DIPLOMAMUNKA

Kárándi Róbert

Debrecen
2010

Debreceni Egyetem
Informatikai Kar

Web alapú alkalmazásfejlesztés

Órarend generáló rendszer megvalósítása genetikus algoritmus segítségével, JavaFx technológiával, RESTful webszolgáltatás alkalmazásával

Témavezető:
Dr. Rutkovszky Edéné
Ügyvivő szakértő

Készítette:
Kárándi Róbert
Programtervező matematikus szak

Debrecen
2010

Tartalomjegyzék

Bevezetés

„Three billion years of evolution can't be wrong.
It's the most powerful algorithm there is."
Dr. David E. Goldberg, University of Illinois, 1989

Napjainkban az informatika egyre inkább szerves részévé válik az életünknek, jelen van mindenhol. Az internet segítségével az emberek kapcsolatot teremthetnek egymással, tájékozódhatnak a világban történt eseményekről, megtalálhatnak rajta bármilyen információt, amelyre szükségük van. Az egyre intelligensebb számítógépes rendszerek alkalmazásával a bonyolult, számításigényes problémák is könnyedén megoldhatók.

Fejlődő világunkban a különféle ütemezési feladatok egyre nagyobb gondot jelentenek. Gondolhatunk itt az egészen egyszerű napi teendők megtervezésétől kezdve, az egészen nagy beruházásokig. Az ütemezési feladatok egy speciális változata az órarendtervezés, amikor szem előtt kell tartanunk a tanárok és diákok elvárásait, valamint az iskola lehetőségeit.

Diplomamunkám témája ezen az oktatásban jelentkező probléma megoldását hivatott szolgálni. Témaválasztásom azért esett erre a területre, mert úgy gondoltam, hogy ez egy kellőképpen összetett feladat, melynek megoldásához szükséges ismeretek csakis szolgálhatják a szakmai fejlődésemet.

Egy órarend szerkezeti, strukturális felépítésének megértésében és az összeállításához szükséges feltételek megismerésében nagy segítségemre volt a debreceni Ady Endre Gimnázium[5] aktuális órarendje, amelyet Rózsavölgyi Gábor igazgató úr bocsátott rendelkezésemre.

Dolgozatomban három fő fejezetet szeretnék kiemelni. Az első fejezetben a genetikus algoritmust ismertetem, amely a feladatban szereplő elvárásokat szem előtt tartva megalkot egy feltételeknek megfelelő órarendet.
A második fejezetben a webszolgáltatások probléma körét, a szerver-kliens közötti kommunikációt és a konkrét megvalósítást tárgyalom.

A harmadik fejezet viszonylagosan egy felhasználói kézikönyvnek tekinthető, amelyen belül leírom az egyes felületek kapcsolatát, szerepét, az egyes felületeken elérhető funkciókat. A fejezet célja, hogy a rendszer működéséről átlátható képet adjon.

Az alkalmazás forráskódja Java és JavaFx nyelven került implementációra, az adatbázis tervezéskor, megvalósításkor a MySql relációs adatbázis-kezelő rendszert alkalmaztam. A webszolgáltatás teljes körű támogatását a GlassFish alkalmazásszerver nyújtja.

1. A genetikus algoritmus

1.1. Definíció

„A genetikus algoritmus egy olyan keresőalgoritmus, amelynek alapja a természetes szelekció, és természetes géntechnológiák, eredménye pedig egy olyan hatékony keresőalgoritmus, amely az emberi keresési stratégia újító hajlamait tartalmazza. A genetikus algoritmusokat megalapozó hasonlat a természetes evolúció hasonlata. Az evolúció során az egyes fajok feladata az, hogy minél jobban alkalmazkodjanak egy bonyolult, ráadásul állandóan változó környezethez. A tapasztalat, melyet az egyes fajok az alkalmazkodás során szereznek, beépülnek a kromoszómákba, és azok tovább öröklődnek.”[1]

A genetikus algoritmusokat olyan keresési, optimalizálási feladatok esetében alkalmazzák, amelyeknek megoldására vagy nem ismerünk megfelelő algoritmust, vagy ha ismerünk is az nem gyors nem szolgáltat hatékony megoldást.

Általában nem használ területfüggő tudást, így akkor is működik, ha a feladat struktúrája kevéssé ismert. Egyszerre több pontos keresést végez a problématérben, ez egyrészt kellő robusztusságot biztosít, hiszen ha az egyik keresési ág zsákutcába vezet, abból még nem következik az algoritmus kudarca. Másrészt mivel több, az optimálishoz közel álló megoldást eredményez lehetőséget biztosít a felhasználó számára a legmegfelelőbb kiválasztására.

1.2. A természetes kiválasztódás - A genetikus algoritmus

Darwin előtt úgy gondolták, Isten megteremtette az összes állatot és növényfajt, és a fajok azóta lényegében változatlan formában élnek a Földön. Charles Darwin „A fajok eredete” című könyvében fogalmazta meg először a természetes kiválasztódás gondolatát, a fajok és egyedek közötti szüntelen versengést. Ennek során a gyengék, életképtelenek elhullnak, kihalnak, míg azok akik képesek a folyton változó környezethez alkalmazkodni túlélnek, fennmaradnak genetikai állományukat örökítik.

A természet számára a keresztezés, a mutáció, a természetes kiválasztódás biztosítja az örökös megújulást az életképesebb egyedek fennmaradását.

A keresztezés során egy újabb egyed jön létre mely, örökli szülei tulajdonságainak egyik, illetve másik részét. A keresztezési folyamatban általában az életképesebb egyedek vesznek részt ezáltal rátermett utódot hoznak létre.

A mutáció során egy egyed tulajdonságainak változása következik be, amely ha véletlenszerű, akkor hagyományos mutációról beszélünk, ha a tulajdonságok egy részének megfordulása következik be, akkor inverzióról beszélhetünk.

Az erőforrások korlátozottsága miatt az egyes egyedeknek versenyezniük kell egymással, a versenyből azok kerülnek ki győztesként, akik rátermettebbek a többinél, ezt a folyamatot nevezzük természetes kiválasztódásnak.
Lényeges kiemelni az evolúció és a genetikus algoritmus közti fő különbséget: míg az evolúció esetében a legfontosabb szempont a dinamikusan változó környezethez való alkalmazkodás, addig a genetikus algoritmus esetében a lényeg az optimalizáció, azaz egy rögzített szempontból a legjobb megoldás megtalálása.

A genetikus algoritmusok a szakkifejezéseket a genetikából vették át. A populáció tagjai az egyedek. Az egyedek génekből állnak, minden gén egy tulajdonság öröklődéséért felelős. Az aktuális populációból minden lépésben egy új populációt állít elő úgy, hogy a legrátermettebb egyedekre alkalmazza a keresztezést és mutációt. Az alapgondolat az, hogy mivel általában minden populáció az előzőnél rátermettebb elemeket tartalmaz, a keresés folyamán egyre jobb megoldások állnak elő.

1.3. Történelmi áttekintés

A hatvanas években merült fel először az a gondolat, hogy az evolúcióban megfigyelhető szelekciós folyamatok mintájára különböző mérnöki, biológiai, fizikai, kémiai és egyéb optimalizálási feladatok megoldására alkalmas számítógépes modelleket hozzanak létre.

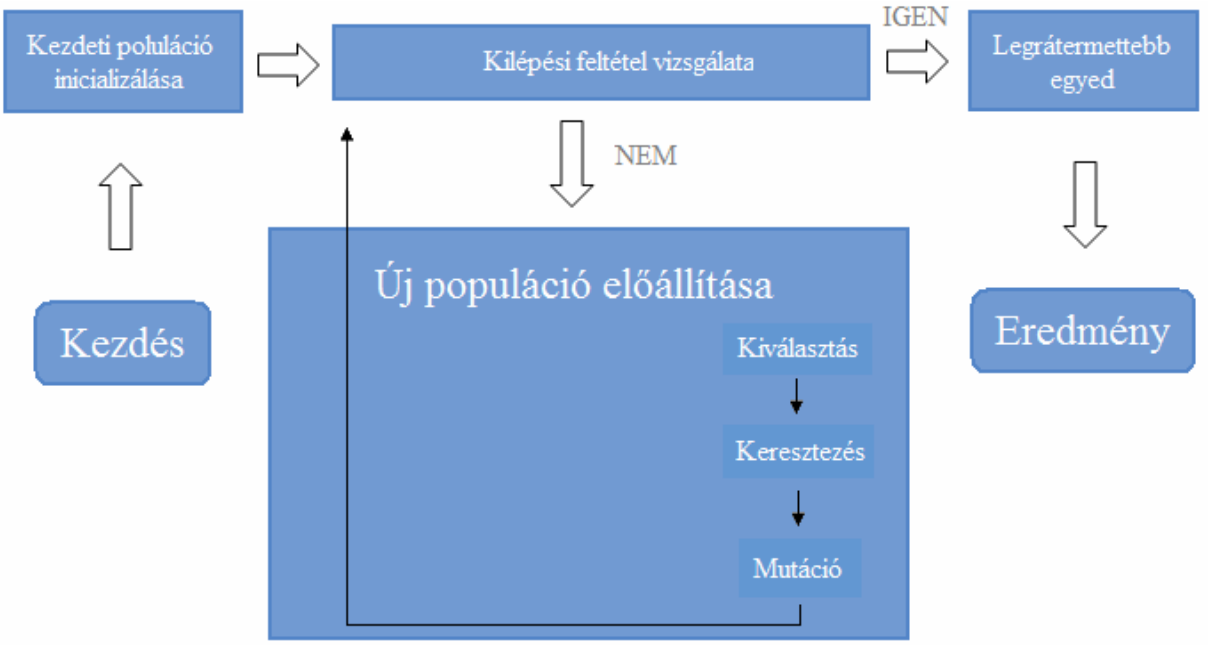
Egymástól függetlenül több próbálkozás is született. Rechenberg az 1960-as években vezette be az evolúciós stratégiáknak (Evolutionary Strategies, ES) nevezett módszert, ezen módszert pl. szárnyszelvények valós paramétereinek az optimalizálására használta. Ez a módszer egy szülő és egy gyerek paramétervektorral dolgozott, az utóbbi a szülő mutáltja volt, ami azt jelenti, hogy néhány paraméter véletlenszerűen módosításra került a vektorban.
1966-ban Owens és Walsh által publikált cikkükben ismertettek egy új evolúciós technikát az evolúciós programozást (Evolutionary Programming, EP) ez a technika egyszerű problémák megoldására szolgáló véges automaták automatikus kifejlesztésével kísérletezett.
Genetikus programozás (Genetic Programming, GP) az evolúciós programozáshoz hasonló elvekre épül, de egy frissebb kutatási terület. Ez lényegében a genetikus algoritmus egy speciális alkalmazási területe, amikor is a cél meghatározott feladatokat végrehajtó számítógép programok (leggyakrabban LISP nyelven) automatikus kifejlesztése. Az első ilyen jellegű próbálkozások Koza (1990-es évek) nevéhez fűződnek, aki ma is a terület vezető alakja.

Témánk szempontjából a legfontosabb kutatási terület a *Genetikus Algoritmusok* (Genetic Algorithms, GA) melynek alapjait John Holland fektette le a Michigani Egyetemen az 1960-as, 70-es években. Holland eredeti célja nem az volt hogy az optimalizációra dolgozzon ki algoritmust hanem, hogy olyan módszereket dolgozzon ki, amelyekben a természetes adaptáció mechanizmusa bevihető lenne a számítógépes rendszerekbe. 1975-ben megjelent publikációjában (Adaptation in Natural and Artificial System) kifejtett elvek lényegét részletesen a következő fejezetben ismertetem.

A fent említett területek gyűjtőneve evolúciós számítások (Evolutionary Computation). Ezek a területek a mai napig megőrizték identitásukat, de megfigyelhető az egyre élénkebb információcsere a területek között, illetve az, hogy a fő komponensek és alapelvek lényegében megegyeznek.

1.4. A genetikus algoritmus

A genetikus algoritmusoknak nem létezik olyan definíciója, amelyet minden, e kutatási területtel foglalkozó közösség elfogadna. Azonban minden módszer, amelyet genetikus algoritmusnak hívnak, tartalmazza a következő elemeket: kromoszómákból álló populáció, jósági mérték alapú kiválasztás, keresztezés új utódok létrehozására és véletlenszerű mutáció az új egyedekben. Ezekhez még egyes kutatók hozzáveszik a Holland által javasolt inverziót is, de ennek használata nincs kellőképpen megalapozva.



1. ábra: Probléma megoldása genetikus algoritmussal

A genetikus algoritmus egy véletlenszerű egyedekből álló kezdeti populációval indul. A fő ciklus minden egyes iterációja során a jelenlevő egyedekre alkalmazzuk a természetben is előforduló genetikai módosulások megfelelőit, a kiválasztás történhet véletlenszerűen vagy a jóság függvény alapján. Az így előálló egyedeknek kiértékeljük a jóság függvényét s a kapott eredmények alapján létrehozunk egy új populációt. Az új populáció állhat kizárólagosan az új egyedekből vagy az új és a szülő egyedek együtteséből jóság függvényük alapján kiválasztva. A ciklust ismételjük mindaddig, míg az optimalizálási feltételek nem teljesülnek.

1.4.1. Reprezentáció

A genetikai algoritmusok esetében fontos szerepet tölt be a reprezentáció - a kódolás - mivel új egyedeket a kromoszómákra alkalmazott különböző genetikai operátorok alkalmazásával hozhatunk létre. Ettől függ, hogy működni fog-e a természetes kiválasztódás mechanizmusa vagy sem.

A reprezentációra több lehetőség is adott. Egy klasszikus megoldás, ha a kromoszómát rögzített hosszúságú bináris sztringnek tekintjük, azaz az egyes megoldásokat egyszerű bitsorozat $\left(x_i, i : \{0,1 \cdots L-1\}, x_i \in \{0,1\}\right)$ reprezentálja, ahol L sorozat hossza, rögzített az egész algoritmus alatt. Ezen reprezentációval kapcsolatban felmerül a kérdés, hogy az adott probléma egyes megoldásait milyen leképezés segítségével feleltessük meg a 2^L darab lehetséges bináris sorozatnak. Egy lehetséges megoldás lehet, ha az egyes megoldásokat egy-egy sorszámmal $\left(j\right)$ megjelölve felsoroljuk. Ebben az esetben a hagyományos bináris kódolás alkalmazható: $j = \sum_{i=0}^{L-1} 2^i x_i$.

1.4.2. Inicializálás

A genetikus algoritmusok futásának fontos tényezője a populáció, amely az egyedek egy halmaza. A populáció az algoritmus futása során minden iterációs lépés során változik újabb és újabb populációk jönnek létre.

Mivel a genetikus algoritmusok a gyenge módszerek közé tartoznak, azaz semmilyen a priori ismerettel nem rendelkeznek, ezért az inicializáció során az egyedek génjeit a lehetséges értékkészletből teljesen véletlenszerűen választják meg egyenletes eloszlás szerint.

A fentebb ismertetett reprezentáció esetén L hosszú bitsztringeket használhatunk egyedenként. Mivel minden bitet 50-50% valószínűséggel állíthatunk 0-ra vagy 1-re, ezért minden bitsztring minta $\frac{1}{2^L}$ valószínűséggel adódhat egy egyed esetén.

A populáció alapvető jellemzője a méret, a könnyebb kezelhetőség érdekében célszerű a méretet az algoritmus működése során állandónak választani.

1.4.3. Kiértékelés és jóság függvény

A genetikai algoritmusok esetében ugyancsak fontos szerepet tölt be kiértékelési függvény vagy célfüggvény, illetve az ebből származtatott fitnessz függvény. Ezek határozzák meg az egyes egyedek jóságát, az algoritmus működéséhez nélkülözhetetlen vezérlő mértéket, mely alapján az egyes egyedeket viszonyítani tudjuk egymáshoz.

Tehát a fitnessz függvény az egyes egyedek által képviselt megoldások jóságát jellemzi, a megfelelő működéshez a függvénynek a valóságot jól kell tükröznie, azaz egyrészt jobb megoldáshoz nagyobb értéket kell, hogy rendeljen, másrészt értékének jól kell mutatnia az egyes egyedek közötti jóságok arányát.
Az algoritmus hatékony működéséhez célszerű minél kisebb számítási bonyolultságú függvényeket konstruálni.

A továbbiakban a különféle fittnessz függvényeket ismertetem.

- **Közelítő fittnessz függvény**

  Ebben az esetben eredeti fitnessz függvényt - amely az adott problémát pontosan leírja, de túl bonyolult és időigényes számításokat végez - jól közelítő függvényt alkalmazunk, amely gyorsabban vezet optimális megoldáshoz. Jobban járunk, ha egy kevésbé pontos közelítő függvényt alkalmazunk, amely több generációt tud kiértékelni, mintha az eredeti függvényt alkalmaztuk volna, amely ugyanannyi idő alatt kevesebb generációt tud kiértékelni.

- **Részcélokat kezelő fitnessz függvény**

  Alapgondolata a következő: egy célt több kisebb részcéllá bontunk, majd egy pont fitnessz értékét a teljesített részcélokból számítjuk ki.

- **Büntetőfüggvény segítségével generált fitnessz függvény**

  A fitnessz függvényt úgy kaphatjuk meg, hogy egy előre meghatározott értékből kivonjuk a büntetőfüggvényt. Azt, hogy egy egyed mennyi ill. milyen mértékben sért meg feltételeket, büntetőfüggvény segítségével írjuk le. Bővebben a következő fejezetben ismertetem.

- **Többértékű fitnessz függvény**

  Többértékű fitnessz függvényt azon esetben alkalmazunk, amikor nem egy függvényt szeretnénk optimalizálni, hanem egyszerre többet, esetleg néhány függvényt optimalizálni, másokat minimalizálni szeretnénk.
  A többértékű függvényeket alkalmazó genetikus algoritmusok egyszerre több optimális megoldást szolgáltatnak.

1.4.4. Kiválasztás

A genetikai algoritmus működése során a kiválasztásnak két formáját figyelhetjük meg, az egyik esetben a populációból egyedeket választunk ki, amelyekre a genetikai operátorokat alkalmazni fogjuk, a második esetben pedig az újabb populációt képező egyedeket választunk ki. Minél nagyobb egy egyed fitnessz függvény értéke, annál nagyobb valószínűséggel kerül majd kiválasztásra, annál nagyobb valószínűséggel örökítheti tulajdonságait.

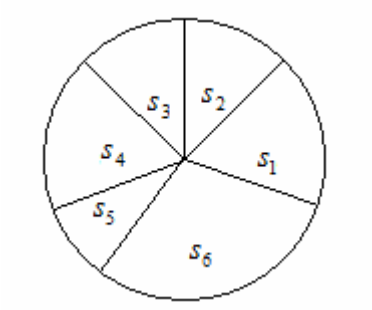
A kiválasztásra több stratégiát is alkalmazhatunk.

- **Véletlen kiválasztásos szelekció**

  A kiválasztási módszer esetében az egyes egyedek véletlenszerűen kerülnek kiválasztásra a populációból. Minden egyed azonos valószínűséggel kerül kiválasztásra. Nem használjuk fel az egyes egyedek fitnessz értéke által nyújtott plusz információt.

- **Fittnesszarányos kiválasztás** (roulette wheal, fitness based)

  Ezen módszer esetében egy egyed kiválasztásának valószínűsége annál magasabb, minél nagyobb a rátermettsége a populáció átlagához képest.
  Megvalósításához az egyik legrégebbi, legnépszerűbb technikát alkalmazzuk, mely esetében egy képzeletbeli rulett kereket az egyes egyedek jóságával arányos cikkekre osztjuk fel.



2. ábra: Rulett kerék kiválasztás

A kiválasztás esélye a fitnessz értékkel lesz arányos.

$$p(s_i) = \frac{f(s_i)}{\sum_{s_j \in P} f(s_j)}$$, ahol s_i egy adott egyed, f pedig a célfüggvény.

Az eljárás előnye, hogy viszonylag könnyen megvalósítható, figyelembe veszi a szülők rátermettségét. Hátránya, hogy egy nagy rátermettséggel rendelkező egyed aránytalanul sokszor kerül kiválasztásra, ennek kiküszöbölésére alkalmazzák a fitnessz függvény skálázását. Például, ha egy fitnessz függvény exponenciális, akkor egy megfelelő függvénnyel lineárissá tehető.

- **Lineáris rangsorolás** (linear ranking selection)

  Az egyedeket rátermettségük alapján sorba rendezzük, a legrosszabb értékűtől haladunk a legjobb felé. Ez a módszer a fitnessz értéket alapul vevő kiválasztás szóródásbeli hátrányait próbálja javítani. A sorszámozás egyedi, tehát az azonos fitnessz értékű egyedek sorszáma eltérő. Az egyes egyedek kiválasztásának valószínűségét csak a sorban elfoglalt helye határozza meg.

  $$p(s_i) = p_{\max} + (i-1).\frac{p_{\max} - p_{\min}}{N-1}$$, ahol N a populáció mérete.

- **Pár-verseny szelekció** (binary tournament selection)

  Ezen módszer nem fitnessz arányos szelekció, egy μ lépéses ciklusból áll. Minden lépésben kiválasztunk T számú egyedet véletlenszerűen a populációból, majd az így kiválasztott egyedek közül a legrátermettebbet választjuk ki nyertesnek. Pár-verseny szelekcióról T=2 esetén beszélhetünk.
  A módszer esetében a konvergencia sebességét a négyszeresére emelhetjük a rulett kerék kiválasztás módszeréhez képest.

- **Egyszerű vagy generációs kiválasztás** (simple, generational selection)

  Az eljárás alkalmazása során a teljes populációt lecseréljük, a szülők elvésznek helyüket az utódok veszik át.

- **Elitizmus**

  Az aktuális populáció legrátermettebb egyedét elit egyednek nevezzük. Azt a módszert, mely során az elit egyedet változtatás nélkül tovább örökítjük az új populációba elitizmusnak nevezzük. Jobb rátermettséggel rendelkező egyed nagyobb valószínűséggel örökítheti tovább tulajdonságait. Azon felmerülő probléma elkerülése

érdekében, hogy egy hibás elit egyed miatt a keresés rossz irányba induljon el, célszerű véletlenszerűen egy egyedet választani elitnek, a hasonlóan magas rátermettségi értékkel rendelkező egyedek közül.
Az elitizmus alkalmazásával jelentős hatékonyság növekedés érhető el.

1.4.5. Genetikai operátorok

1.4.5.1. Keresztezés

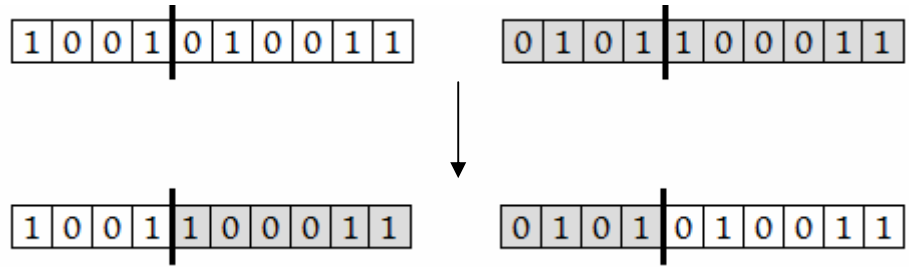
A genetikai algoritmusok fontos része a keresztezés, melynek során a két szülő egyed tulajdonságait átörökíti az új egyedbe. Az utódok generálása során nem minden esetben történik meg a szülők tulajdonságainak kereszteződése, lehetőséget biztosítva így arra, hogy a szülő tulajdonságai változatlanul kerüljenek át az új populációba.
Keresztezés során a két szülő egyedből egy jobb tulajdonsággal rendelkező új egyedet szeretnénk létrehozni. Ez azonban nem valósul meg minden esetben, mivel az operátorok nem vizsgálják meg az egyes egyedek belső szerkezetét, nem tudnak tulajdonságok között válogatni.

A továbbiakban a különféle keresztezési módszereket ismertetem.

- **Egypontos keresztezés**

  Ezen operátor a két szülő egyed esetében véletlenszerűen meghatároz egy i pozíciót $(1 \le i \le n)$ és itt szétvágja azokat, majd az utód az első szülő kezdő részéből és a második szülő második részéből épül fel. Lehetőség van egy második utód létrehozására is, ez analóg módon a szülők másik feléből épül fel.

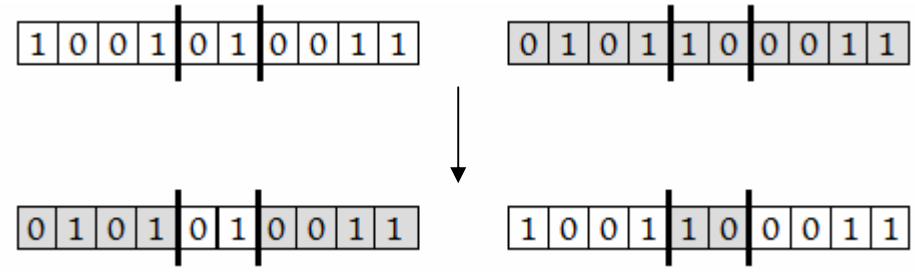
  Szemléletesen:



- **Kettő vagy több pontos keresztezés**

  Ezen esetben nem egy, hanem két vagy több vágási pontot fogunk meghatározni és e pontok szerint előálló részekből fogjuk az új egyedeket meghatározni.

Szemléletesen:



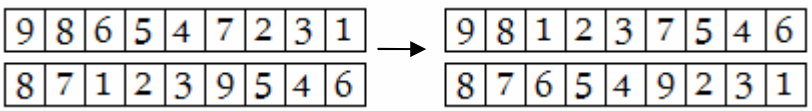
Több vágási pont alkalmazása esetén a szülők tulajdonságai egyre inkább összekeverednek. Ez egyrészt pozitívum, mivel így nagyobb a genetikai változatosság, másrészt negatívum, mert ha egy szülő pozitív tulajdonságai egymás mellett helyezkednek el, akkor a sok vágási pont hatására egyre inkább eltávolodnak egymástól, ezáltal a keresztezés nem eredményezhet a szülőnél rátermettebb utódot.

Több vágási pont alkalmazása mellett szól az a tény, hogy ha a populáció egyre inkább konvergál, akkor az egypontos keresztezés hatására a szülőtől kismértékben eltérő utódok keletkezhetnek. Ezt elkerülhetjük, ha úgy módosítjuk a programot, hogy ilyen esetben több vágási pontot határozzon meg és a keresztezést így alkalmazza.

- **Ciklikus keresztezés**

A ciklikus keresztezés más megközelítést használ, nem használ keresztezési pontokat. Alapja az az észrevétel, hogy minden pozícióra vagy az egyik, vagy a másik szülő eleme kerül. Először az első szülő első elemét átmásoljuk az első leszármazottba, majd megnézzük, hogy a második szülőben mi szerepel az első helyen és ezt megkeressük az első szülőben. Átmásoljuk az első leszármazottba, majd megnézzük, hogy a második szülőben mi van ezen a helyen, ezt ismét megkeressük az első szülőben. Mindez addig folyik, míg egy olyan elemhez nem érünk, melyet már egyszer átmásoltunk, ekkor az első szülő első elemétől induló ciklussal körbeértünk. Az első leszármazott helyét a második szülőből töltjük fel az azonos helyekről.

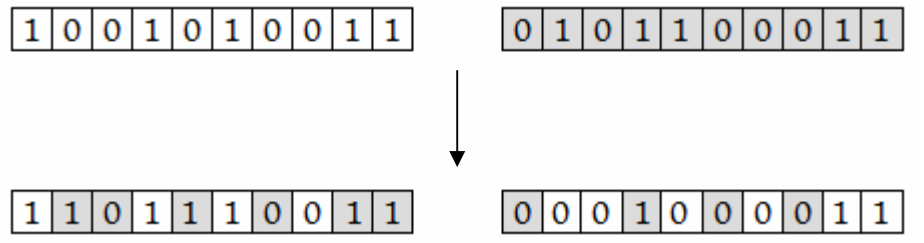
Szemléletesen:



- **Egyenletes keresztezés**

  Ebben az esetben alkalmazunk egy keresztező maszkot, mely ugyanolyan hosszú, mint az egyes egyedek. Értékei: 0 és 1 lehet. Kezdetben véletlenszerűen határozzuk meg az elemeit.
  Az eljárás működése a következő: ha 1-es található a maszkban, akkor az első leszármazottba az első szülő tulajdonsága kerül, a második leszármazottba pedig a második szülő tulajdonsága. Abban az esetben, ha 0-ás található a maszkban, akkor az első leszármazottba a második szülő tulajdonsága, a második leszármazottba az első szülő tulajdonsága kerül.

  Szemléletesen:



A módszerek közül, hogy melyik a leghatékonyabb nagyban függ a probléma típusától, illetve az algoritmusban használt más tényezőktől is. Viszont általánosságban a következő mondható el: nagy populáció esetében a 2-pontos keresztezést érdemes használni, kisebb populációk esetében az egypontos keresztezés használata az előnyösebb.

1.4.5.2. Mutáció

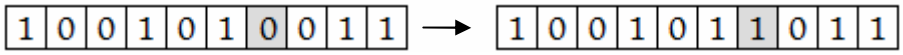
A második alapvető genetikai operátor a mutáció, mely meglévő egyedeket módosít úgy, hogy tulajdonságait véletlenszerűen megváltoztatja. A mutációk valószínűségét elég alacsonyan szokták meghatározni - 0,001–0,01 között - azért, hogy a mutáció hibás használata miatt ne keletkezhessen túl sok hibás egyed, ezzel elrontva a gondosan felépített populációt.

A továbbiakban a különféle mutációs módszereket ismertetem.

- **Bit-flip mutáció**

  Ez a mutációs operátor a legegyszerűbb, az egyedekben véletlenszerűen megváltoztatunk adott számú tulajdonságot.

Személetesen:

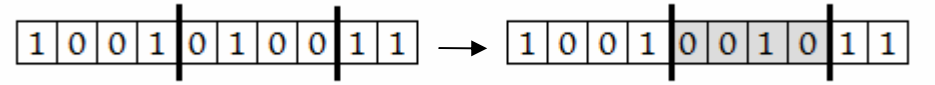


A módosítás kétféleképpen valósítható meg. Az egyik során megadjuk a módosítani kívánt tulajdonságok számát, a módosítás pozícióját pedig véletlenszerűen határozzuk meg. A másik lehetőség során pedig egy gyakoriságot adunk meg, amellyel az egyedben a biteket megváltoztatjuk.

- **Inverzió**

  A mutáció egy speciális fajtájának tekinthetjük az inverziót, melynek során meghatározunk két pontot és a közé eső intervallumot megfordítjuk.

  Szemléletesen:



Ezen operátor alkalmazása a szakirodalom szerint nagy valószínűséggel nem eredményez életképes egyedeket.

1.4.6. Kilépési feltétel

Az algoritmus működése során a kilépési függvény határozza meg azt, hogy a megoldandó probléma szempontjából optimális megoldást kaptunk-e. Ennek eldöntésére többféle megközelítés is létezik:

- Elértünk egy előre meghatározott generációszámot.
- A legrátermettebb egyed és a populáció átlagos rátermettségének aránya elért egy előre meghatározott értéket.
- A populáció egyedei konvergálnak.

1.4.7. A genetikus algoritmus korlátai

A genetikus algoritmusok nem minden probléma esetében képesek optimális megoldást szolgáltatni, szemben az adott területre specifikus módszerekkel. Az algoritmus működését nagyban befolyásolják az alapvető paraméterek: reprezentáció, populációk száma, kereső operátorok, kilépési feltétel. Ezen jellemzőket csak az adott probléma szerkezetének mélyreható ismeretével, illetve az előző eredmények kiértékelésével állíthatjuk be megfelelően.

1.5. Az órarendkészítés és a genetikus algoritmus kapcsolata

1.5.1. Miért pont a genetikus algoritmus?

A genetikus algoritmusokat olyan keresési, optimalizálási feladatok esetében alkalmazzák, amelyeknek megoldására vagy nem ismerünk megfelelő algoritmust, vagy ha ismerünk is az nem gyors, nem szolgáltat hatékony megoldást.

Az órarend készítés problémája az NP-teljes feladatosztályba sorolható. NP-teljes feladatok esetében nem ismerünk olyan algoritmust, amely polinomiális időn belül elvégezné a feladatot. A feladat megoldásának ideje a bemenet méretének exponenciális függvénye lesz.
A példa kedvéért vegyünk egy iskolát, melynek 6 évfolyama van és minden évfolyamon 3 tagozat, ez 18 osztály. Legyen minden osztálynak 8 különböző órája heti 3 alkalommal, ez 24 óra, valamint tételezzük fel, hogy heti bontásban 24 fix órahelyünk van. Ezen adatokat alapul véve 24^{3*8*18} különböző órarend adódik. A hagyományos algoritmusok számára ekkora méretű problématér feldolgozása igencsak időigényes volna.

Mivel a genetikus algoritmusok egyszerre több pontos keresést végeznek a problématérben, ez kellő robusztusságot biztosít számukra, illetve a párhuzamosság által a keresési idő is csökken.

1.5.2. Hagyományos órarend reprezentáció

A hagyományos órarend reprezentáció esetében egy kétdimenziós ábrázolási módot veszünk alapul, melynek a függőleges tengelyén az osztályok szerepelnek, a vízszintes tengelyen pedig órákra felosztva az időt ábrázoljuk.

A kétdimenziós mátrix (i, j) mezője azon tanárt tartalmazza, aki az i-edik osztályt tanítja a j-edik órában. Természetesen a hét több napból áll így az időtengelyen az egyes napok órabeli osztását egymás után vesszük fel. A hét egy bizonyos időpontját a következő képen érhetjük el: $j = N * (n-1) + o$, ahol N az egyes naponkénti tartott órák száma, n a nap, az o pedig a napon belüli időpont.

Szemléletesen:

	1. nap				…	*n.* nap			…
	1. óra	2. óra	…	*N*. óra	…	…	*j*. óra	…	…
1. osztály									
2. osztály									
⋮					⋱				
i. osztály							X. Y.		

Ez az ábrázolási mód biztosít egy implicit kényszert, amely bizonyos feltételeknek ellentmondó órarend készítését eleve megakadályozza. Szemléletesen jelzi az ütközéseket, így azokat fel sem vihetjük az órarendbe. Például, jól mutatja, hogy két tanár egy időpontban nem oktathatja ugyanazt az osztályt.

Nem támogatja az olyan esetek ábrázolását, - bontott órákat - amikor egy osztályt egy időben két tanár oktat, például nyelvi órák esetében a kezdőket egy tanár, a haladókat egy másik tanár oktatja. Ezt az esetet hasonlóan kezeli, mintha két eltérő tantárgyról lenne szó, az ábrázolás egyáltalán nem tesz különbséget a tantárgy és a tanár fogalma közt.

A tantárgy és a tanár fogalma közt nélkülözhetetlen volna különbséget tenni az olyan esetek kezelésekor is, amikor egy tanár esetleg több tantárgyat oktat egy osztálynak, valamint az órarend tervezése során figyelembe kéne venni a tantárgyak megfelelő heti eloszlását is.

1.5.3. Halmazos órarend reprezentáció

Ezen reprezentáció alapvető építő egysége a halmaz, ez egy olyan struktúra, amely tetszőleges számú osztályt, tanárt, tantárgyat és osztálytermet tartalmaz. Szemantikája a következő: adott időpontban, a halmazban felsorolt osztályokat a felsorolt tanárok oktatják, a megadott termekben. Arról nem tartalmaz konkrét információt, hogy pontosan mely tanár, mely osztályt hol oktatja.

Az említett struktúra alkalmazása lehetővé teszi az összevont és bontott órák teljekörű kezelését. Például a fentebb említett nyelvi óra esetében a halmazban, felsoroljuk az osztályt és a két tanárt, akik az adott osztály egyik felét kezdő, másik felét haladó szinten oktatják.
Az az eset is könnyen kezelhető, amikor tornaóra alkalmával, több osztályt oktat a tanár a tornateremben.

A halmazos órarend reprezentáció nem két, hanem csak egy dimenziós ábrázolási módot alkalmaz, melynek egyetlen tengelye az idő, az egyes napok órabeli osztását egymás után

vesszük fel. Az időtengelyen ún. periódusok állnak egymás mellett, amelyek a tanórák lehetséges idejét szemléltetik. A periódusokba vesszük fel a halmazokat, azaz az egy periódusban szereplő halmazok órái azonos időben lesznek megtartva.

Szemléletesen:

1. nap			2. nap			…
1. óra	2. óra	…	…	…	…	…
Halmaz_1	Halmaz_{N+1}					
Halmaz_2	Halmaz_{N+2}					
⋮	⋮	⋱				
Halmaz_N	Halmaz_{2N}					

Mint láthatjuk egy periódusba több halmaz is kerülhet, azonban ügyelnünk kell arra, hogy ne sértsünk meg bizonyos feltételek, például egy tanár egy perióduson belül csak egy halmazban szerepelhet. Az is látható, hogy minden periódusban legfeljebb N darab halmaz szerepelhet, ez a megkötés a genetikus algoritmus reprezentációnál lesz fontos.

1.5.4. Linearizálás

A genetikai algoritmusok esetében fontos szerepet tölt be a reprezentáció - a kódolás - mivel új egyedeket a kromoszómákra alkalmazott különböző genetikai operátorok alkalmazásával hozhatunk létre. Ettől függ, hogy működni fog-e a természetes kiválasztódás mechanizmusa vagy sem.

A reprezentáció során az egyed tulajdonságait célszerű kódolva, lineárisan egymás után felírva tárolni. A halmaz elméletű reprezentáció fentebb említett formája ennek a felírási módnak nem felel meg, mert egy plusz második dimenziót feszítenek az egyes periódusokban elhelyezkedő halmazok. Erre a látszólagos második dimenzióra a reprezentáció során nincs szükségünk, csak azt kell tudnunk, hogy az egyes halmazok mely periódusba tartoznak. Ezért a halmazos reprezentáción egy transzformációt kell végrehajtanunk, melynek hatására már csak egy dimenziót fog az tartalmazni.

A továbbiakban a különféle linearizálási módszereket ismertetem.

- **Függőleges linearizálás**

  Ezen módszer esetében az időtengely mentén sorba felvesszük egymás után a halmazokat. Először az első periódus első halmazát, majd második halmazát, így haladva az utolsó periódus utolsó halmazáig.

Szemléletesen:

1. periódus	2. periódus	…	k. periódus
Halmaz $_1$	Halmaz $_{N+1}$		Halmaz $_{(k-1)N+1}$
Halmaz $_2$	Halmaz $_{N+2}$		Halmaz $_{(k-1)N+2}$
⋮	⋮		⋮
Halmaz $_N$	Halmaz $_{2N}$		Halmaz $_{kN}$

↓

Halmaz $_1$	Halmaz $_2$	…	Halmaz $_N$	Halmaz $_{N+1}$	…	Halmaz $_{kN}$

Így egy lineáris struktúrát kapunk, és mivel megköveteltük, hogy minden periódusban N darab halmaz szerepeljen - ha szükséges a periódusba felvehetünk ún. üres halmazokat - a leképezés kölcsönösen egyértelmű lesz. A lineáris struktúrában az $(i-1)*N+1$-edik pozíciótól az $i*N$-edik pozícióig terjednek az i-edik periódus halmazai.

- **Vízszintes linearizálás**

A vízszintes linearizáció során, a függőleges linearizációtól eltérően nem az egyes periódusok halmazait vesszük fel sorban egymás után, hanem az egyes periódusok első, majd a második stb. halmazait mérjük fel egymás után.

Szemléletesen:

1. periódus	2. periódus	…	k. periódus
Halmaz $_1$	Halmaz $_{N+1}$		Halmaz $_{(k-1)N+1}$
Halmaz $_2$	Halmaz $_{N+2}$		Halmaz $_{(k-1)N+2}$
⋮	⋮		⋮
Halmaz $_N$	Halmaz $_{2N}$		Halmaz $_{kN}$

↓

Halmaz $_1$	Halmaz $_{N+1}$	…	Halmaz $_{(k-1)N+1}$	Halmaz $_2$	…	Halmaz $_{kN}$

1.5.5. Büntető függvény

A genetikai algoritmusok fontos tényezője a kiértékelési függvény vagy célfüggvény, illetve az ebből származtatott fitnessz függvény. Ezek határozzák meg az egyes egyedek jóságát, az

algoritmus működéséhez nélkülözhetetlen vezérlő mértéket, mely alapján az egyes egyedeket viszonyítani tudjuk egymáshoz.

A populáció egyes egyedeinek jóság értékének meghatározásához büntető függvényt alkalmazunk, így leírhatjuk azt, hogy az egyes egyedek mennyire térnek el egy viszonylag optimális megoldástól. Büntető függvény alkalmazásának oka az, hogy a rosszaságot egyértelműen meg tudjuk határozni, definiálni tudjuk, míg a jóságot nem.
Különféle követelményeket határozhatunk meg, amelyeket betartva az optimális eredmény előáll. A követelményeket két csoportba sorolhatjuk:

- Az egyik csoportot a kemény követelmények alkotják, melyek olyan feltételeket tartalmaznak, amelyeket ha az egyed megsért az használhatatlanná válik. Például, nem fordulhat elő, hogy egy tanár egy időpontban egyszerre két helyen tartson órát.

- **Tanár ütközés**

  Tanár ütközés abban az esetben fordulhat elő, ha egy tanárnak egyszerre két helyen kellene órát tartania. Ilyenkor az adott órarend használhatatlan, mert a tanár fizikailag csak egy helyen lehet egyszerre. Kivételt képeznek azon esetek, amikor egy tanár több osztálynak tart órát egy teremben. Bontott órák során az egyes osztályok tanulóinak csak egy része vesz részt a közös órán. Ebben az esetben a halmazba felvesszük: a tanárt, a tantárgyat, a résztvevő osztályokat, valamint a termet.

- **Osztály ütközés**

  Ezen esetről akkor beszélünk, amikor egy osztálynak, egy időben több helyen kellene megjelennie. Ez lehetetlen, mert az osztály tanulói egyszerre csak egyféle órán tudnak részt venni. Ezen eset alól kivételt képez az, amikor egy időpontban az osztály egyik fele egy órán vesz részt, másik fele pedig egy másikon. Osztály ütközés során a halmazba felvesszük: az osztályt, a tantárgyakat a megfelelő tanárokkal és a tantermeket is.

- **Terem ütközés**

  Terem ütközésről akkor van szó, amikor egy teremben, egy időpontban egyszerre több óra is folyik. Ilyen nem fordulhat elő, mert egy terembe csak egy osztályt ültethetünk le. Ez alól is van kivétel, amikor egy óra keretén belül az oktató egyszerre több osztályt is tanít. Ekkor a halmazba felvesszük a termet, az oktatót vagy oktatókat, a tantárgyat és az osztályokat.

- A másik csoportot a lágy követelmények alkotják, melyek olyan feltételeket tartalmaznak, amelyeket az egyed megsértve az még működőképes marad, csak kényelmetlen az oktatásban résztvevő felek számára.

- **Tanárok ráérése**

  Előfordulhat olyan eset, hogy a tanár egyéb elfoglaltságai miatt nem ér rá a teljes tanítási időben. Ekkor lehetősége van év elején egy ráérési skála segítségével meghatározni, hogy az egyes időpontok mennyire felelnek meg neki. Ha azt állítja be, hogy egy időpont számára nem jó, akkor az erős követelmény és az ezt megsértő órarend használhatatlan, viszont ha a skálán egy kisebb értéket állít be, akkor az lágy követelmény, az algoritmus megpróbálja figyelembe venni az optimális eredmény előállítása során.

- **Nulladik óra, lyukas óra**

  A nulladik órák vagy lyukas órák jelenlétét jobb elkerülni egy órarendben, ha mégis fennállnak, akkor büntető pontokat kapnak az órarendek. Ilyen órák akkor jelentkeznek, amikor az algoritmus az erős követelményeket figyelembe véve alakítja az órák menetét, például egy lyukas óra beszúrásával a tanár ütközés megoldottá válik.

- **Többszörös órák**

  Az órarendben egy óra az alap időegység, de az oktatásban előfordulnak olyan órák melyek, ennél többet igényelnek. Ekkor az azonos órák megtartása egymás után következik. Például informatika tagozatos osztályoknál akár 4-5 laborgyakorlat is követi egymást. A többszörös órákat egymástól elválasztva, az órarend büntető pontokat kap.

- **Az órák héten belüli egyenletes elosztása**

  Az órarend készítésekor célszerű szem előtt tartani, hogy az egyes órák a héten belül egyenletesen legyenek szétosztva. Ez a reprezentáció a köztes napok beszúrásával segíti a diákokat a tudás elmélyítésében. Ha az órarend ezt a megkötést megsérti büntető pontot kap.

- **Az óraszám héten belüli egyenletes elosztása**

  Az ideális megoldás az, ha a heti óraszám közel egyenletesen oszlik el a különböző napok közt. Ha a heti óraszám nem osztható a napok számával, akkor megengedünk ± 1 óra ingadozást, viszont ezt az ingadozást átlépő órarendet büntető pontokkal látjuk el.

1.6. A genetikus algoritmus megvalósítása

Az alap algoritmus megvalósítása során a Java nyelvű JGAP (Java Genetic Algorithms Package) keretrendszert alkalmaztam, amelyről az irodalomjegyzék [7] pontjában szereplő címen olvashatunk bővebben. Választásom azért esett erre a keretrendszerre, mert előzetes, kevésbé komplex problémák megoldása során a rendszer nagyfokú modularitását, átgondolt, átlátható felépítését tapasztaltam, mely részletes dokumentációval rendelkezik.

1.6.1. A genetikus algoritmus finomhangolása

Az alap algoritmus csak az optimalizáció főbb lépéseit határozza meg, amelyen belül tág lehetőségünk van a részletek megválasztására ezáltal is növelve a keresés hatékonyságát.

- A feltételeknek leginkább megfelelő eredmény megállapítása során az első kérdés, amely az algoritmus működését döntően befolyásolja a populációméret és a generációszám viszonya. Ezen két tényező egymással szoros kapcsolatban áll, a futási időt nagymértékben meghatározzák. Abban az esetben, ha a populáció méretét túl alacsonynak határozzuk meg, akkor az algoritmus nem lesz képes egy esetleges zsákutcából kijönni, ezáltal az alacsony populációméretű generációk viszonylag gyors kiszámítási ideje nem tud érvényesülni. Ha viszont a populáció méretét túl magasnak határozzuk meg, akkor a kezdeti generációk között a kiértékelés szempontjából jelentős javulás tapasztalható, de mindinkább haladva a feltételeknek megfelelő egyed felé az egyes generációk közt a javulás lecsökken, a nagy populáció már nem képes jobb egyedeket kitermelni. A nagy populáció megválasztása ellen szól még az egyes generációk kiértékelésének hosszabb ideje, amelyek összeadódva a futási időt nagyban megnövelik.

  Mind kicsi, mind nagy populáció méretet választva a keresési folyamatban előrehaladva az egyes generációk közötti különbség egyre inkább lecsökken. Ezt a problémát a generációszám megnövelésével oldhatjuk meg. A generációk egészen addig követik egymást, amíg a feltételeknek teljesen megfelelő egyed elő nem áll, vagy elértünk egy előre meghatározott generációszámot, vagy egy futási időkorlátot.

  A fentebb említettek alapján, az implementálás során a populációszámot 30 - 40 nagyságúnak volt célszerű megválasztanom, az 1 – 1,5 milliós generációszám mellett.

- Az **1.4.4.** fejezetben tárgyalt különböző kiválasztási stratégiák hatékonyságának vizsgálata alapján a választásom verseny szelekcióra esett, mivel érzékelhető javulást mutatott a többi kiválasztási stratégiához képest. Minden lépésben `populációszám/2` darab egyedet választok véletlenszerűen a populációból, és ezek versenyeztetéséből kerül ki a nyertes.

A kiválasztási stratégia megválasztása mellett az elitizmus alkalmazásának kérdése is nagyon fontos volt, ugyanis nélküle az algoritmus eléggé lassan közelítette a feltételeket kielégítő egyedet.

- Az **1.4.5.** fejezetben említett genetikai operátorok hatékonyságának vizsgálata alapján az egypontos keresztezés hatékonyabb működését eredményezte az algoritmusnak, a többi keresztezési operátor alkalmazásához képest. Az inverzió illetve mutáció alkalmazása esetén a mutáció egyértelműen jobb választásnak bizonyult, inverzió esetén nem igazán volt megfigyelhető az egyes generációk közt jelentősebb javulás a kiértékelés szempontjából.

- Az algoritmus legoptimálisabb működésének meghatározásához már csak az **1.5.5** fejezetben tárgyalt követelmények értékének meghatározása szükséges. Azaz az egyes egyedek mennyire válnak használhatatlanná ezeket a feltételeket megsértve. A fő cél nem a büntető pontok pontos meghatározása, hanem azok arányának tisztázása. Legtöbb hibapontot egy egyed akkor kapja, amikor kemény követelményt sért, ekkor ütközés áll fenn, amely esetén az órarend használhatatlan.
  A büntető pontok további nagyság szerinti sorrendje a következő:
  - Tanár ráérés
  - Osztályok óraszáma
  - Osztályok lyukas órái
  - Többszörös órák
  - Órák egyenetlen eloszlása

A `manhattan\src\manhattan\GA\` könyvtár által tartalmazott osztályok valósítják meg a genetikus algoritmust, illetve a fentebb említett finomhangolási tényezőket. A JGAP keretrendszer működéséhez a `jgap.jar` állomány `lib` könyvtárba való importálására van szükség.

2. Webszolgáltatás

2.1. Történelmi áttekintés

A 90-es évek közepe-vége fele az internet robbanásszerű elterjedésnek indult, népszerűbb lett, mint valaha. Az emberek egy időben tájékozódhatnak a világban történt eseményekről, megtalálhatnak rajta bármilyen információt, amelyre szükségük van. A kezdetekkor statikus HTML oldalak álltak az emberek rendelkezésére, de amint a vállalatok felismerték az internetben rejlő lehetőségeket, elkezdtek törekedni arra, hogy szolgáltatásaik az ügyfelek számára online módon is rendelkezésre álljanak. Persze az eltérő rendszerek és architektúrák miatt egy platform és programozási nyelv független eszközt kellett megalkotni, melynek egyik fő célja, hogy támogassa az egyes programok közötti kommunikációt akár emberi beavatkozás nélkül is.

A vezető nagyvállalatok eltérő módon határozzák meg a webszolgáltatás fogalmát. Nézzük meg ezek közül az IBM definícióját:

„A webszolgáltatás egy olyan felület, amely a hálózaton keresztül, szabványos XML üzenetekkel elérhető műveletek egy csoportját írja le. A webszolgáltatások kielégítik egy adott feladat vagy feladatcsoport igényeit. A webszolgáltatáshoz szabványos, formális XML leírás tartozik, amelyet a szolgáltatás leírásának nevezünk, és ez tartalmazza a szolgáltatás igénybevételéhez szükséges összes részletet, többek között az üzenetek formátumát, az átviteli protokollokat és a szolgáltatás pontos helyét. A felület elrejti a szolgáltatás megvalósításának részleteit, így a szolgáltatás használata független az azt megvalósító hardver- és szoftverfelülettől, illetve az adott megvalósítás megírásához használt programozási nyelvtől. Ez teszi lehetővé, hogy a webszolgáltatásokon alapuló programok közötti kapcsolat igen laza maradhasson, maguk a megvalósítások pedig elemekre épülő, többféle eljárást felhasználó megoldások legyenek. A webszolgáltatásokat használhatjuk önmagukban, illetve más webszolgáltatásokkal együtt is, és összetett üzleti tranzakciókat hajtunk végre a segítségükkel”[3]

Ezek alapján a webszolgáltatás egy dokumentum alapú kommunikáció, mely során a felek szabványos üzeneteket küldenek egymás számára, amelyeket feldolgoznak, értelmeznek.
A szolgáltatást igénybe vevő felhasználó szempontjából a szolgáltatást implementáló platform teljesen lényegtelen, ugyanilyen szempontból a szolgáltatásnak sem kell ismernie a felhasználót. Ezen tényezők megvalósulása lehetővé teszi a lazán csatolt komponensekből felépülő rendszer megvalósítását, amely több pozitív jellemzővel bír, mint például a rendszerünk könnyen karbantartható, esetleges új szolgáltatások könnyen beépíthetők illetve

javul a hibakeresés. A webszolgáltatások rugalmas, skálázható, módosítható alkalmazások létrehozását és implementációját teszik lehetővé.

Üzleti szempontból a webszolgáltatás alkalmazása mellett szólnak a következők:

- az új szolgáltatások integrációja a meglévő rendszerbe meglehetősen egyszerűen megvalósítható
- az üzleti partnerek szolgáltatásait építhetjük be saját rendszerünkbe
- a szabványos formátum alkalmazása a fejlesztést átláthatóbbá, hatékonyabbá teszi
- a webes megjelenésnek köszönhetően a szolgáltatások a világon bárhol igénybe vehetők.

A legelterjedtebb webszolgáltatás implementációk:

- REST - Representational State Transfer
- XML-RPC - Extensible Markup Language – Remote Procedure Call
- SOAP - Simple Object Access Protocol

Mindháromnak megvannak a hívei. A SOAP technológia mostanra egy kiforrott, megbízható, tesztelt rendszer lett, míg a REST szoftver-architektúra stílus napjainkban is alakulóban van, de általa lazábban csatolt rendszerkomponensek alkothatók. Diplomamunkám során az utóbbit alkalmaztam, bővebben a következő részben taglalom.

2.2. Representational State Transfer (REST) típusú webszolgáltatás

A Reprezentációs Állapot Átvitel, vagyis REST (Representational State Transfer) a szoftver-architektúra egy stílusa az elosztott rendszerekben, mint például World Wide Web. A REST meglévő szabályokra, protokollokra építkezik, ahelyett, hogy újakat találjon ki. A REST fogalmát Roy Fielding – aki a HTTP protokoll egyik készítője – a 2000-ben megjelent doktori disszertációjában [4] vezette be, amely később a hálózati kommunikációval foglalkozó szakemberek körében egyre ismertebbé vált.

A szoftver-architektúra stílus központi eleme az erőforrás (resource), a kliens és a szerver közötti kommunikációban az erőforrások különböző reprezentációjának átvitele történik. Minden erőforrás rendelkezik egy egyedi azonosítóval, mely alapján a kliens lekéri az erőforrásokat. Az erőforrások megjelenési módjait MIME type-pal adhatóak meg, mely lehet egyszerű szöveg, HTML, XML vagy JSON is. Az erőforrások egyedi azonosítója az URI, ezáltal minden megadható erőforrásként, aminek van URI-je az megcímezhető. A kliens-szerver architektúrát a HTTP protokoll szolgáltatja, alkalmazása esetén biztosítja az állapotmentességet, különböző rétegek kialakítását.

Azon szolgáltatások, amelyek megfelelnek a REST szoftver-architektúra stílusnak RESTful webszolgáltatásnak nevezzük. Fielding a következő alapelveket fektette le a RESTful webszolgáltatásokkal kapcsolatban:

- Kliens-szerver architektúrát kell, hogy megvalósítson. Tehát létezni kell egy szervernek, amely a kliens felől érkező kéréseket figyeli, fogadja és feldolgozza, majd a megfelelő adatokat szolgáltatja.
- Az erőforrás a kliens által elérhető egység, amely megfelelően van ábrázolva.
- Az erőforrások egyedi címmel rendelkeznek, azaz egyedileg azonosíthatók.
- Egy univerzális interfészen valósul meg a kliens és a szerver közötti kommunikáció, az interfész meghatározott műveletekkel bír. Egységes interfész alkalmazása kevésbé hatékony kommunikációt eredményez, mintha az adott alkalmazásra szabott megoldást használnánk.
- A kommunikáció állapot nélküli, a szerverben nem kerül tárolásra semmilyen plusz információ, minden szükséges adat a klienstől érkező kérésben található.

Az erőforrás alapú szemlélet biztosítja, hogy az alkalmazásnak csak az azonosítót és a megfelelő műveletet kell megadni, nem kell ismernie, hogy a kapcsolat hány köztes elemen keresztül valósul meg.

2.3. JAX-RS és Jersey

A JAX-RS egy annotáció vezérelt keretrendszer, stabil formában a Java EE 6 specifikációban található. HTTP annotációkat tartalmaz, melyek az erőforrást és az erőforrást kezelő műveleteket határozzák meg.
A Jersey a JAX-RS referencia implementációja, részletes leírás az irodalomjegyzék [9] pontjában található.

Az JAX-RS annotációk alkalmazását egy egyszerű példában szemléltetem:

```
package model;

import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;

@Path("item")
public class ItemResource {

    @GET
    @Produces("text/plain")
    public String getText() {
        return "Ok!";
    }
}
```

A `@Path` annotáció azon URI-t azonosítja, amellyel az erőforrást elérhetjük.
A `@GET` annotáció arra utal, hogy a `getText()` metódus egy HTTP GET kérésre fog válaszolni.
A `@Produces` annotáció azt jelzi, hogy a `getText()` metódus szöveggel fog visszatérni.

A kliens HTTP kérése, amelyben megadja az erőforrást és a végrehajtandó metódust a következő formában adható meg.

```
GET http://localhost:8080/item
```

Válaszként pedig az ItemResource osztály `getText()` metódusa az `Ok!` szöveget adja.

2.4. A webszolgáltatás megvalósítása

A következő részekben ismertetem az alkalmazás megvalósításához felhasznált technológiákat, a rendszer alapját képező adatbázis szerkezetét és végül a szerver és a kliens oldalt megvalósító csomagokat.

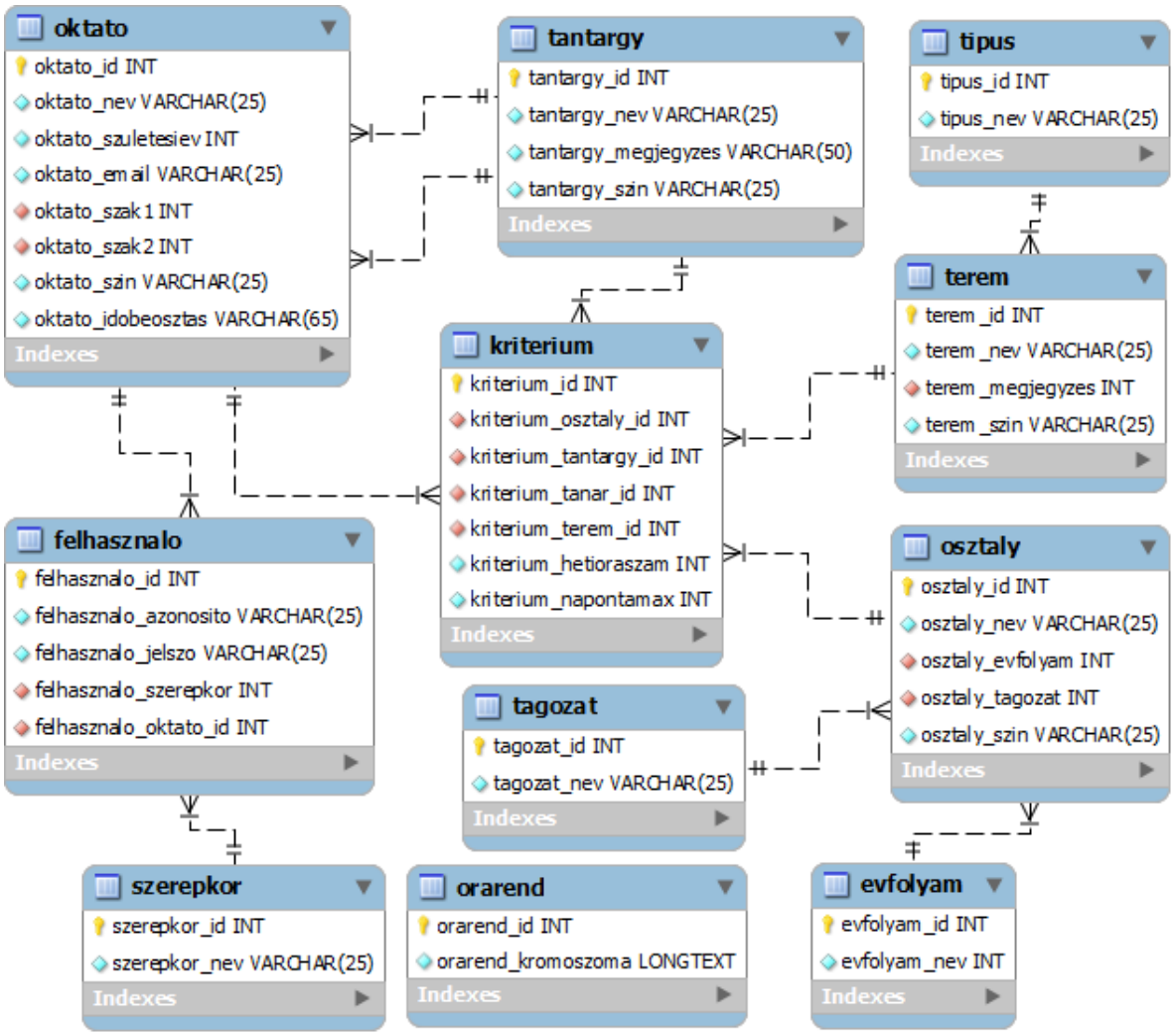
2.4.1. Alkalmazott technológiák

A dolgozatom megírása során NetBeans IDE 6.7.1, GlassFish Server v2.1 és MySQL 5.1 Community Servert alkalmaztam, amelyek a következő címeken érhetők el:

- http://netbeans.org/downloads/index.html
- http://dev.mysql.com/downloads/mysql/5.1.html
- https://glassfish.dev.java.net/public/downloadsindex.html

A GlassFish alkalmazásszerver teljes mértékben megfelel a Java EE 5 specifikációnak. A rendszer kereskedelmi verziójának neve: Sun Java System Application Server 9.x. Az eszköz nagyban építkezik a Sun alkalmazásszerver előző verzióira, illetve az Oracle TopLink nevű perzisztenciakezelő rendszerre. A Web-konténerét az Apache Tomcat-ből származtatták, amelyet a hatékonyabb működés céljából kibővítették a Grizzly nevű Java NIO-t használó komponenssel.
A MySql egy többfelhasználós, többszálú, SQL-alapú relációs adatbázis-kezelő szerver. A világon talán a legnépszerűbb nyílt forráskódú adatbázis kezelő rendszer.

2.4.2. Az adatbázis



3. ábra: Az adatbázis tábláinak kapcsolata

Az alkalmazás működése során felhasznált és kezelt adatok a `manhattandb` nevű adatbázisban tárolódnak. Az adatbázisban szereplő táblákat és a köztük levő kapcsolatot az előző ábra szemlélteti.

A következőkben az adatbázis egyes tábláinak szerepét ismertetem:

- Az `Oktato` tábla tartalmazza az egyes oktatók adatait: név, születési év, e-mail cím, az egyes oktatók által tartható órákat, – például egy oktató matematika-fizika szakos lehet – legfeljebb két elemet adhatunk meg. Továbbá eltárolásra kerül az egyes oktatókhoz rendelt szín és időbeosztás. Minden az órarendben szereplő erőforráshoz rendelünk egy színkódot, amellyel majd ábrázolásra kerül, ezáltal az órarend átláthatóbbá, könnyebben kereshetőbbé válik. Az időbeosztás mező a tanár időbeli ráérési követelményeit tárolja.

- Az `Osztaly` tábla tárolja az egyes osztályok esetén azok nevét, évfolyamát – például 6. B – és a tagozat tábla megfelelő sorának azonosítóját. Az osztályok esetén is eltárolásra kerül azok színkódja.

- A `Tantargy` tábla magába foglalja az egyes tárgyak nevét, illetve egy rövid leírást – például Matematika „Függvényvizsgálat, *s*zélsőérték számítás, integrálszámítás”– és a színkódot.

- A `Terem` tábla öleli fel az iskolában található termeket, nevüket – például 107 – és a típus tábla megfelelő sorának azonosítóját, valamint a színkódot.

- A `Felhasznalo` tábla tárolja a felhasználókat azonosító adatokat, úgymint azonosító, jelszó – például bk6pya, j9qd4y – és a szerepkör, oktató táblák megfelelő sorainak azonosítóját.

- A `Kriterium` tábla tartalmazza az órarend generáláshoz szükséges feltételeket, az elemi részeket, a halmazokat, amelyből az órarend előáll.

- A `Orarend` tábla tartalmazza a genetikus algoritmus által generált, a feltételeknek leginkább megfelelő órarendet.

Az eddig felsorolt táblák megfelelő jogosultsággal a grafikus felhasználói felületen keresztül bővíthetők, módosíthatók és törölhetők. A következőkben a segéd táblákat tekintjük át, amelyek a felhasználói felületen keresztül nem érhetők el, de a rendszer működése szempontjából nélkülözhetetlen meglétük. Az egyes táblák a következő adatokat tartalmazzák:

- Az `Evfolyam` tábla

id	nev
1	7
2	8
3	9
4	10
5	11
6	12
7	13

- A `Szerepkor` tábla

id	Nev
1	Adminisztrátor
2	Asszisztens
3	Tanár

- A `Tagozat` tábla

id	Nev
1	Hat évfolyamos
2	Hatosztályos
3	NYEK angol-német
4	Magyar-angol két tannyelvű
5	NYEK dráma-ének

- A `Tipus` tábla

Id	Nev
1	Osztályterem
2	Kémia terem
3	Fizika terem
4	Tornaterem
5	Pódium
6	Pince
7	K

2.4.3. A webszolgáltatás megvalósítása szerver oldalon

A szerver oldali szolgáltatást három csomag által valósítottam meg:

- model csomag
- service csomag
- converter csomag

2.4.3.1. A model csomag

A `manhattanS\src\java\model\` könyvtár által tartalmazott osztályok az alkalmazás modelljét ábrázolják. Az `Entity` osztályok az adatbázis tábláknak feleltethetők meg, egy-egy példány a megfelelő tábla egy sorát jelképezi. Mivel az egyes `Entity` osztályok felépítése nagyban hasonlít, ezért a példa kedvéért csak az `Oktato` osztály kódrészletét mutatom be.

```
@Entity
@Table(name = "oktato")

public class Oktato implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "oktato_id")
    private Integer oktatoId;
    @Basic(optional = false)
    @Column(name = "oktato_nev")
    private String oktatoNev;
    @Basic(optional = false)
    @Column(name = "oktato_szuletesiev")
    private int oktatoSzuletesiev;
    @Basic(optional = false)
    @Column(name = "oktato_email")
    private String oktatoEmail;
    @Basic(optional = false)
    @Column(name = "oktato_szin")
    private String oktatoSzin;
    @Basic(optional = false)
    @Column(name = "oktato_idobeosztas")
    private String oktatoIdobeosztas;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "felhasznaloOktatoId")
    private Collection<Felhasznalo> felhasznaloCollection;
    @JoinColumn(name = "oktato_szak1", referencedColumnName = "tantargy_id")
    @ManyToOne(optional = false)
    private Tantargy oktatoSzak1;
    @JoinColumn(name = "oktato_szak2", referencedColumnName = "tantargy_id")
    @ManyToOne(optional = false)
    private Tantargy oktatoSzak2;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "kriteriumTanarId")
    private Collection<Kriterium> kriteriumCollection;
```

2.4.3.2. A service csomag

A `manhattanS\src\java\service\` könyvtár által tartalmazott osztályok a RESTful webszolgáltatás erőforrásait szemléltetik. Két csoportra bonthatjuk az osztályokat: az egyik csoport tagjai adott erőforrást szemléltetnek, a másik csoport tagjai pedig az adott erőforrást tartalmazó listát. Például az `OktatoResource` osztály egy oktatót, mint erőforrást határoz meg, az `OktatoesResource` pedig az oktatóknak, mint erőforrásoknak a listáját szemlélteti.

Az erőforrásokat szemléltető osztályok mellett a csomag tartalmaz egy `PersistenceService` segéd osztályt, amely egy Java perzisztencia osztály.

Mint ahogy a **2.2.** alfejezetben említettem a RESTful webszolgáltatás esetében a kliens és a szerver közötti kommunikációban az erőforrások különböző reprezentációjának átvitele történik. Az erőforrások rendelkeznek egy egyedi URI-vel, amellyel megcímezhetők, illetve a kliens-szerver kommunikációja HTTP protokollon keresztül valósul meg. A HTTP metódusaival a CRUD (Create, Read, Update és Delete) műveletek is megvalósíthatóak, a leggyakrabban használt a GET és a POST, de alkalmazhatóak még a PUT és DELETE műveletek is.

Egy táblázat segítségével nézzük át a webszolgáltatás kommunikációjának főbb részeit.

Erőforrás	**URI Path**	**HTTP metódus**	**Leírás**
OktatoesResource	/oktatoes	GET	Oktatókat tartalmazó listát ad vissza
		POST	Egy új oktatóval bővít
OktatoResource	/oktatoes/{id}	GET	Az adott oktatót adja vissza
		PUT	Módosítja az adott oktatót
		DELETE	Törli az adott oktatót

A következőkben tekintsük át az erőforrás osztályok tartalmát. Mivel az egyes erőforrás osztályok felépítése nagyban hasonlít, ezért a példa kedvéért csak az `OktatoesResource` és `OktatoResource` osztályok kódrészletét taglalom.

Az `OktatoesResource` osztály metódusai közül kiemelném a `get()` és `post()` metódusokat:

- a `get()` metódus egy `OktatoesConverter` objektummal tér vissza, amely egy speciális konverter osztály. Részletesen az alábbiakban fogom kifejteni, jelenleg elég annyit tudni működéséről, hogy egy oktató objektumokból álló kollekciót állít elő.

```
@Path("/oktatoes/")
public class OktatoesResource {
    @Context
    protected UriInfo uriInfo;
    @Context
    protected ResourceContext resourceContext;

    @GET
    @Produces({"application/xml", "application/json"})
    public OktatoesConverter get(@QueryParam("start")
    @DefaultValue("0")
    int start, @QueryParam("max")
    @DefaultValue("10")
    int max, @QueryParam("expandLevel")
    @DefaultValue("1")
    int expandLevel, @QueryParam("query")
    @DefaultValue("SELECT e FROM Oktato e")
    String query) {
        PersistenceService persistenceSvc = PersistenceService.getInstance();
        try {
            persistenceSvc.beginTx();
            return new OktatoesConverter(getEntities(start, max, query),
                                         uriInfo.getAbsolutePath(), expandLevel);
        } finally {
            persistenceSvc.commitTx();
            persistenceSvc.close();
        }
    }

    protected Collection<Oktato> getEntities(int start, int max, String query) {
        EntityManager em = PersistenceService.getInstance().getEntityManager();
        return em.createQuery(query).setFirstResult(start).
                setMaxResults(max).getResultList();
    }
```

- a `post()` metódusa egy `Response` objektummal tér vissza. HTTP kérések esetén a választ ilyen objektumok tartalmazzák. A `post()` metódus argumentuma egy `OktatoConverter` objektum, ez az osztály rendelkezik egy `resolveEntity` metódussal, amely a bemeneti paramétert megfelelő módon adja vissza.

```
    @POST
    @Consumes({"application/xml", "application/json"})
    public Response post(OktatoConverter data) {
        PersistenceService persistenceSvc = PersistenceService.getInstance();
        try {
            persistenceSvc.beginTx();
            EntityManager em = persistenceSvc.getEntityManager();
            Oktato entity = data.resolveEntity(em);
            createEntity(data.resolveEntity(em));
            persistenceSvc.commitTx();
            return Response.created(uriInfo.getAbsolutePath().
                                    resolve(entity.getOktatoId() + "/")).build();
        } finally {
            persistenceSvc.close();
        }
    }
```

Az osztályban található annotációk a következő szereppel bírnak:

- `@Path` annotáció azon URI-t azonosítja – esetünkben ez az `/oktatoes` – amelyre, az `OktatoesResource` osztály a metódusait végrehajtja.
- `@GET` annotáció arra utal, hogy a `get()` metódus egy HTTP GET kérésre fog válaszolni.
- `@Produces` annotáció a válasz formátumát határozza meg, ebben az esetben XML vagy JSON formátum.
- `@QueryParam` annotáció a `get()` metódus paramétereit határozza meg.
- `@DefaultValue` annotáció a paraméterek alapértelmezett értékeit határozza meg.
- `@Post` annotáció arra utal, hogy a `post()` metódus egy HTTP POST kérésre fog válaszolni.
- `@Consumes` annotáció a klienstől érkező kérés elfogadható formátumát határozza meg, ebben az esetben XML vagy JSON formátum.

Az `OktatoResource` osztály által tartalmazott metódusok az oktató objektum példányokat törölhetik, módosíthatják, illetve visszaadhatják azokat.

```
public class OktatoResource {
    @Context
    protected UriInfo uriInfo;
    @Context
    protected ResourceContext resourceContext;
    protected String id;

    @GET
    @Produces({"application/xml", "application/json"})
    public OktatoConverter get(@QueryParam("expandLevel")
    @DefaultValue("1")
    int expandLevel) {
        PersistenceService persistenceSvc = PersistenceService.getInstance();
        try {
            persistenceSvc.beginTx();
            return new OktatoConverter(getEntity(),
                                       uriInfo.getAbsolutePath(),
                                       expandLevel);
        } finally {
            PersistenceService.getInstance().close();
        }
    }

    @PUT
    @Consumes({"application/xml", "application/json"})
    public void put(OktatoConverter data) {
        PersistenceService persistenceSvc = PersistenceService.getInstance();
        try {
            persistenceSvc.beginTx();
            EntityManager em = persistenceSvc.getEntityManager();
            updateEntity(getEntity(), data.resolveEntity(em));
            persistenceSvc.commitTx();
        } finally {
            persistenceSvc.close();
        }
    }
```

```
@DELETE
public void delete() {
    PersistenceService persistenceSvc = PersistenceService.getInstance();
    try {
        persistenceSvc.beginTx();
        deleteEntity(getEntity());
        persistenceSvc.commitTx();
    } finally {
        persistenceSvc.close();
    }
}
```

Az `OktatoesResource` osztályban található annotációk az `OktatoesResource` osztályban kiegészülnek egy `@DELETE` annotációval, amely azt jelzi, hogy a `delete()` metódus egy HTTP Delete kérésre fog válaszolni.

2.4.3.3. A converter csomag

A következőben tekintsük át a `manhattanS\src\java\converter\` könyvtár tartalmát. Ezen csomag a `manhattanS\src\java\service\` csomag által tartalmazott minden osztályhoz rendel egy converter osztályt. Mivel az egyes osztályok felépítése nagyban hasonlít, ezért a példa kedvéért csak az `OktatoesConverter` és `OktatoConverter` osztályok kódrészletét ismertetem.

Az `OktatoesConverter` osztály JAXB (Java Architecture for XML Binding) annotációkat alkalmaz. A JAXB alkalmazásával kollekcióba rendezhetjük XML vagy JSON formátumban a Java objektumokat.

```
@XmlRootElement(name = "oktatoes")
public class OktatoesConverter {
    private Collection<Oktato> entities;
    private Collection<OktatoConverter> items;
    private URI uri;
    private int expandLevel;

    @XmlElement
    public Collection<OktatoConverter> getOktato() {
        if (items == null) {
            items = new ArrayList<OktatoConverter>();
        }
        if (entities != null) {
            items.clear();
            for (Oktato entity : entities) {
                items.add(new OktatoConverter(entity, uri, expandLevel, true));
            }
        }
        return items;
    }

    @XmlAttribute
    public URI getUri() {
        return uri;
    }
```

A kódrészletben található annotációk a következő szereppel bírnak:

- `@XmlRootElement` annotáció jelzi, hogy az `oktatoes` egy felső szintű XML csomópont vagy JSON objektum.
- `@XmlElement` annotáció által jelezhetjük, hogy a kollekció egy XML vagy JSON elem.
- `@XmlAttribute` annotáció alkalmazásával megfeleltethetjük az `uri` JavaBeans tulajdonságot, XML attribútumnak vagy JSON tulajdonságnak.

Az `OktatoConverter` osztály konstruktorokat, valamint getter és setter metódusokat tartalmaz. Nézzük meg a kódrészletét!

```
@XmlRootElement(name = "oktato")
public class OktatoConverter {
    private Oktato entity;
    private URI uri;
    private int expandLevel;

    public OktatoConverter(Oktato entity,
                           URI uri, int expandLevel,
                           boolean isUriExtendable) {
        this.entity = entity;
        this.uri = (isUriExtendable) ? UriBuilder.fromUri(uri).
                                path(entity.getOktatoId() + "/").build() : uri;
        this.expandLevel = expandLevel;
        getFelhasznaloCollection();
        getOktatoSzak1();
        getOktatoSzak2();
        getKriteriumCollection();
    }

    @XmlElement
    public Integer getOktatoId() {
        return (expandLevel > 0) ? entity.getOktatoId() : null;
    }

    public void setOktatoId(Integer value) {
        entity.setOktatoId(value);
    }
```

A `manhattanS\src\java\converter\` könyvtár a konverter osztályokon kívül tartalmaz egy `UriResolver` osztályt is, amelynek szerepe a converter osztályokban az URI kezelése.

2.4.4. A webszolgáltatás megvalósítása JavaFx kliens oldalon

A kliens oldalt három csomag által valósítottam meg, a csomagok JavaFx osztályokat tartalmaznak:

- model csomag
- client csomag
- parser csomag

2.4.4.1. A model csomag

A `manhattan\src\manhattan\client\model\` könyvtárban szereplő osztályok a kliens oldalon ábrázolják az erőforrásokat. Mivel az egyes osztályok felépítése nagyban hasonlít, ezért a példa kedvéért csak az `OktatoModel` osztály kódrészletét mutatom be.

```
public class OktatoModel {
    private Integer oktatoId;
    private String oktatoNev;
    private int oktatoSzuletesiev;
    private String oktatoEmail;
    private String oktatoSzin;
    private String oktatoIdobeosztas;
    private Collection<FelhasznaloModel> felhasznaloCollection;
    private TantargyModel oktatoSzak1;
    private TantargyModel oktatoSzak2;
    private Collection<KriteriumModel> kriteriumCollection;

    public static List<OktatoModel> oktatok = new ArrayList<OktatoModel>();

    public OktatoModel(Integer oktatoId, String oktatoNev, int oktatoSzuletesiev,
            String oktatoEmail, String oktatoSzin, String oktatoIdobeosztas) {
        this.oktatoId = oktatoId;
        this.oktatoNev = oktatoNev;
        this.oktatoSzuletesiev = oktatoSzuletesiev;
        this.oktatoEmail = oktatoEmail;
        this.oktatoSzin = oktatoSzin;
        this.oktatoIdobeosztas = oktatoIdobeosztas;
    }

    public Integer getOktatoId() {
        return oktatoId;
    }

    public void setOktatoId(Integer oktatoId) {
        this.oktatoId = oktatoId;
    }
```

2.4.4.2. A client csomag

A `manhattan\src\manhattan\client\` könyvtárban szereplő osztályok valósítják meg a kliens-szerver közötti kommunikációt, ezek alapján feladatukat két részre bonthatjuk:

- a webszolgáltatás irányába megfogalmazzák a kérést, az adott erőforrás URI és HTTP művelet megadásával.
- a webszolgáltatás irányából érkező XML formátumú választ megfelelő Parser típusú osztály alkalmazásával feldolgozzák. Részletesen a Parser típusú osztályokat az alábbiakban taglalom, jelenleg elég annyit tudni működéséről, hogy az érkező válaszokból az erőforrások adattagjait kinyerik, és az adott erőforrásnak megfelelő model osztálynak adják át.

Mivel az egyes osztályok felépítése nagyban hasonlít, ezért a példa kedvéért csak az `OktatoClient` osztály metódusait mutatom be.
Az osztály metódusai által megvalósított műveletek a következők

- valamennyi oktató erőforrás lekérdezése

```
public function loadMetadata() {
    println("Call Restful Web Service... (Oktato)");

    var httpRequestError: Boolean = false;
    var request: HttpRequest = HttpRequest {
        location: 'http://localhost:8080/manhattanS/resources/'
                                        'oktatoes/?start=0&max=0&expandLevel=2'
        method: HttpRequest.GET

        onException: function(exception: Exception) {
            exception.printStackTrace();
            alert("Error", "{exception}");
            httpRequestError = true;
        }
        onResponseCode: function(responseCode:Integer) {
            if (responseCode == 200) {
                println('(Oktato) load OK, response:'
                                    '{responseCode}{request.responseMessage}');
            }
        }
```

```
            onInput: function(input: java.io.InputStream) {
                try {
                    var parser = OktatoParser {
                        onDone: function( data:OktatoModel[] ) {
                           if(not httpRequestError) {
                            OktatoModel.oktatok.clear();
                             for(oktato in data)
                              OktatoModel.oktatok.add(oktato);
                              delete OktatoTableHelper.oktatok;
                              OktatoTableHelper.oktatok = OktatoModel.getOktatokA();
                            }
                        }
                    };
                    parser.parse(input);
                } finally {
                    input.close();
                }
            }
        }
        request.start();
    }
```

A `loadMetadata()` metódus a `javafx.io.http.HttpRequest` osztályt alkalmazza a HTTP GET kérés küldésére a RESTful webszolgáltatás irányába. A `HttpRequest` osztály rendelkezik egy `start()` metódussal, a kérelem ezzel küldhető el.

A `HttpRequest` objektum esetében két változó értékét kell meghatározni:

- `location` értéke `http://localhost:8080/manhattanS/resources/oktatoes/` azaz az oktató erőforrások URI-je. A kódrészletben szereplő URI végén található argumentumok a következő jelentéssel bírnak: a lekérdezés a 0-dik elemtől kezdődik (`start=0`), nincsen elemszámra vonatkozó megkötés (`max=0`) és a beágyazási mélység kettő (`expandLevel=2`).
- `method` értéke `HttpRequest.GET` azaz egy HTTP GET kérés valósul meg.

A `location` és a `method` változókon kívül a `HttpRequest` objektum függvényekkel is rendelkezik:

- `onException` függvény a kommunikációs folyamat során fellépő hibákról ad értesítést.
- `onResponseCode` függvény a webszolgáltatás által küldött HTTP státuszkódot értékeli ki. Bővebb leírás az irodalomjegyzék [14] pontjában található.
- `onInput` függvény a webszolgáltatás válasz üzenetét az `OktatoParser` osztály alkalmazásával dolgozza fel.

- oktató erőforrás törlése

```
public function deleteMetadata(id: Integer) {
    println("Call Restful Web Service... (Oktato)");

    var httpRequestError: Boolean = false;
    var request: HttpRequest = HttpRequest {
        location: "http://localhost:8080/manhattanS/resources/oktatoes/{id}"
        method: HttpRequest.DELETE

        onException: function(exception: Exception) {
            exception.printStackTrace();
            alert("Error", "{exception}");
            httpRequestError = true;
        }
        onResponseCode: function(responseCode:Integer) {
            if (responseCode == 204) {
                println('(Oktato) delete OK, response:'
                                    '{responseCode}{request.responseMessage}');
                 TantargyClient.loadMetadata();

                 loadMetadata();
            }
        }
    }
    request.start();
}
```

A `deleteMetadata()` metódus változásai az előző esethez képest a következők:

- változott az URI, az `{id}-vel` az erőforrások közül az `id` –edik kerül kijelölésre. Az `id` értéket a `deleteMetadata()` metódus paraméterként kapja meg.
- változott a `method` értéke is `HttpRequest.DELETE -re` azaz egy HTTP DELETE kérés valósul meg.

- oktató erőforrás módosítása

```
public function editMetadata(id: Integer, nev: String, email: String,
                             szuletesiev: Integer, szin: String,
                             idobeosztas: String, szak1: Integer, szak2: Integer){
    println("Call Restful Web Service... (Oktato)");

    var httpRequestError: Boolean = false;
    var request: HttpRequest = HttpRequest {
        location: "http://localhost:8080/manhattanS/resources/oktatoes/{id}"

        method: HttpRequest.PUT

        onException: function(exception: Exception) {
            exception.printStackTrace();
            alert("Error", "{exception}");
            httpRequestError = true;
        }
```

```
            onResponseCode: function(responseCode:Integer) {
                if (responseCode == 204) {
                    println('(Oktato) edit OK, response:'
                                        '{responseCode}{request.responseMessage}');
                     TantargyClient.loadMetadata();
                     loadMetadata();
                }
            }
            onOutput: function(output: java.io.OutputStream) {
                try {...} finally {
                    output.close();
                }
            }
        }
        request.start();
    }
```

Az `editMetadata()` metódus változásai a következők:

- változott a `method` értéke `HttpRequest.PUT -ra` azaz egy HTTP PUT kérés valósul meg.
- az `onInput` függvény helyett `onOutput` függvényt alkalmazunk, amely egy XML üzenetben határozza meg a módosítandó erőforrást, és az új adattagokat.

- új oktató erőforrás felvétele

```
    public function newMetadata(nev: String, email: String, szuletesiev: Integer,
                                szin: String, idobeosztas: String,
                                szak1: Integer, szak2: Integer) {
    println("Call Restful Web Service... (Oktato)");

        var httpRequestError: Boolean = false;
        var request: HttpRequest = HttpRequest {
            location: "http://localhost:8080/manhattanS/resources/oktatoes/"

            method: HttpRequest.POST

            headers: [
            HttpHeader {
                name: HttpHeader.CONTENT_TYPE;
                value: "application/xml";
                }
            ];

            onException: function(exception: Exception) {
                exception.printStackTrace();
                alert("Error", "{exception}");
                httpRequestError = true;
            }
```

```
        onResponseCode: function(responseCode:Integer) {
            if (responseCode == 201) {
                println('(Oktato) new OK, response:'
                                    '{responseCode}{request.responseMessage}');
                   TantargyClient.loadMetadata();
                   loadMetadata();
            }
        }
        onOutput: function(output: java.io.OutputStream) {
            try {...} finally {
                output.close();
            }
        }
    }
    request.start();
}
```

Az `newMetadata()` metódus változásai a következők:

- változott az URI, ebben az esetben nem tartalmaz argumentumot. Új erőforrás felvétele a meglévő erőforrások mögé történik.
- változott a `method` értéke `HttpRequest.POST -ra` azaz egy HTTP POST kérés valósul meg.
- az `onOutput` függvény XML üzenetben határozza meg az új erőforrást. A XML üzenet alkalmazását előzőleg jeleztük a `HttpHeader` objektum `value` változójában "application/xml" érték megadásával.

2.4.4.3. A parser csomag

A `manhattan\src\manhattan\client\parser\` könyvtárban szereplő osztályok a webszolgáltatás válaszaiból – esetünkben XML formátum – nyerik ki az erőforrások adattagjait és adják tovább a megfelelő model osztályoknak.
Mivel az egyes osztályok felépítése nagyban hasonlít, ezért a példa kedvéért csak az `OktatoParser` osztály kódrészletét mutatom be.

```
public class OktatoParser {

 var oktato: OktatoModel;
 var oktatok: OktatoModel[];

 public var onDone: function(data : OktatoModel[]) = null;

    def parseEventCallback = function(event: Event) {
        if (event.type == PullParser.START_ELEMENT) {
            processStartEvent(event)
        } else if (event.type == PullParser.END_ELEMENT) {
            processEndEvent(event)
        }else if (event.type == PullParser.END_DOCUMENT) {
            if (onDone != null) {
                onDone(oktatok);
            }
        }
    }
```

```
    function processStartEvent(event: Event) {
        if(event.qname.name == "oktato" and event.level == 1) {
            oktato = OktatoModel {};
        }
    }

    var oktatoSzak = 2;

    function processEndEvent(event: Event) {
        if(event.qname.name == "oktato" and event.level == 1) {
            insert oktato into oktatok;
        } else if(event.qname.name == "oktatoId" and event.level == 2) {
            oktato.setOktatoId(java.lang.Integer.parseInt(event.text));
        }  else if(event.qname.name == "oktatoNev" and event.level == 2) {
            oktato.setOktatoNev(event.text);
        } else if(event.qname.name == "oktatoSzuletesiev" and event.level == 2){
            oktato.setOktatoSzuletesiev(java.lang.Integer.parseInt(event.text));
        } else if(event.qname.name == "oktatoEmail" and event.level == 2) {
            oktato.setOktatoEmail(event.text);
        } else if(event.qname.name == "oktatoIdobeosztas" and event.level == 2){
            oktato.setOktatoIdobeosztas(event.text);
        } else if(event.qname.name == "oktatoSzin" and event.level == 2) {
            oktato.setOktatoSzin(event.text);
        } else if(event.qname.name == "tantargyId" and event.level == 3) {
            if(oktatoSzak == 1 and oktato.getOktatoSzak1() == null){
                oktato.setOktatoSzak1(
                new TantargyModel(java.lang.Integer.parseInt(event.text)));
                oktatoSzak = 3 - oktatoSzak}
            if(oktatoSzak == 2 and oktato.getOktatoSzak2() == null){
                oktato.setOktatoSzak2(
                new TantargyModel(java.lang.Integer.parseInt(event.text)));
                oktatoSzak = 3 - oktatoSzak}
        } else if(event.qname.name == "felhasznaloId" and event.level == 4) {
            oktato.addFelhasznaloCollection(
            new FelhasznaloModel(java.lang.Integer.parseInt(event.text)));
        } else if(event.qname.name == "kriteriumId" and event.level == 4) {
            oktato.addKriteriumCollection(
            new KriteriumModel(java.lang.Integer.parseInt(event.text)));
        }
    }
    public function parse(input: InputStream) {

        def parser = PullParser {
            input: input
            onEvent: parseEventCallback
        }
        parser.parse();
    }

}
```

Az osztály `parse` függvénye a `javafx.data.PullParser` osztály `parse()` metódusát hívja meg az input értelmezéséhez. A `PullParser` objektum `onEvent` metódusát az osztály `parseEventCallback` metódusa valósítja meg, amely értelmezi és feldolgozza XML üzenetet.

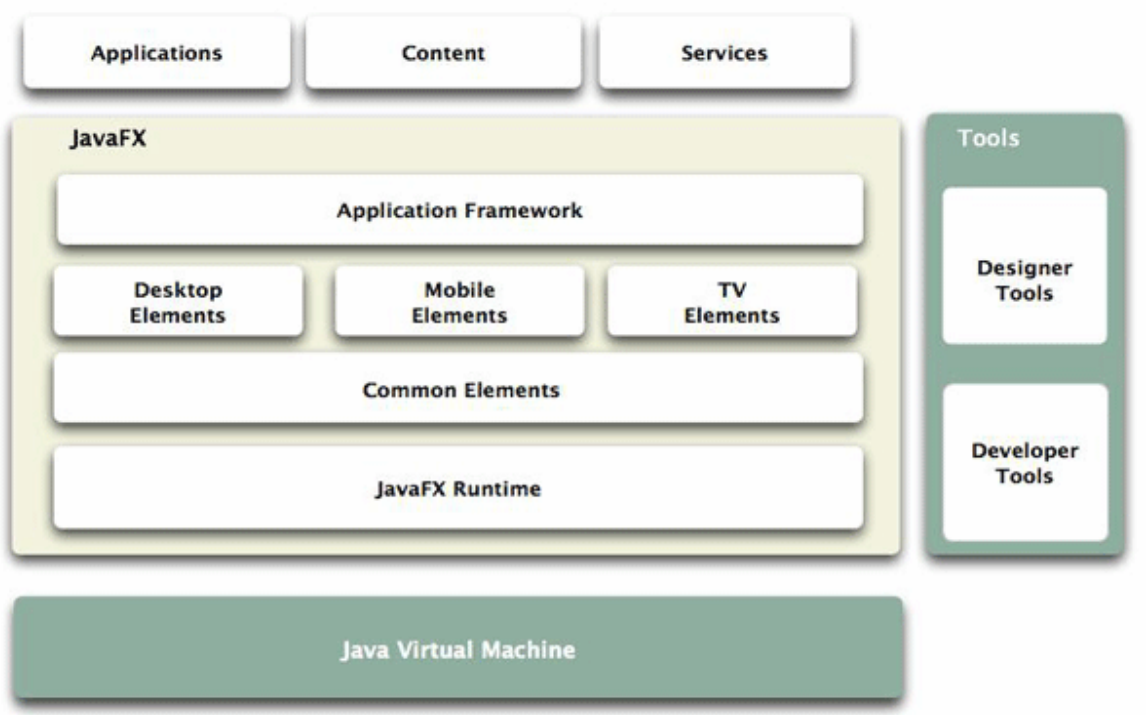
3. Felhasználói felületek és funkciók

3.1. JavaFx technológiáról röviden

A Sun állítása szerint Java kód futtatására a világon mintegy 800 millió PC és 2,2 milliárd mobiltelefon képes, ezáltal a Java a legszélesebb körben alkalmazott programozási környezetté vált.

"A Java-technológia a legerőteljesebb, legrugalmasabban méretezhető, egyúttal pedig a legbiztonságosabb fejlesztési platformmá vált a vállalati és mobil alkalmazások széles köre számára" – fogalmazta meg Rich Green.

Mindezek ellenére mivel a platform nélkülözi a fejlett audiovizuális képességeket kimaradt a webkettes térhódításából, mely során a web egyre interaktívabbá és vizuálisabbá vált. A vállalat ezen változtatni akarva megalkotta a JavaFx-et, mely által a Java sokkal kifejezőbbé vált.



4. ábra: A JavaFX nyelv szerkezete [16]

A JavaFx alkalmazások elvileg változtatás nélkül futtathatók a különböző platformokon, a programok kódbázisának nagy része megegyezik a PC-s változattal. A Sun beindította a JavaFX Production Suite-ot, amellyel a fejlesztők sokkal szélesebb körét kívánja elérni,

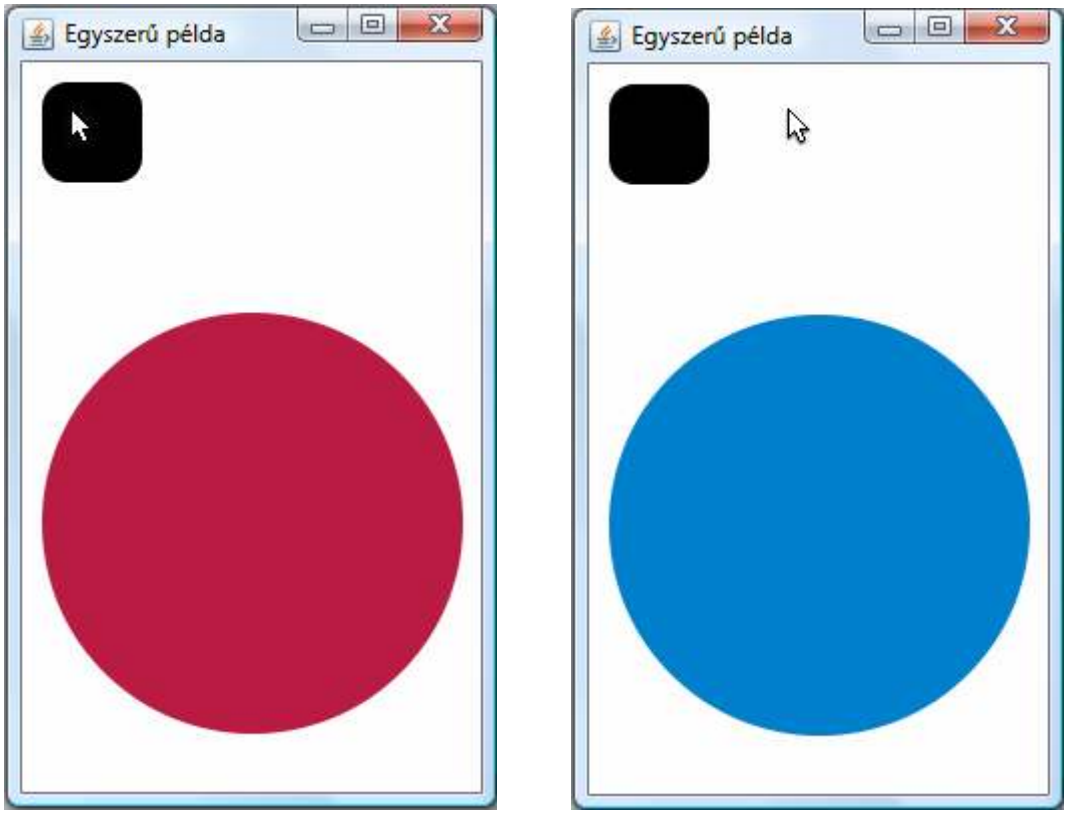
ugyanis lehetőséget ad professzionális grafikus szoftverek által előállított állományok JavaFx-ben való felhasználására.

A felhasználók szempontjából talán a legérdekesebb megoldás az a drag&drop telepítési módszer, amikor egy applettet futás közben kiemelünk a böngészőből és az asztalra húzzuk, majd a böngészőt bezárva az alkalmazás fut tovább, azaz az alkalmazások nem kötődnek a böngészőhöz, képesek offline működésre.
A Java platform támogatását élvezve a JavaFx képes komplex számítási feladatok és kiterjedt rendszer-integráció megvalósítására.
Mindezek alapján elmondhatjuk, hogy a JavaFX Script egy megfelelő eszköz a webes dizájnerek, szkript- és alkalmazásfejlesztők kezében arra, hogy képesek legyenek gyorsan létrehozni, illetve megjelentetni RIA-alkalmazásokat (Rich Internet Applications) számítógépeken, mobil készülékeken és más fogyasztói eszközökön.

3.2. A JavaFX szkriptnyelv

A Sun 2005-ben az F3 (Form Follows Function) nyelvre alapozva kezdte meg a JavaFX szkriptnyelv kifejlesztését, amely segítségével egyszerűen és gyorsan fejleszthetünk interaktív grafikus alkalmazásokat. A nyelvben a deklaratív és objektum orientált szemlélet keveredik, ez alapozza meg az alkalmazások nagyfokú rugalmasságát és robosztusságát.

Egy példán keresztül ismertetem a nyelv néhány elemét:

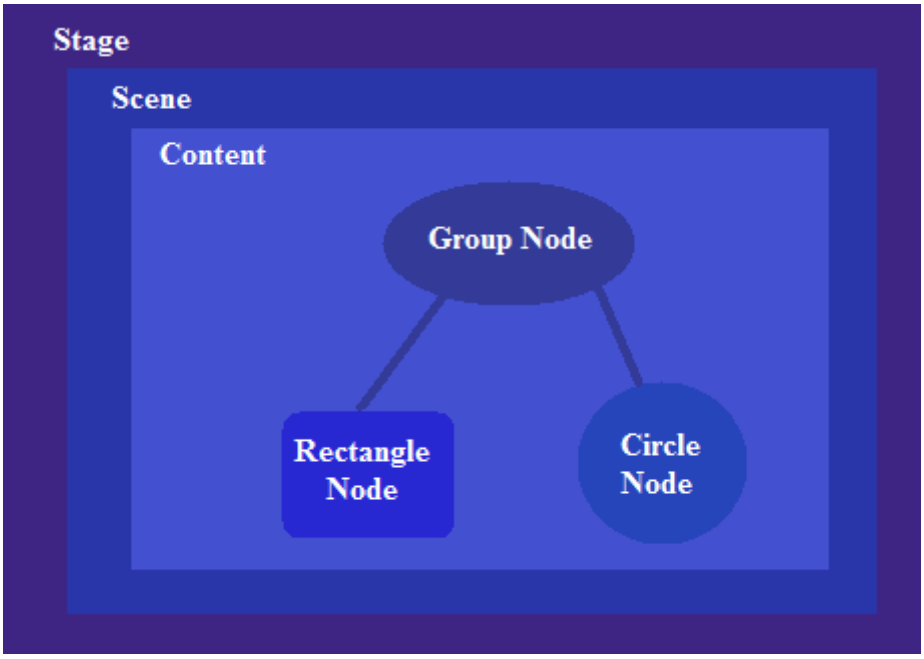


Az alkalmazás működése nagyon egyszerű:

- ha a négyzetben van az egérkurzor, akkor piros a kör.
- ha a négyzeten kívül van az egérkurzor, akkor kék a kör.
- ha a körbe kattintunk, akkor kilépünk az alkalmazásból.

```
package javafxpelda;

import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.shape.Rectangle;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.input.MouseEvent;


def w = 245;
def h = 400;

var color_r = 0;
var color_g = 0;
var color_b = 0;

Stage {
    title: "Egyszerű példa"
    width: w
    height: h
    scene: Scene {
        content: Group {
         content: [
           Rectangle {
            x: 10, y: 10
            width: 50, height: 50
            arcWidth: 25 arcHeight: 25
            fill: Color.rgb(0,0,0)
            onMouseEntered: function( e: MouseEvent ):Void {
                color_r = 186; color_g = 27; color_b = 67;
            }
            onMouseExited: function( e: MouseEvent ):Void {
                color_r = 0; color_g = 128; color_b = 204;
            }
           },
           Circle {
            centerX: 115, centerY: 230
            radius: 105
            fill: bind Color.rgb(color_r, color_g, color_b);
            onMousePressed: function( e: MouseEvent ):Void {
                javafx.lang.FX.exit();
            }
          }
         ]
        }
    }
}
```

Mint látható egy egyszerű szerkezet valósítja meg a felhasználói felületet. A felület a `Stage`, `Scene`, `Rectangle`, és `Circle` objektumokból épül fel. Ezek viszonyát az alábbi ábra mutatja be:



5. ábra: Az objektumok kapcsolata

A legkülső szinten a `Stage` szerepel, feladata az ablakot megjeleníteni, változói (például: `title`, `width`, `height`) által megfelelően formázható. A megjelenítést szabályozó változókon kívül tartalmaz egy `scene` változót, amely egy `Scene` objektum példányt határoz meg. A `Scene` objektum a grafikai tartalmak számára megjelenítési felületet biztosít, ezeket az objektumokat egy hierarchikus struktúrába szervezi. A struktúra elemeit nodoknak hívjuk, egy node lehet: szöveg, kép valamilyen média formátum, vagy mint esetünkben kör vagy négyzet.

A nyelv egy fontos jellemzője az adatok összekötését megvalósító `bind`, amely egy azonnali direkt kapcsolatot hoz létre két változó vagy változó és függvény vagy kifejezés kimenet közt. Azaz, ha változás következik be, akkor a változó értéke is megfelelően módosul. A példa kódban ilyen kapcsolat figyelhető meg az egérkurzor pozíciója és a kör színe közt.

A felsoroltak mellett a program kódban szerepel változó, illetve függvény deklarálás is. A nyelv referenciája az irodalomjegyzék [19] pontjában található címen érhető el, illetve az 1.2-es API a [20] pontban található címen.

Ezek alapján elmondhatjuk, hogy a JavaFX szkriptnyelvvel egyszerűen és könnyen valósíthatunk meg egészen kifejező felhasználói felületeket.

3.3. Az alkalmazás működése

Az alkalmazás segítségével három egymással szorosan együttműködő felületen keresztül az alábbi tevékenységek elvégzésére van lehetőség:

- Az órarend generálásához szükséges erőforrások kezelése.
- Lágy, illetve kemény követelmények meghatározása.
- A generált órarendben való keresés különböző szempontok alapján.

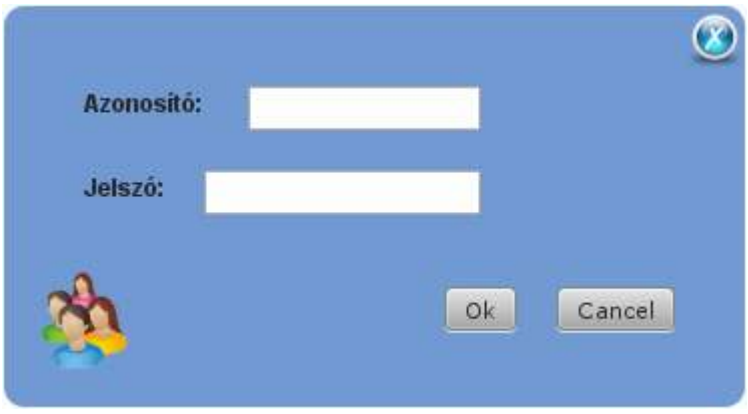
3.3.1. Felhasználói csoportok

A felhasználókat az alkalmazás működése szempontjából három csoportba sorolhatjuk. Minden egyes felhasználó egyedi azonosítóval rendelkezik, amellyel az adott felületre bejelentkezhet. Az egyes jogosultságokat az adminisztrátor rendeli a felhasználókhoz. A rendszer megengedi, hogy egy felhasználó több jogkörrel is rendelkezzen. Ilyen eset fordulhat elő, ha az iskola informatika tanárja a rendszer adminisztrátori feladatait is ellátja és oktat is egyben.

A következő táblázat az egyes csoportokat és azok funkcióit mutatja be:

Felhasználói csoport	Funkciók
Adminisztrátor	Az órarendben megjelenő erőforrások kezelése: - létrehozás - módosítás - törlés
Asszisztens	A kemény illetve lágy követelmények létrehozása
Tanár	Órarendben való keresés Ráérési táblázatának létrehozása

3.3.2. Bejelentkező ablak



Az alkalmazás elindítását követően az előző ablak jelenik meg, amely a bejelentkezéshez szükséges adatok megadására és ellenőrzésére szolgál.

Működése a következő: minden felhasználó rendelkezik az adott jogosultságához tartozó azonosítóval, illetve az ezt megerősítő jelszóval, ezeket a megfelelő mezőkbe kell beírni. Mivel a jelszó bizalmas információ, ezért a jelszó mező kitöltésekor úgynevezett helyettesítő karakterek fognak megjelenni.
A mezők kitöltése során tartsuk szem előtt, hogy a rendszer a kis- és a nagybetűket megkülönbözteti, ezt figyelmen kívül hagyva a belépési kísérlet hibával zárul.

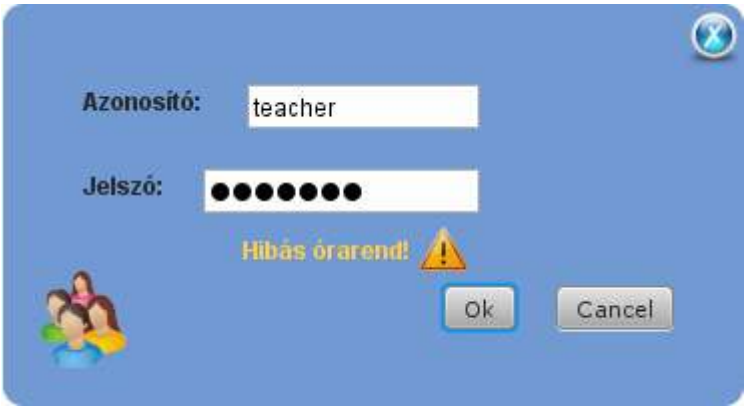
A telepítést követően az egyes felhasználói csoportokhoz tartozó ablakok alapértelmezett azonosító-jelszó párosa a következő:

Azonosító	**Jelszó**
admin	admin
assistant	assistant
teacher	teacher

Miután a megfelelő mezőket kitöltöttük az egérrel az 

gombra kattintva léphetünk tovább a megfelelő ablakba. Ha az adatok nem megfelelőek, akkor a felhasználót a rendszer egy hibaüzenettel tájékoztatja, a következő formában:

Ezek mellett a tanár jogkörrel rendelkező felhasználók csak akkor léphetnek be a felületükre, ha a genetikus algoritmus már legenerált egy órarend példányt, ellenkező esetben ezt is jelzi a rendszer:



3.3.3. Felhasználói felületek

Az alábbiakban a sikeres bejelentkezést követően megjelenő, a felhasználó jogosultságának megfelelő ablakok szerkezetét és azok funkcionális működését ismertetem.

A felhasználói felületek szerkezeti felépítésükben az alábbi közös elemmel rendelkeznek:

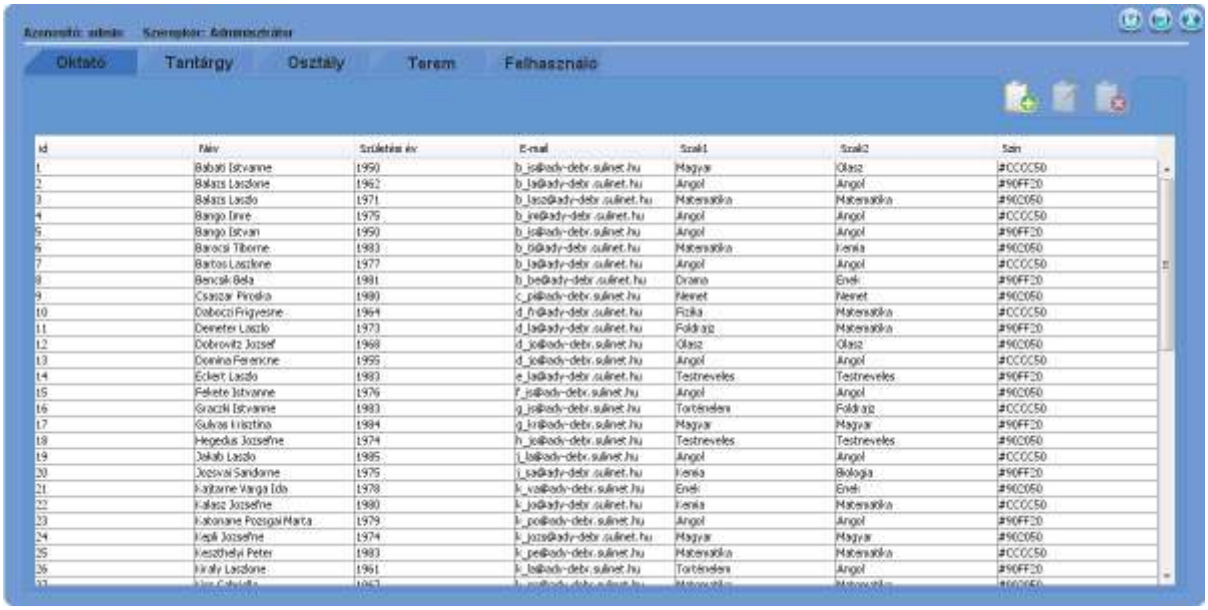
Megjeleníti a bejelentkezett felhasználó azonosítóját, illetve a felhasználó jogosultságához tartozó szerepkört is.

Jobb oldalon az ablakkezelő ikonok szerepelnek, amelyek a következő funkciókkal rendelkeznek:

- az  ikon megnyomására a felhasználó kijelentkezik a rendszerből.
- az  ikon megnyomására a felhasználói ablak a háttérbe kerül.
- az  ikon megnyomására az alkalmazás bezárul.

3.3.3.1. Az adminisztrátor ablak

Ha a felhasználó adminisztrátori jogkörrel rendelkezik, valamint a bejelentkezési ablak mezőit megfelelően töltötte ki, akkor következő ablak jelenik meg:

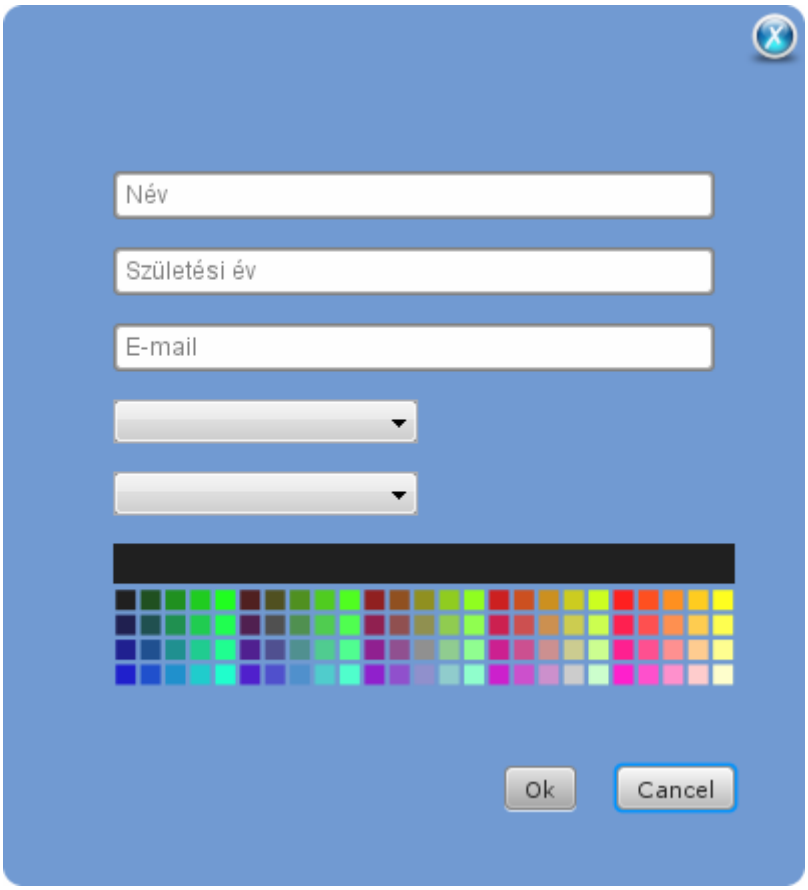


A megjelenő felületen a bejelentkezett felhasználó az órarend generáláshoz, illetve a rendszer működéséhez szükséges erőforrásokat adminisztrálhatja. Az ablak tartalmaz egy választó menü részt, amely segítségével a felhasználó az egyes erőforrások között navigálhat.

Az oktató fülre kattintva a táblázatban kilistázódnak az adatbázisban tárolt oktató erőforrások részletes adatai, úgy mint:

- id
- Név
- Születési év
- E-mail
- Szak1
- Szak2
- Szín

Abban az esetben, ha az adatbázisba új erőforrást akarunk felvenni, akkor ezt az  ikonra kattintva tehetjük meg. Ennek hatásra az adminisztrátori felület inaktívvá válik és megjelenik az új erőforrás bevitelét szolgáló ablak.



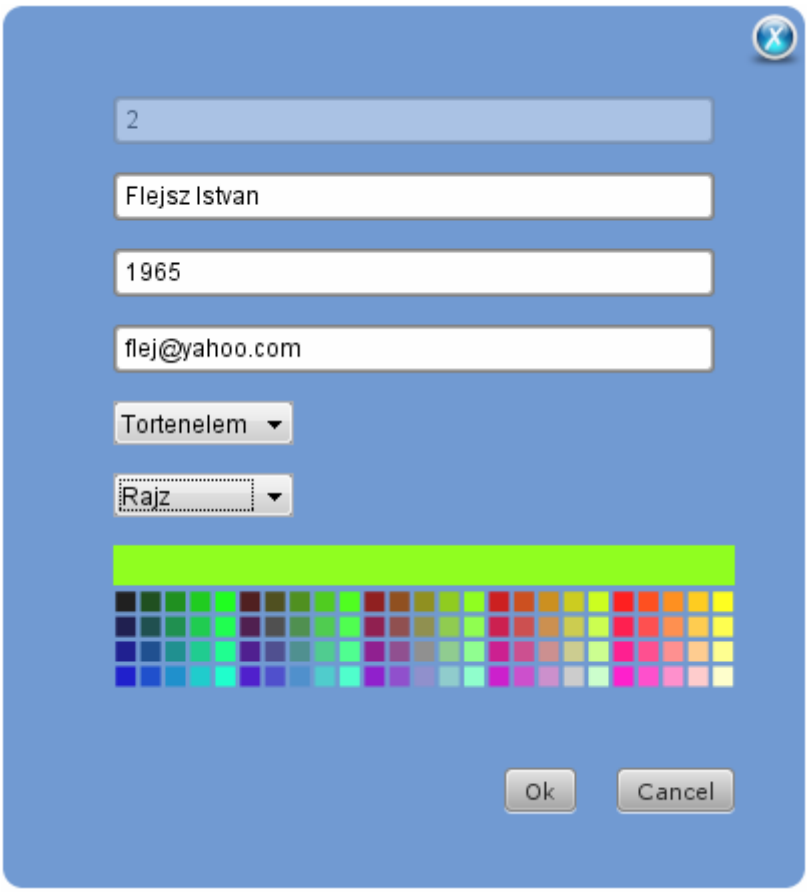
Ezen a felületen a felhasználó, a mezőket kitöltve egy új erőforrást vehet fel a rendszerbe. A mezőkbe a felhasználó a mezők címzése alapján viheti be a megfelelő adatokat. A beviteli mezőkön kívül az ablak tartalmaz lenyíló választó elemeket is, amelyek segítségével a rendszerben található megfelelő erőforrások közt választhatunk. Mivel a végleges órarend áttekintését, a benne való keresést nagyban megkönnyíti, ha az egyes erőforrásokat vizuálisan megkülönböztetjük, ezért lehetőség van minden erőforráshoz egy szín hozzárendelésére, amellyel az órarendben szerepelni fog.

A bevitt értékek elmentéséhez az 

gombra kattintunk, vagy ha mégsem akarunk új elemet felvenni akkor a 

gombra vagy  ikonra kattintva térhetünk vissza az adminisztrátori felületre.

Abban az esetben, ha az adatbázisban szereplő erőforrás adatait szeretnénk módosítani, akkor ezt az  ikonra kattintva tehetjük meg. Kezdetben az ikon inaktív, aktívvá válik, ha a táblázatból kiválasztottunk egy erőforrást szemléltető sort. Az ikonra kattintva az adminisztrátori felület inaktívvá válik és megjelenik az erőforrás módosítására szolgáló ablak.



A megjelenő felületen az adott erőforrás adataival kitöltött elemek szerepelnek, a kívánt módosítások elmentéséhez az Ok gombra kattintunk, vagy ha mégsem akarunk módosítani akkor a Cancel gombra vagy ikonra kattintva térhetünk vissza az adminisztrátori felületre.

Abban az esetben, ha az adatbázisba szereplő erőforrást törölni szeretnénk, akkor ezt az ikonra kattintva tehetjük meg. Kezdetben az ikon inaktív, aktívvá válik, ha a táblázatból kiválasztottunk egy erőforrást szemléltető sort. Abban az esetben, ha az erőforrás kapcsolatban áll egy másik erőforrással, akkor a rendszer megerősítést kér az illető erőforrásra vonatkozóan, a következő ablak formájában.

Minden az adatbázissal kapcsolatos módosítást a webszolgáltatás azonnal feldolgoz, módosítja az adatbázist és a művelet eredménye a felhasználói felületen az erőforrásokat listázó táblázatban jelenik meg.

A minden erőforrás típus esetében a fentebb említett „új”, „módosít”, „töröl” művelet hasonló módon működik.

A tantárgy fülre kattintva a táblázatban kilistázódnak az adatbázisban tárolt tantárgy erőforrások részletes adatai, úgy mint:

- id
- Név
- Megjegyzés
- Szín

Az osztály fülre kattintva a táblázatban kilistázódnak az adatbázisban tárolt osztály erőforrások részletes adatai, úgy mint:

- id
- Név
- Évfolyam
- Tagozat
- Szín

A terem fülre kattintva a táblázatban kilistázódnak az adatbázisban tárolt terem erőforrások részletes adatai, úgy mint:

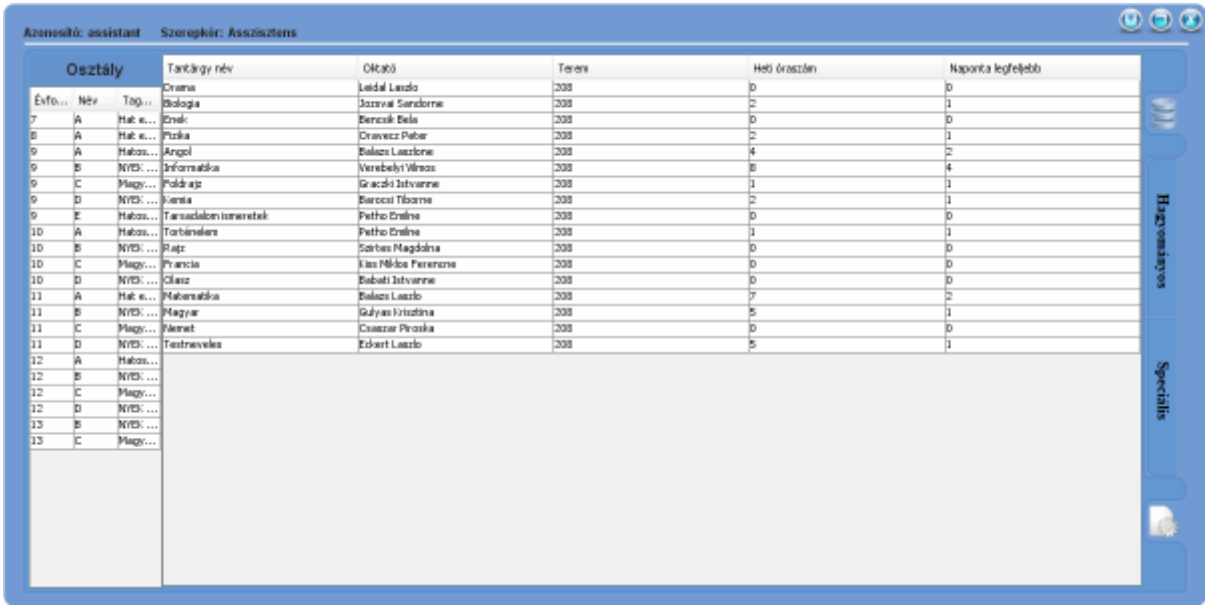
- id
- Név
- Megjegyzés
- Szín

A felhasználó fülre kattintva a táblázatban kilistázódnak az adatbázisban tárolt felhasználó erőforrások részletes adatai, úgy mint:

- id
- Azonosító
- Jelszó
- Szerepkör
- Felhasználó

3.3.3.2. Az asszisztens ablak

Ha a felhasználó asszisztens jogkörrel rendelkezik, valamint a bejelentkezési ablak mezőit megfelelően töltötte ki, akkor következő ablak jelenik meg:



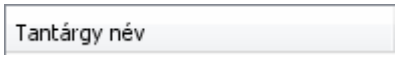
A megjelenő felületen a bejelentkezett felhasználó az órarend generáláshoz szükséges kemény illetve lágy követelményeket határozhatja meg. A követelményeket bővebben az **1.5.5.** fejezetben fejtettem ki.

A követelményeknek két csoportját különböztethetjük meg. Vannak azon követelmények, amelyek csak egy osztályt érintenek, és vannak azon követelmények, amelyek osztályok, tanárok, tantermek egy halmazát érintik. Az egyes csoportok a jobb oldali választó menü rész segítségével érhetők el.

A hagyományos követelmények megjelenítésekor az ablak lényegében két táblázatot tartalmaz, amelyekben az osztály, majd a kritérium erőforrások közt navigálhatunk.

A bal oldali táblázatban az iskola egyes osztályai közt választhatunk. A kiválasztott osztályhoz tartozó követelményeket az 

ikonra kattintva jeleníthetjük meg a jobb oldali táblázatban.

A jobb oldali táblázat tehát osztályonként mutatja a követelményeket, pontosabban a táblázat 

oszlopa tartalmazza az iskola minden tantárgyát, és így soronként megjelenítésre kerül, hogy az adott osztályt az adott tantárgyból ki oktatja, mely teremben, heti hány órában, és ha esetleg naponta több órában, akkor legfeljebb mennyiben.

Abban az esetben, ha az adatbázisban szereplő kritérium erőforrás adatait szeretnénk módosítani, akkor ezt az  ikonra kattintva tehetjük meg. Kezdetben az ikon inaktív, aktívvá válik, ha a táblázatból kiválasztottunk egy erőforrást szemléltető sort. Az ikonra kattintva az asszisztensi felület inaktívvá válik és megjelenik az erőforrás módosítására szolgáló ablak.

A megjelenő felületen az adott erőforrás adataival kitöltött elemek szerepelnek, a kívánt módosítások elmentéséhez az 

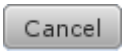
gombra kattintunk, vagy ha mégsem akarunk módosítani

akkor a  gombra vagy  ikonra kattintva térhetünk vissza az adminisztrátori felületre.

A speciális követelményeket kezelő felület megvalósítására nem került sor, de funkcionális működése a következő lett volna. Megjelenítésekor az ablak egyetlen táblázatot foglal magába, amely a speciális órák órarendbe viteléhez szükséges kritériumokat tartalmazza.

A kritériumok ebben az esetben az osztály, tanár, tantárgy, terem erőforrások több elemű halmazából állnak. Ezen kritériumok segítségével vihetjük fel az olyan órákat, amikor egy óra keretén belül az oktató egyszerre több osztályt is tanít. Ekkor a halmazba felvesszük a termet, az oktatót vagy oktatókat, a tantárgyat és az osztályokat.

Az egyes elemek módosítását az  ikonra kattintva tehetjük meg. Kezdetben az ikon inaktív, aktívvá válik, ha a táblázatból kiválasztottunk egy erőforrást szemléltető sort. Az ikonra kattintva az asszisztensi felület inaktívvá válik és megjelenik az erőforrás módosítására szolgáló ablak.

A megjelenő felületen az adott erőforrás adataival kitöltött elemek szerepelnek. A kritériumot alkotó osztály, tanár, tantárgy, terem erőforrások módosításának elmentéséhez az  gombra kattintunk, vagy ha mégsem akarunk módosítani akkor a  gombra vagy  ikonra kattintva térhetünk vissza az adminisztrátori felületre.

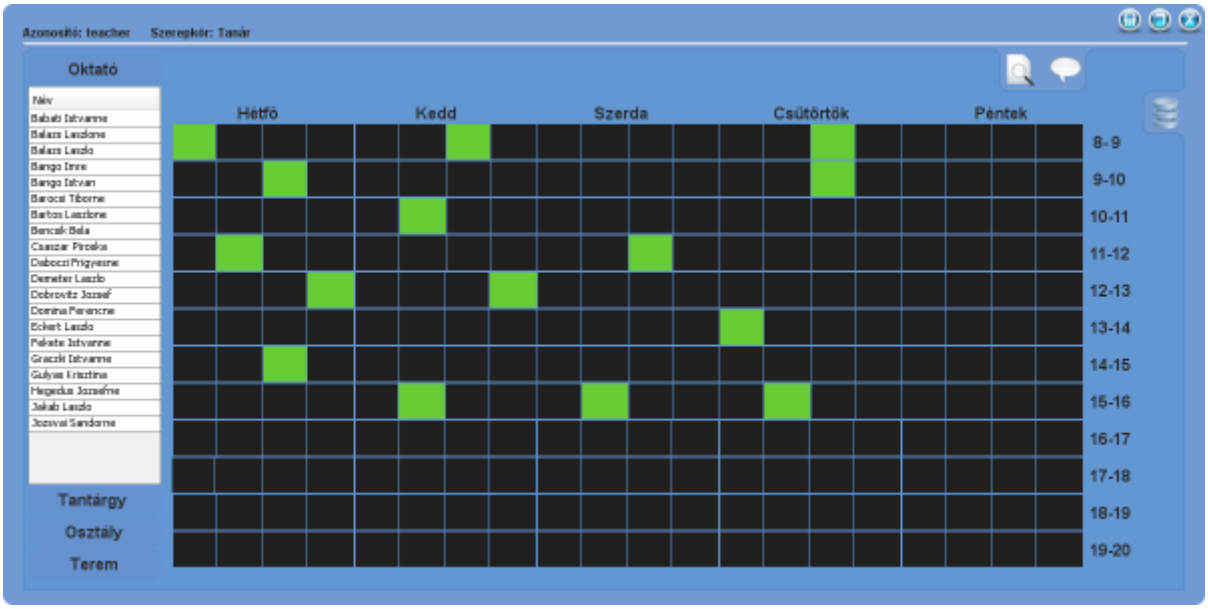
A felhasználói felületen megadható kritériumokon kívül az algoritmus kezeli a lyukasórával illetve az órák egyenletes eloszlásával kapcsolatos feltételeket is.

A felhasználó a felület jobb alsó részében található  ikonra kattintva indíthatja el az órarend generálás folyamatát. Ekkor az asszisztensi felület inaktívvá válik és a generálás folyamata alatt a következő ablak jelenik meg.

A genetikus algoritmusok problémakörét az **1.** fejezetben részletesen tárgyaltam, az algoritmus futási idejét nagymértékben befolyásolja a számításban résztvevő kritériumok száma.

3.3.3.3. A tanár ablak

Ha a felhasználó tanár jogkörrel rendelkezik, valamint a bejelentkezési ablak mezőit megfelelően töltötte ki, akkor következő ablak jelenik meg:



A tanár felhasználói ablak két felülettel rendelkezik:

- az első felületet az  ikonra kattintva érhetjük el.
- a második felületet az  ikonra kattintva érhetjük el.

Az első felületen a bejelentkezett felhasználó a genetikus algoritmus által generált órarendet tekintheti meg, az erőforrások szerint különböző nézetekben.

Az egyes nézeteket a baloldali elem segítségével választhatja meg, amelyben az 

, 

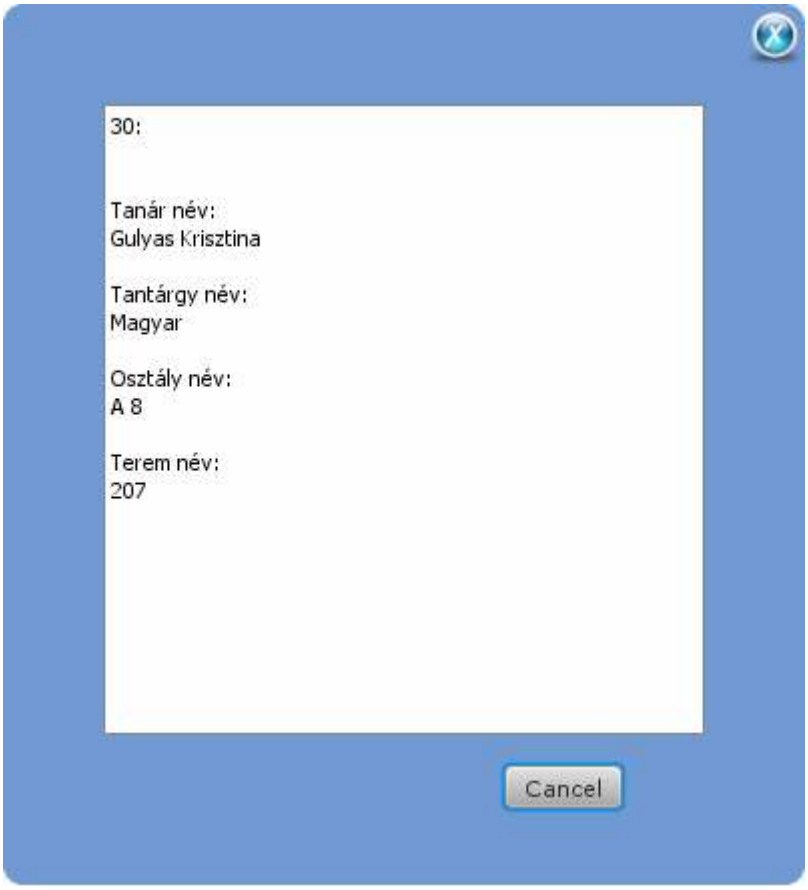
, 

, 

ikonok valamelyikére kattintva az adott erőforrásokat tartalmazó táblázat nyílik le. Kiválasztva a megfelelő erőforrást a lenyíló táblázatból, majd újból az adott ikonra kattintva, az órarendben megjelenik az adott erőforrás, a hozzárendelt szín által ábrázolva.

Az órarend táblázat minden egyes időpontja annyi almezőt tartalmaz, ahány osztálya van az iskolának, így lehetőség van az egyes erőforrás csoportok összes elemének egyidejű megjelenítésére, ezáltal is elősegítve az egyes erőforrások egymáshoz viszonyított vizsgálatát.

Az órarend táblázatban a megfelelő elemre kattintva megtekinthetjük annak részletes adatait a következő ablak formájában:



A második felület segítségével az oktató meghatározhatja a genetikus algoritmus futásához szükséges ráérési követelményeit egy táblázat segítségével.

A táblázat a következő speciális elemből épül fel:

Az elemre kattintva az öt fokozatban állítatható, a zöld színtől a pirosig. A zöld szín választása esetén az elem által képviselt időpont az oktató számára teljesen elfogadott. Egyre magasabb fokozatokat választva az elemen, az elem által képviselt időpont az oktató számára egyre inkább elfogadhatatlanná válik. Az oktató a ráérési követelményeit a  ikon megnyomásával mentheti el.

A táblázat beállítását a követező példán keresztül szemléltetem. A példán jól látszik, hogy a tanár a pénteki napokon, illetve délután öt óra után nem kíván órákat tartani, valamint látszik, hogy a csütörtök kora délutáni órákban is esetleg más elfoglaltsága van.

4. Telepítési útmutató

A következő részben ismertetem az alkalmazás futtatásához szükséges szoftvertermékek telepítési folyamatát, illetve azok főbb beállításait.

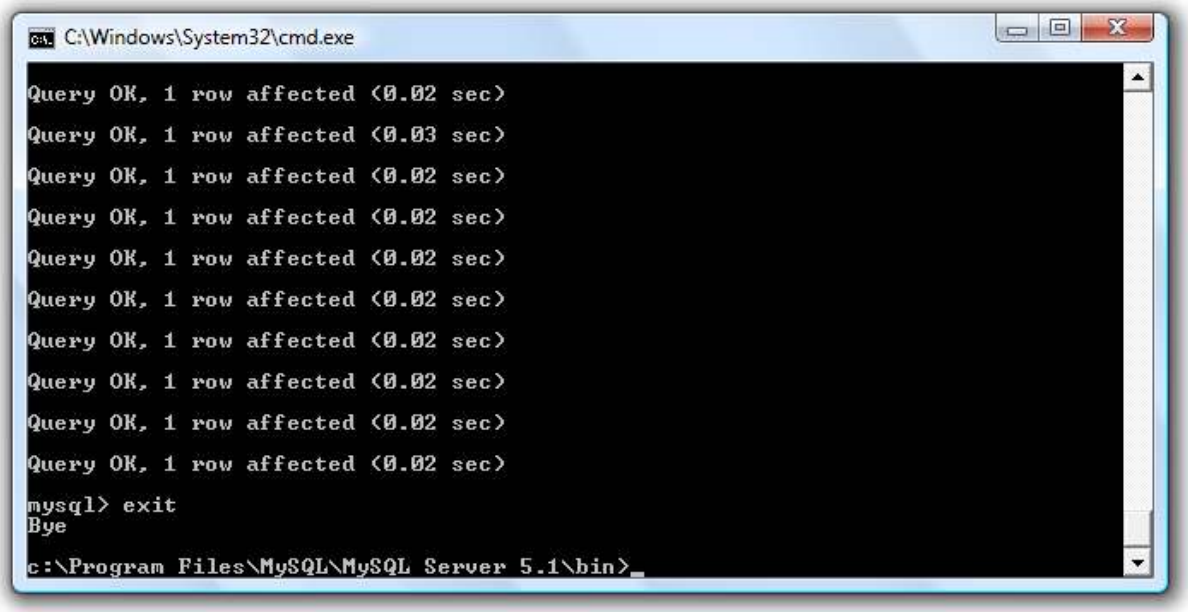
Első lépésként telepítsük a MySql adatbázis-kezelő rendszert. Ezt a CD-n található `mysql-essential-5.1.43-win32.msi` futtatható állomány segítségével tehetjük meg, a telepítés folyamán kövessük a telepítő varázsló utasításait.

A telepítést követően automatikusan elindul a konfigurációs varázsló, amely segítségével testre szabhatjuk a rendszert. A `MySQL GUI Tools 5.0` eszközzel az adatbázist könnyen módosíthatjuk, kezelhetjük. Az állományt csupán a célkönyvtárba kell kicsomagolni.

Az adatbázis-kezelő rendszer telepítése után telepítsük az alkalmazás adatbázisát, a CD-n található `manhattandb_schema1.sql` vagy `manhattandb_schema2.sql` állományok egyikével. Az állományok az órarend generáláshoz szükséges erőforrásokat tartalmazzák, segítségükkel egy nagyobb, illetve egy kisebb iskola órarendje készíthető el.

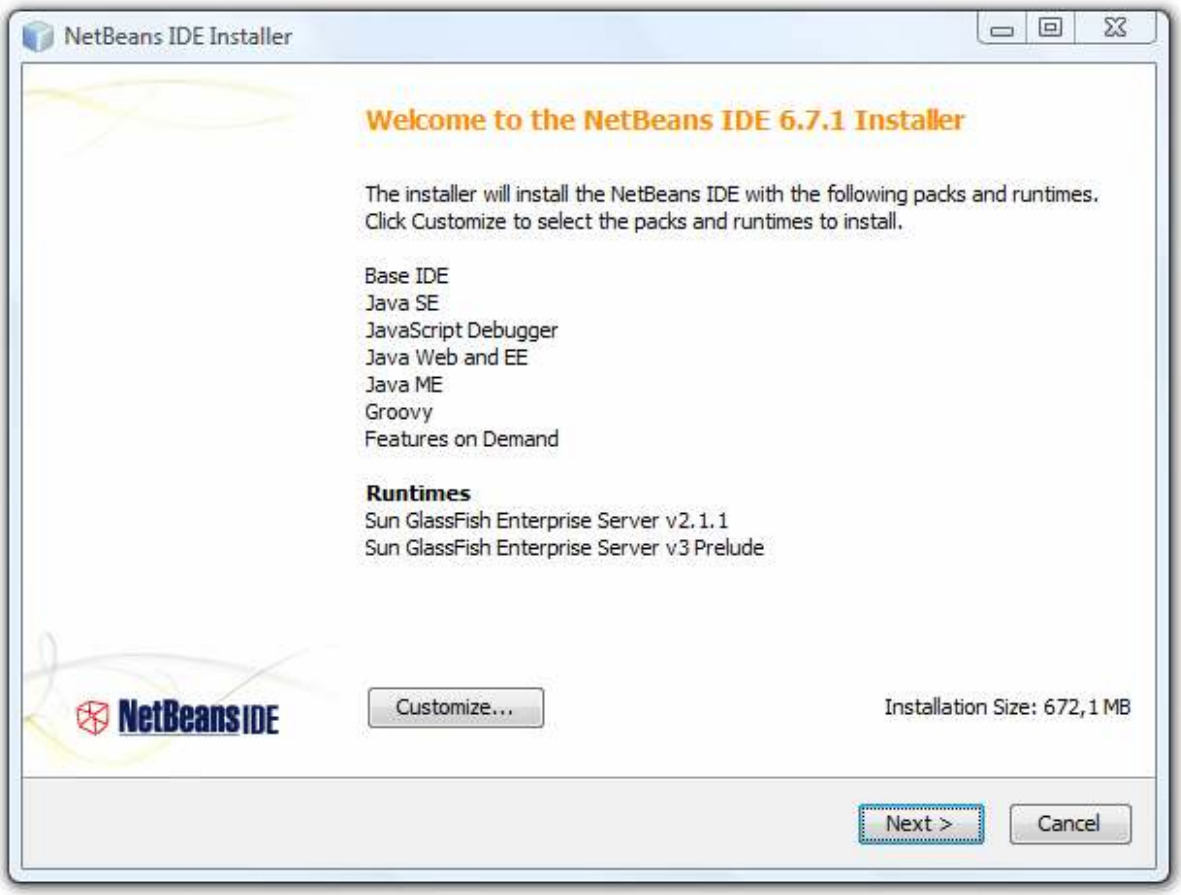
A telepítéshez másoljuk az állományokat a MySql adatbázis-kezelő rendszer `bin` alkönyvtárába, majd futassuk az itt található `mysql.exe` állományt parancssorból a következő paraméterekkel:

Mint látható megadtuk az azonosítót, a jelszót, majd a `source` parancs után az állomány nevét. A parancs hibamentes végrehajtása után az adatbázis sikeresen importálódott az adatbázis-kezelő rendszer adatbázisai közé.

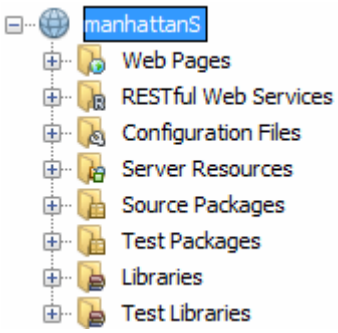


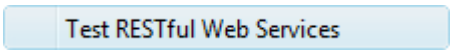
Ezt követően telepítsük a Java illetve JavaFX futtató környezeteket a mellékelt CD-n található `jdk-6u19-javafx-1_2_3-windows-i586.exe` állomány segítségével.

A RESTful webszolgáltatás támogatásához telepítsük a CD-n található `netbeans-6.7.1-ml-java-windows.exe` futtatható állományt, amely tartalmazza a GlassFish Server v2.1 alkalmazás szervert, illetve a NetBeans 6.7.1 integrált fejlesztőkörnyezetet.

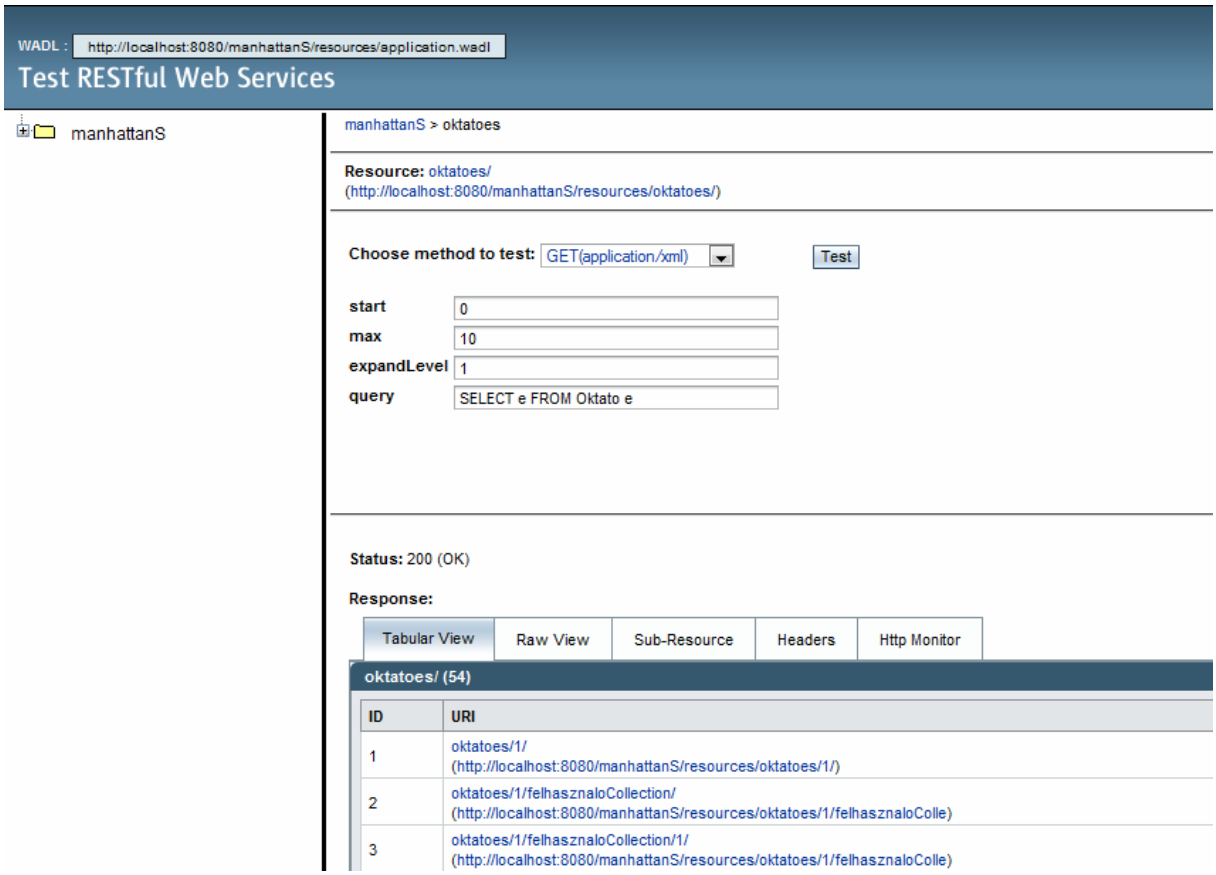


A fejlesztőkörnyezetben nyissuk meg a CD-n mellékelt `manhattanS` projektet:



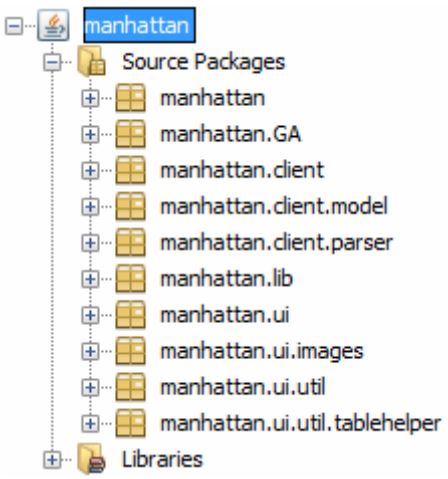
Lehetőség van a webszolgáltatás működésének tesztelésére egy grafikus felületen keresztül. A tesztfelület a 

menüelem megnyomása után automatikusan megnyílik a böngészőben.



Ezen a felületen lehetőség van az egyes erőforrásokon a HTTP műveletek tesztelésére és ezen műveletek eredményének megtekintésére.

A tesztelést követően nyissuk meg a CD-n mellékelt `manhattan` projektet:



A 

menüelem kiválasztásával indítsuk el először a RESTful webszolgáltatást, majd a kliens oldali alkalmazást.
Ezen lépések végrehajtása után az alkalmazás elindul és kész az erőforrások kezelésére, az órarend generálására.

Összefoglalás

Az elkészített és dolgozatomban bemutatott alkalmazás segítségével egy iskola működéséhez elengedhetetlen órarend generálható, figyelembe véve a tanárok és diákok elvárásait, valamint az iskola lehetőségeit.

Dolgozatom első fejezetében egy rövid történelmi áttekintés után ismertettem a genetikus algoritmus főbb elemeit. Ezek közül megemlíteném a reprezentációt, amely az algoritmus működése szempontjából döntő tényező. Ebben az esetben határozzuk meg, hogy a különböző generációkon keresztül az egyes populációk elemeit, azaz az órarend példányokat milyen formában tároljuk. A fejezetben továbbá tárgyaltam a különböző kiértékelési és jóság függvények, valamint kiválasztási módszerek típusait, amelyek megfelelő megválasztásával az algoritmus generációról generációra képes jobb egyedeket előállítani. Kiválasztva a megfelelő szülőket, amelyre alkalmazva a genetikai operátorokat, még jobb értékkel rendelkező gyermek egyedek állíthatók elő. A fejezet részét képezte a hagyományos órarend reprezentáció ismertetése és az ábrázolásból következő hátrányok taglalása, valamint egy új, céljaimnak megfelelő reprezentáció, amely alkalmazásával a különböző óra típusok meglehetősen egyszerűen kezelhetők, az órarend összeállítása során. A fejezet utolsó részében ismertettem az algoritmus konkrét megvalósítását, illetve a rendszer hatékonyabb működését elősegítő finomhangolási tényezőket.

A felhasználói felület és az adatbázis közötti kapcsolatot REST típusú webszolgáltatás koordinálja. Diplomamunkám második fejezetében egy rövid történelmi áttekintés és a REST típusú webszolgáltatás fogalmainak kifejtése után, ismertetem az alkalmazás mögött álló adatbázis szerkezeti és működési felépítését. Ezek után részletesen taglalom az alkalmazás kódrészleteinek magyarázatával a webszolgáltatás megvalósítását, a szerver illetve kliens oldalon.

A harmadik fejezetben egy rövid áttekintés után, amely a JavaFx nyelv megalkotásának okait ragozza, egy egyszerű példán keresztül kiemelem a nyelv főbb elemeit. Ezt követi a felhasználói felületek, azok szerepei és a rajtuk végrehajtható funkciók ismertetése.

Sikerült megvalósítanom a kitűzött célom, azaz a program alkalmas egy iskola működéséhez elengedhetetlen erőforrások menedzselésére. Egy életképes órarend megalkotásához szükséges követelmények kezelésére, illetve mindezek alapján egy közel optimális órarend generálásához, továbbá az órarend különböző erőforrások szerinti megjelenítésére.

A fejlesztés során fontos volt számomra, hogy a program átlátható és moduláris legyen, ezzel is támogatva a későbbi fejlesztés lehetőségét. Egy esetleges továbbfejlesztési irány lehetne az egészen összetett órák kezelésének lehetősége, valamint a webszolgáltatás működésének optimalizálása.

Irodalomjegyzék

[1] David E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.

[2] Álmosi Attila, Győri Sándor, Horváth Gábor, Várkonyiné Kóczy Annamária: Genetikus algoritmusok, Typotex kiadó, 2002.

[3] Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, Peter Brittenham, Yuichi Nakamura, Paul Fremantle, Dieter Koenig, Claudia Zentner: Java alapú web szolgáltatások. Kiskapu kiadó, 2001.

[4] Roy Thomas Fielding Architectural Styles and the Design of Network-based Software Architectures, Dissertation, 2000.

[5] http://www.ady-debr.sulinet.hu/

[6] http://people.inf.elte.hu/sirpepe/ea_presentations.html

[7] http://jgap.sourceforge.net/

[8] https://jsr311.dev.java.net/

[9] http://dlc.sun.com/pdf/820-4867/820-4867.pdf

[10] http://blogs.sun.com/enterprisetechtips/entry/implementing_restful_web_services_in

[11] http://java.sun.com/developer/technicalArticles/glassfish/GFandMySQL_Part4Intro.html

[12] http://hu.wikipedia.org/wiki/GlassFish

[13] http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

[14] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

[15] http://www.javapassion.com/handsonlabs/javafx_webservices/

[16] http://java.sun.com/javafx/index.jsp

[17] http://www.javafx.com/

[18] http://www.javapassion.com/javafx/

[19] http://openjfx.java.sun.com/current-build/doc/reference/JavaFXReference.html

[20] http://java.sun.com/javafx/1.2/docs/api/

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Rutkovszky Edénének a dolgozat készítése során nyújtott szakmai segítségért és útmutatásért. Köszönet illeti Rózsavölgyi Gábor igazgató urat, hogy rendelkezésemre bocsátotta a debreceni Ady Endre Gimnázium aktuális órarendjét, amelyből megismertem egy órarend szerkezeti, strukturális felépítését és az összeállításához szükséges feltételeket.

Hálás köszönettel tartozom *családomnak, szeretteimnek,* akik mindvégig mellettem álltak, nélkülük ez a diplomamunka nem születhetett volna meg.