer könnyű, és rugalmas használatát valamint dinamikus bővíthetőségének lehetőségét szem előtt tartva biztosítani szerettük volna a keretrendszert felhasználó és szolgáltatásait igénybe vevő kliensnek a dinamikus és adaptív kommunikáció lehetőségét a felügyelő alkalmazás felé.

Ennek biztosítására több problémát is át kellett hidalni a keretrendszer fejlesztése során. Az egyik ezek közül, hogy előre nem ismert, hogy a kliens milyen módon szeretne kommunikálni a felügyelő alkalmazással, milyen metódusokat kívánna meghívni a kommunikáció lebonyolítása céljából. Fontos, hogy megkötni sem szeretnénk a kliensoldalt azzal a megszorítással, hogy csak előre definiált metódusok közül biztosítjuk számára a választás lehetőségét. Ezt alátámasztva biztosítani szerettük volna a keretrendszer funkcionalitását igénybevevő számára, hogy teljes mértékben ő határozhassa meg, hogy milyen metódussal, milyen paraméterekkel, és milyen visszatérési értékkel kíván kommunikációt végrehajtani a keretrendszer irányába.

A másik felmerülő probléma, hogy magát a kommunikációt aszinkron módon kell tudni lebonyolítani, különválasztva a metódus meghívását, az eredmény megkapásától. Ezt a kommunikációval szemben támasztott elvárásunkat az indokolja, hogy a kommunikáció üzenet alapon történik, mely üzenet csatornájának a hálózat tekinthető, így ebből fakadóan szükséges az aszinkronitás támogatása és biztosítása. Mindezt úgy szerettük volna véghezvinni, hogy a kliens oldal számára ez a lehető leginkább transzparens módon történjen meg, ugyanakkor az alkalmazott megoldás ne menjen a keretrendszer rugalmasságának, kényelmes használatának és könnyű bővíthetőségének a rovására.

A megoldást a fenti problémára a korábban említett két tervezési minta felhasználása jelentette, valamint ezek elegyének kiegészítése a dinamikus proxy nyújtotta lehetőségekkel.

A kliensnek valamilyen módon definiálni szükséges, hogy milyen metódusokat szeretne meghívni, illetve, hogy a felügyelő alkalmazástól erre milyen választ vár. Tulajdonképpen elég egy függvény deklarációt megadnia, az implementáció ilyen szempontból irreleváns. Erre a legmegfelelőbb módszer egy interfész deklarálása, melyben kliens felveszi a kommunikációra szánt metódusait a megfelelő paraméterezéssel.

A keretrendszer által szolgáltatott proxynak a fentebb leírt interfészt kell becsomagolni, és hasonló interfész visszaszolgáltatni a kliens felé. Itt a nehézséget az

jelenti, hogy előre nem ismerjük, hogy milyen interfész kell megvalósítania a keretrendszer *proxy*-jának.

Erre jelent megoldást a dinamikus proxy. A dinamikus proxy a Java 1.3-as változatába került bele, mely funkcionalitását tekintve interfészek dinamikus, futásidejű „példányosítására" volt alkalmazható. Dinamikus proxy létrehozásához és használatához két fontos elemet kell felhasználnunk. Az egyik a *java.lang.reflect.Proxy* osztály, melynek statikus *factory* metódusával (*newProxyInstance*()) gyárthatjuk le az interfészt megvalósító objektumot futási időben. A fenti metódusnak paraméterként át kell adni egy *ClassLoader*-t, amely képes betölteni a kívánt interfészt, egy Class tömböt, mely tartalmazza a betölteni kívánt interfészeket, valamint egy *InvocationHandler* implementációt. A dinamikus proxy létrehozásának másik fontos alkotóeleme a *java.lang.reflect.InvocationHandler* interfész.

Az *InvocationHandler* interfész tartalmaz egy *invoke*() nevezetű függvényt. Tulajdonképpen a proxy objektum és a valódi objektum közötti köztes réteg képviseletéért felel a fent említett metódus. A függvény paraméteréül megkapjuk a proxy objektumot, a metódust, melyet meghívtak rajta, valamint a paraméterek tömbjét. A keretrendszerben tulajdonképpen ezen a ponton válik külön a kérés és a végrehajtás egymástól. A fenti függvényen keresztül megtörténik magának az üzenetnek a legenerálása, mely tartalmazza a függvény hívás kontextusát, és mely a megfelelő módon továbbítódik a keretrendszer felé.

Ezen a ponton azonban a dinamikus proxy implementációját kiegészítettük az *Activation Object* minta által bevezetett *Future* objektum felhasználásával. A fenti *invoke* metódus egy *Future* objektummal fog visszatérni (ha az eredeti metódus visszatérési értéke nem *void* lett volna, tehát eleve a hívó nem is várt volna választ a felügyelő oldaltól). A *Future* objektum segítségével az *Activation Object* minta alapján a hívó fél hozzájuthat az elvárt visszatérési értékekhez.

```
public class Future<T> {
     private T result;
     private OnResultListener<T> onResultListener;
     private Object syncObject = new Object();

     public T get() {...}
     public T get(long millis) {...}
     public boolean isDone() {...}
     public void setOnResultListener(OnResultListener<T> onResultListener)
     {...}
     private void setResult(T result) {...}
}
```

Az általunk definiált *Future* osztályt generikus típussal láttuk el, így könnyen tud tárolni bármilyen elvárt eredménytípust. A kliens az *isDone()* metódussal ellenőrizheti, hogy az eredmény megérkezett-e már vagy sem. A *get()* metódus segítségével blokkolva várakozhat a visszatérési érték megkapására. Ennek megvalósítására implementációs oldalon szinkronizációs objektumot használtuk fel, valamint a *wait() notify()* függvények együttesét. Lehetőség van ezen kívül csupán egy megadott timeout erejéig várakozni. A kliensnek továbbá lehetősége van *listener (OnResultListener<T>)* beregisztrálására is, mellyel szintén aszinkron módon juthat a kívánt eredményhez.

```
private void setResult(T result) {
     this.result = result;
     synchronized (syncObject) {
          syncObject.notify();
     }
     if (onResultListener != null) {
          onResultListener.onResult(result);
     }
}
```

A *Future* osztályt belső osztályként (*inner class*) definiáltuk. Ezt a döntést az tette indokolttá, hogy a *Future* objektumon az eredmény beállítható legyen erre alkalmas publikus API kiajánlása nélkül. A külső osztály (*outer class*) hozzáfér a belső osztályának privát láthatóságú adattagjaihoz, így esetünkben az eredmény beállítását végző *setter* metódushoz. Eredmény beérkezésekor a *Future* objektum felelőssége, hogy értesítse a nála esetlegesen beregisztrált *listener* objektumot ennek megtörténtéről, és átadja neki az említett értéket.

Problémát jelenthet, hogy válasz érkezésekor miként azonosítjuk a *Future* objektumot, amely felelős annak továbbításáért, tárolásáért.

Ennek kivitelezésére minden egyes létrejövő *Future* objektumhoz generáltunk egy egyedi azonosítót, mely az adott objektumot azonosítja a válasz érkezésekor. Ez az azonosító a generált üzenet egy extra mezője segítségével kerül továbbításra a keretrendszer részére. A *framework* számára ezek az extra mezők értéke irreleváns, viszont ugyanakkor fontos, hogy ezeket változtatás nélkül visszaküldje a válasz üzenetben, hiszen az azonosítás szempontjából az említett értéknek elengedhetetlen szerepe van, hiszen a válasz lekezeléséért felelős *Future* objektum ez alapján lesz azonosítható.

Felmerül a kérdés, hogy hogyan tudunk egyedi azonosítót generálni Java környezetben.

Használhatnánk a minden *Object* leszármazott számára *rendelketésre* álló *hashCode()* metódusát, mely az objektumot azonosító *hash* értéket lenne hivatott visszaadni. Ennek kiszámításában illetve generálásában jelentős szerepet játszik az objektum létrehozásakor kapott eredeti memóriacím. A probléma viszont, hogy a fent említet metódus felüldefiniálható, így az ezen keresztüli egyediség nem feltétlenül biztosított.

A fentinél kifinomultabb megoldást jelenthet *System.identifyHashCode()* metódus használata, mely az objektum „eredeti" *hash* kódját adja vissza. Ezzel a módszerrel szemben az egyediséget illetően a (leginkább modern) *JVM*-ek működési elve jelenthetne kifogást. Az, hogy tudjuk egy objektumról annak létrehozásakor kapott memóriacímet, az nem jelenti feltétlenül azt, hogy ez a későbbiekben is érvényes marad, hiszem a *Garbage Collector* működése közben áthelyezheti az objektumokat egyik memóriaterületről egy másikra. Az élő objektumokhoz tartozóan pedig a *hash* értékek ilyen szempontú folyamatos frissítése igen erőforrás igényes művelet lenne, másrészt egy objektumhoz tartozó *hashcode* állandóságára igen sok helyen támaszkodnak.

Harmadik lehetőség a Java által biztosított UUID osztály használata, melynek segítségével globálisan egyedi azonosítót generálhatunk magunknak[5].

Jelen helyzetben nem szükséges globálisan egyedi azonosító, elég, ha csak magán a *process*-en belül garantált az egyediség. Ennek az igénynek a kielégítésére megfelelő az *AtomicInteger* vagy az *AtomicLong getAndIncrement()* metódusa.

Mivel fontos, hogy az azonosítók generálásáért felelős objektum csak egyszer kerüljön inicializálásra, ezért magát, a dinamikus proxy megvalósításáért felelős osztályt a *Singleton* tervezési minta szerint valósítottuk meg, biztosítva ezzel a fent leírtakat.

Miután rendelkezésünkre áll egy azonosító, szükséges, hogy valamilyen módon tároljuk azt egységben a *Future* objektumokkal.

---

[5] Annak az esélye, hogy a UUID osztály segítségével két megegyező azonosítót generálunk igen elenyésző. Annak esélye, hogy valakit eltaláljon egy meteorit nagyjából ahhoz mérhető, hogy néhány tíz trillió legenerált azonosító között legyen két egyforma. Ha másodpercenként 1 milliárd azonosítót generálnak az elkövetkezendő 100 évben, akkor is annak esélye, hogy ezek között legyen duplikátum körülbelül 50%.[15]

A tárolás technikájának kiválasztásakor törekedtünk, hogy elkerüljük az esetleges *memory leak* fennállásának esélyét. Ennek érdekében a Java által biztosított *WeakReference* generikus osztályt használtam a *Future* objektumok becsomagolására. Ennek az osztálynak a használatával elérhető, hogy magára a *Future* objektumra a keretrendszer által fenntartott hivatkozás egy gyenge hivatkozás lesz, azaz ha már csak ilyen típusú hivatkozások érvényes az objektumra, akkor a *Garbage Collector* felszabadíthatja az objektumot. Ennek előnye abban rejlik, hogyha a felhasználó úgy dönt, hogy a válasz figyelésére alkalmas *Future* objektumra neki már nincs szüksége, és ennek következtében elengedi az erős referenciát, akkor ennek ellenére, hogy a keretrendszer még gyengén hivatkozik rá, a *GC*-t nem akadályozzuk a memória felszabadításban ezzel.[6]

A következő kérdés, hogy milyen kollekciót alkalmazzunk a kulcs érték párok tárolására. Egyik lehetőségként adódik valamilyen Map implementáció használata. Ugyanakkor mivel jelen helyzetben a kulcsok értékét Integer objektumok szolgáltatják, így *Androidon* lehetőségünk van a *SparseArray* kollekció alkalmazására. Utóbbit a dokumentáció alapján teljesítményi szempontokból inkább érdemes használni, mint a Map-et, akkor, ha a kulcsaink *Integer*-ek, mivel különböző optimalizálásoknak köszönhetően a kollekciókon végrehajtható alapműveletek hatékonyabban és gyorsabban hajtódnak végre.

---

[6] A Java nyelv rendelkezésünkre bocsájt még további referencia típusokat. A *SoftReference* típusról annyit érdemes tudni, hogy a memóriában tartás mikéntjét illetően valahol a *hard* és a *weak* típusok között helyezkedik el, ugyanis az említett referencia típusra az a jellemző, hogy ha egy objektum már csak az említet típus által hivatkozott, akkor a lehetőségekhez mérten nem történik meg annak felszabadítása, kivéve ha a JVM memória szűkébe kerül. Ekkor a már csak *soft reference* típussal hivatkozott objektumok kerülnek felszabadításra. Ennek a referenciatípusnak leginkább memóriabeli cache implementálásánál van nagy haszna, hiszen így a *cache*-elt adatokat a lehető leginkább tovább a memóriában tarthatjuk, viszont mindezt a memória túlcsordulásának veszélyeztetése nélkül.

A teljessége kedvéért érdemes lehet még megemlíteni a *PhantomReference* típust, mely a leggyengébbnek tekinthető az eddig említettek közül. Ritkán szokták alkalmazni, leginkább különböző objektum felszabadítási műveletek implementálásánál jöhet jól a használata.

5 Játékok

A keretrendszer működésének igazolásához szükséges oktatójáték alkalmazások elkészítése is. Ezek kiválasztásakor fontosnak találtuk, hogy jól bevált módszereken alapuló játékok kerüljenek elkészítésre, így könnyen összevethetőek a keretrendszerrel megtámogatott új digitális változat, és a korábbi, digitális technikát mellőző változat eredményei. A projektben rejlő lehetőségek nagyságát jól mutatja, hogy már két nagynevű intézmény számára is készülnek ilyen játékok: a Meixner Alapítvány, valamint a Liszt Ferenc Zeneművészeti Egyetem számára. Most ezen játékok bemutatása következik, de ezeken kívül még önálló laboratórium keretein belül más hallgatók is foglalkoznak a keretrendszerhez illesztett játékok fejlesztésével.

5.1 A Meixner Alapítvány számára készülő játék

A Meixner Alapítvány Dr. Meixner Ildikó gyógypedagógus, pszichológus, a pedagógiai tudományok kandidátusa iskolateremtő tudományos munkásságának eredménye. 50 éves pályafutásának kezdetén a gyermeki beszédfejlődéssel foglalkozott, később érdeklődése az olvasás-, írástanulás és tanítás felé fordult, mert munkája során azt tapasztalta, hogy a beszédhibás és a beszédzavarban szenvedő gyermekek igen nehezen sajátítják ezt el.

Munkássága során kidolgozta a dyslexia és a beszédfogyatékosság diagnosztizálásának, megelőzésének és kezelésének módszertanát, mely a magyar gyógy- és fejlesztőpedagógia büszkesége. Ezek alkalmazása hatékonyan segíti az ilyen problémáktól szenvedő gyermekek felzárkóztatását a bölcsődétől a középiskoláig.

Ennek jelentőségét mi sem mutatja jobban, mint hogy a szakirodalom szerint a populáció 7-10%-a kehet érintett a dyslexia szempontjából. Ebbe a 7-10%-ba gyakran jó, vagy kiváló intellektusú gyermekek is tartoznak, akik ebből a problémából adódóan nem a képességeiknek megfelelő karriert futják be, az esetek többségében elkerülhetetlen lemaradásuk miatt. Sőt az is előfordulhat, hogy az iskolai kudarcok viselkedési problémák kialakulásához vezetnek, a későbbiekben pedig szorongás vagy pszichoszomatikus betegségek is kialakulhatnak.

A dyslexia prevenció módszere még írni-olvasni nem tudó gyermekeknél alkalmazandó, melynek segítségével a jobb és a gyengébb képességű gyermekek fejlődése egyaránt biztosítható. Segítségével a gyermekek olvasási és szövegértési készségük sokkal biztosabb lesz, ami annak köszönhető, hogy épít az életkori és nyelvi sajátosságok mellett a nyelvi működésre is. A módszerre egy tankönyvcsalád is épült, melynek feladattípusai lehetővé teszik az eltérő képességű tanulók számára a differenciált feladatadást, ezt bizonyítandó évente kb. 20000 kisgyermek tanul eredményesen ebből az olvasókönyvből olvasni, akik között egyaránt találhatók dyslexia veszélyeztetettek és harmonikusan fejlődőek egyaránt.

A dyslexia reedukáció módszerét azoknál a gyermekeknél alkalmazzák, akiknél a dyslexia már kialakult. A felzárkóztatási folyamat évekig is eltarthat, ezalatt a különböző szakemberek (gyógypedagógusok, fejlesztőpedagógusok, logopédusok) többnyire csoportos terápiát alkalmaznak, melynek sikeressége függ a szakember felkészültsége mellett a gyermek adottságaitól, a foglalkozások gyakoriságától és a szülői támogatás mértékétől is.

A terápiának az iskolai tanulmányok végéig folytatódnia kell, hiszen az olvasás megfelelő technikai szintre emelésén kívül a szövegértés fejlesztése, a tantárgyi szókincs elsajátítása, és az önálló ismeretszerzés fontossága is kiemelkedő.

Ehhez sok jó minőségű és hatékony taneszközre lenne szükség, ám ezek hiánya tovább nehezíti a szakemberek feladatát a rendelkezésükre álló szűkös óraszám mellett. Általában ez visszariasztja a pedagógusokat, hiszen a rendkívül nagy mennyiségű és változatos gyakorlóanyagot így nekik kellene pótolniuk, amire sok esetben nincs lehetőségük.

Mivel ma Magyarországon a Meixner-féle módszertant alkalmazza a dyslexia reedukációt végző szakemberek többsége, ezért egy az alapítvány által kiadott mobil alkalmazás orvosolhatná a fent említett problémát, hiszen rövid idő alatt nagyobb mennyiségű feladat megoldását tenné lehetővé, valamint könnyen hozzáférhető lenne a pedagógusok számára.

Az ő számukra különösen nagy jelentőséggel bírhat az általunk fejlesztett rendszer, hiszen a jól bevált módszereiket a XXI. századi technika segítségével újabb szintre emelhetik. Módszerük egyik kulcseleme az úgynevezett borítékos módszer, melynek lényege, hogy egy borítékba tesznek szótagokat, szavakat, mondatokat, képeket, és ezeket

kell a gyerekeknek az asztalon sorba rakniuk, illetve összepárosítaniuk. Ennek a digitális megfelelője készült el a projekt keretein belül, kihasználva a tabletek által biztosított intuitív kezelőfelület sajátosságait, így a tableten is – ahol ez indokolt - szabadon mozgathatóak a felhasználói felület elemei, mintha azok valóban papírdarabok lennének.

5.1. ábra Meixner Alapítványnak készített játék: mondat – kép párosítás

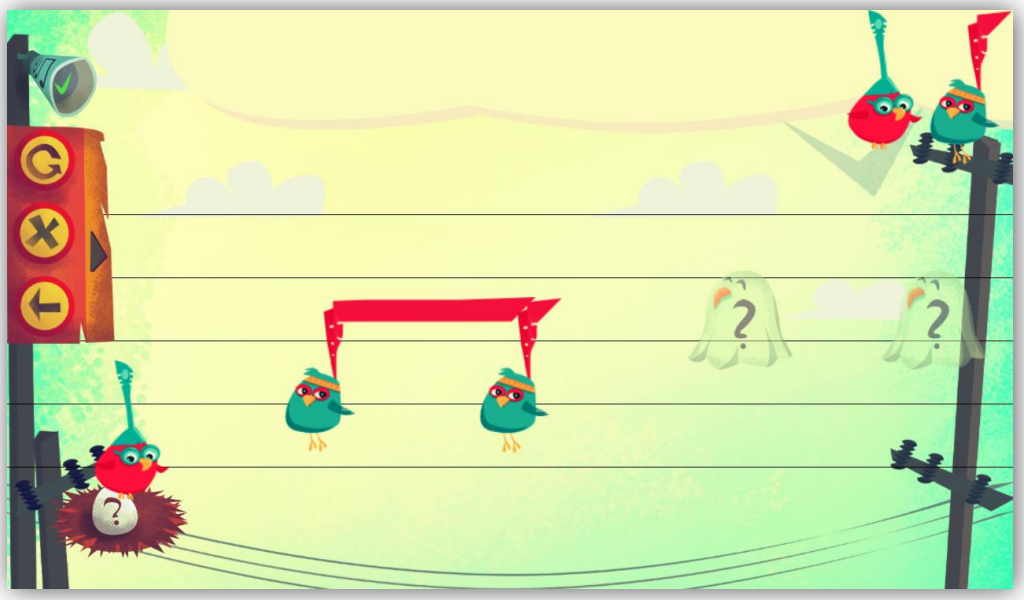
5.2. ábra Meixner Alapítványnak készített játék: szinonimák összekapcsolása

Az egyes pályák nehézségének meghatározása pedagógus szakemberek segítségével történt, így a programot használó gyermek mindig a fejlődését lehető legjobban elősegítő feladatot oldhatja meg, ezáltal elkerülve a túl nehéz feladat okozta

csalódást, vagy a túl könnyű feladattal járó unalmat, és otthon is bármikor gyakorolhat az alkalmazás segítségével. Az egyes pályák végén kapott jutalmat egyelőre csillagok jelentik, és ezek száma jelzi a jutalom nagyságát, de a későbbiekben lehetőség lesz ezek méretének és színének módosítására, vagy akár más jellegű jutalom adására is, alkalmazkodva a gyermek egyéni ízléséhez, hiszen nem biztos, hogy mindenkinél ugyanaz a jutalomérzet tartozik egy adott jutalomhoz. Így lehetővé válik a motiváció fenntartása.

5.2 A Liszt Ferenc Zeneművészeti Egyetem számára készülő játék

A Liszt Ferenc Zeneművészeti Egyetemmel együttműködve készített alkalmazás szintén a 6-10 éves korosztály számára készült, célja a kottakép bevezetése, a kottaírás, kottaolvasás készségének elsajátítása egy a gyerekek számára érdekesebb, színesebb, játékosabb módon. Mivel a Meixner Alapítvány számára készített játékkal ellentétben itt nem volt adott a játék dizájnja, és a játék eredményességének szempontjából is fontos a figyelmet megragadó, igényes megjelenés, ezért a felhasználói felület tervezését valamint a grafikai munkákat a Moholy-Nagy Művészeti Egyetemen végezték. A játékban a hangjegyeket madarak jelképezik, ritmusonként eltérő külsővel, a kottavonalak pedig villanypóznák között lógó drótok képében jelennek meg. Ezekre kell a feladványnak megfelelően a pálya elején megmutatott kottakép, vagy hallás után elhelyezni a megfelelő ritmus értéket jelképező madarat.



5.3. ábra Zeneakadémiának készített játék: feladvány

A pálya végén kapott jutalmat kukacok jelképezik, melyek külseje a játékos teljesítményétől függően változik.

5.4. ábra Zeneakadémiának készített játék: pálya végi jutalom

A különböző szinteken megjelenített feladatok nehézségének meghatározása ebben az esetben is pedagógusokkal egyeztetve történt, így ezek helyességéhez sem férhet kétség.

5.3 Játékok illesztése

A játékok illesztése az elvárásoknak megfelelően könnyű feladatnak bizonyult. Mivel a keretrendszerrel történő kommunikációhoz szükséges logika elérhető egy osztálykönyvtár formájában, a játékok fejlesztésekor csupán ebben a könyvtárban található osztályokból kellett leszármaztatni. A játékot futtató *Activity*-k a *GameBase*-ből származnak, a felhasználói interakciókhoz kapcsolódó események a *GameEvent*-et terjesztik ki, mint például egy hangjegy behúzását jelző esemény, a pálya végi jutalmat reprezentáló osztály őseként pedig a *Reward* osztály használatos. Az esemény és jutalom típusok esetén annotációkat használva adhatóak meg az olyan metaadatok, mint például a Meixner játékok esetében a pálya végi csillagok minimális és maximális száma. Ez rendkívül kényelmes, könnyedén használható megoldás, gyakorlatilag semmilyen plusz terhet nem jelent ahhoz képest, mintha keretrendszer nélkül fejlesztenénk a játékot. A keretrendszerrel történő kommunikáció felépítése, valamint az egyes üzenetek küldése és

fogadása teljesen transzparens. Az üzenetek küldéséhez egy egyszerű függvényhívásra van szükség, esemény fogadása pedig *callback* függvényen keresztül történik. Elmondható tehát, hogy megvalósult az a tervezési szempont, miszerint a játékfejlesztőkre semmilyen plusz terhet nem róhat a keretrendszer felhasználása.

6 Összefoglalás, további feladatok

A leírtak alapján tehát elmondható, hogy sikerült megvalósítani a rendszer tervezésekor kitűzött célokat, mint a tanulás hatékonyságának maximalizálása és a felhasználói élmény fokozása a szenzoroktól származó visszacsatolás alapján, alkalmazkodva a felhasználói igényekhez. Mindeközben ezt a komplexitását sikerült teljesen elfedni a játékok fejlesztői elől, hiszen a rendszer szolgáltatásainak használata semmilyen plusz terhet nem ró rájuk, ezzel segítve az elterjedését. Továbbá elkészült két oktatójáték is, olyan neves intézmények pedagógusainak közreműködésével, mint a Meixner Alapítvány, vagy a Liszt Ferenc Zeneművészeti Egyetem. Utóbbi lehetőséget nyújt a rendszer első éles tesztelésére is, hiszen november-december folyamán 80 gyermek fogja a kipróbálni a XXI. század legújabb kottaolvasást tanító módszerét. Ezek alapján elmondható, hogy a dolgozatban bemutatott szoftver alkalmas lehet arra, hogy forradalmasítsa az oktató és készségfejlesztő játékokat.

A rendszerben rejlő lehetőségeket jól mutatja, hogy számos ötletünk van a már így sem csekély funkcionalitás továbbfejlesztésére. A továbbiakban még több szenzor illesztését tervezzük, továbbá neurális hálók alkalmazását a nehézségi szint és a jutalom meghatározásában. EEG és a szívritmus mérő jelének hatékony, wavelet alapú vizsgálatával további összefüggések feltárása fog lehetőség nyílni. Szerver oldali adatok adatbányász algoritmusokkal történő elemzése alapján pedig lehetőség nyílik a hatékonyság számszerűsítése. További alkalmazások illesztésére is sor fog kerülni, mint például a Meixner Alapítvány által alkalmazott szintfelmérő digitális változata.

Irodalomjegyzék

[1] M.W.G. Dye, C.S. Green, and D. Bavelier. The development of attention skills in action video game players.

[2] Kiel Mark Gilleade, Alan J. Dix, and Jennifer Allanson. Affective videogames and modes of affective gaming: Assist me, challenge me, emote me (ace). In DIGRA Conf., 2005.

[3] Lennart Erik Nacke, Michael Kalyn, Calvin Lough, and Regan Lee Mandryk. Biofeedback game design: using direct and indirect physiological control to enhance game interaction. In Proceedings of the 2011 annual conference on Human factors in computing systems, CHI '11, pages 103–112, New York, NY, USA, 2011. ACM.

[4] R.S. Sutton, A.G. Barto, "Reinforcement Learning", MIT Press, Cambridge, MA., 1998.

[5] W. Schultz, "Reward signaling by dopamine neurons," Neuroscientist, 7(4), pp. 293-302, Aug 2001.

[6] W. Schultz, P. Dayan, P. R. Montague, "A neural substrate of prediction and reward," Science, 275, pp. 1593–1599. 1997.

[7] M. Aggarwal, B. I. Hyland, J. R Wickens, "Neural control of dopamine neurotransmission: implications for reinforcement learning," Euro, J. Neurosci., 35, pp. 1115–1123, 2012.

[8] S. B. Flagel, J.J. Clark, T. E. Robinson, L. Mayo, A. Czuj, I. Willuhn, C. A. Akers, S. M. Clinton, P. E. Phillips, H. Akil, "A selective role for dopamine in stimulus-reward learning," Nature, 469, pp 53-57, 2011.

[9] K.C. Berridge, T.E. Robinson, "What is the role of dopamine in reward: hedonic impact, reward learning, or incentive salience?" Brain Res. Rev., 28, pp. 309–369, 1998.

[10] H.M. Bayer, P.W. Glimcher, "Midbrain dopamine neurons encode a quantitative reward prediction error signal." Neuron, 47, pp. 129–141. 2005.

[11] M.J. Koepp, RN Gunn, A.D. Lawrence, V.J. Cunningham, A. Dagher, T. Jones, D.J. Brooks, C.J. Bench, P.M. Grasby, "Evidence for striatal dopamine release during a video game," Nature., 393(6682):266-8, 21 May 1998.

[12] S. Kuhn, A.Romanowski, C. Schilling, R. Lorenz, C. Morsen, N. Seiferth, T. Banaschewski, A. Barbot, G. J. Barker, C. Buchel, P. J. Conrod, J. W. Dalley, H. Flor, H. Garavan, B. Ittermann, K. Mann, J-L. Martinot, T. Paus, M. Rietschel, M. N. Smolka, A. Strohle, B. Walaszek, G. Schumann, A. Heinz, J. Gallinat, "The neural basis of video gaming," Transl Psychiatry, 1, e53, 2011.

[13] Yan Wu, Xiaolin Zhou: The P300 and reward valence, magnitude, and expectancy in outcome evaluation, Brain Research, Volume 1286, 25 August 2009, Pages 114–122

[14] Károly Hercegfi: Event-Related Assessment of Hypermedia-Based E-Learning Materials With an HRV-Based Method That Considers Individual Differences in Users, International Journal of Occupational Safety and Ergonomics (JOSE) 2011, Vol. 17, No. 2, 119–127

[15] Budapesti Műszaki és Gazdaságtudományi Egyetem, Automatizálási és Alkalmazott Informatikai Tanszék, Szoftverarchitektúrák (VIAUM105) tárgyának segédlete

[16] Wikipedia, Universally Unique Identifier http://en.wikipedia.org/wiki/Universally_Unique_Identifier

[17] Java SE 1.6 SDK Documentation, http://docs.oracle.com/javase/6/docs/api/ (2013. október)

[18] Android SDK Documentation, http://developer.android.com/reference/packages.html (2013. október)

[19] Charlie Collins, Michael D. Galpin, Matthias Käppler: Android in Practice, ISBN 9781935182924

[20] Pipelines, http://bigballofmud.wordpress.com/2009/04/18/pipelines/

[21] Active Object Pattern http://www.codeproject.com/Articles/56617/Applied-Long-Running-Active-Object-Pattern

[22] Wikipedia, Proxy design pattern, http://en.wikipedia.org/wiki/Proxy_pattern

[23] Reflections library, https://code.google.com/p/reflections/

[24] Android Developer, Android Building and Running http://developer.android.com/tools/building/index.html

[25] JavaAssist, https://www.javassist.org/