

Eötvös Loránd Tudományegyetem  
Informatikai Kar

## **Menetrendi adatok feldolgozása és megjelenítése**

Nyitrai Erika  
egyetemi tanársegéd

Balog Péter  
Programtervező informatikus BSc

Budapest, 2012



## Tartalomjegyzék

Bevezető .....	5
A program .....	5
Környezet .....	5
Eszközök .....	6
Hozzáállás.....	6
Kezdeti terv, nehézségek .....	7
Felhasználói dokumentáció .....	9
Bevezető .....	9
Futtatási környezet .....	10
Kliens.....	10
Szerver .....	14
Üzembe helyezés .....	14
Rendszerüzenetek.....	15
Fejlesztői dokumentáció .....	16
Az alkalmazás dióhéjban.....	16
Felhasznált technológiák .....	17
Nyelvek: .....	17
Technológiák:.....	17
Külső könyvtárak, eszközök: .....	19
Tiszta Kód .....	20
Java konvenciók .....	20
Változók .....	20
Függvények .....	21
Osztályok.....	21
Adatbázis.....	22
Demeter törvénye(Law of Demeter) .....	22
„Extract Till You Drop” - szabály .....	23
Kommentek.....	23
Az indítás és futtatás elválasztása .....	23
Refaktorálás .....	24
Az úttörők szabálya(Boy Scout rule) .....	24

Rendszerarchitektúra.....	25
Programtervezési minták a kódban.....	25
Ant script.....	27
Csomag, Modul és Osztályszerkezet.....	28
Adatbázis.....	36
Felhasználói esetek.....	40
Tesztelés.....	42
Manuális tesztek.....	42
JUnit.....	45
Skálázhatóság.....	46
RDF Adatbázis.....	47
Összegzés.....	49
A kód tisztasága.....	49
Kezdeti problémák.....	50
Kudarok.....	51
Sikerek.....	51
Függelék.....	52
Felhasznált irodalom.....	53

# Szakdolgozat

---

## Bevezető

### A program

A programom egy weboldal, aminek két célja van, felhasználói szempontból. Az egyik, hogy bárki kényelmesen találhasson magának és ajánlhasson másoknak utazási célt, és ott található érdekes látnivalókat, szórakozóhelyeket. A másik, hogy népszerűsítse és egyszerűvé tegye az országon belüli utazást.

Az első célt egy térképes kereső biztosítja. Az oldalon egy térképen lehet úticélokat keresni és kinézni. A kinézett utazási célok magyarországi falvak vagy városok, ahova el lehet jutni vonattal. Ezekben az utazási céloknál meg lehet nézni a látnivalókat, szórakozóhelyeket, múzeumokat, és egyéb érdekes helyeket, illetve bárki hozzáadhat egy általa ismert új érdekességet is.

A másik célt a kereséssel összefűzött elvira MÁV menetrendje biztosítja. Az utazás keresésekor időpontot és a vonatokon igénybevehető kedvezményeket is ki lehet választani. Ezekkel az adatokkal az utazással kapcsolatos vonat információkat is megjeleníti a honlap, például a vonatok indulását, érkezését, a jegy árát, indulási vágány számát és egyéb hasznos adatokat mutat meg, minden keresés alkalmával.

Fejlesztői szempontból a kódnak szintén két célja van. Az egyik, hogy elsajátítsam a Java nyelvet, és egy hozzá tartozó új és elterjedőben lévő keretrendszert. A másik pedig, hogy megtanuljak kódot írni úgy, hogy az valóban öndokumentáló legyen, és mindenki más is könnyedén megértse azt, amit írtam.

### Környezet

A feltelepített program az internetről bárholnan elérhető, és bármelyik korszerű böngészővel használható. A program logikai kódját Java (1.6) nyelven készítettem, a grafikus felületet pedig XML, HTML és CSS nyelveket felhasználva raktam össze. Az

elkészült alkalmazás egy war fájlként futtatható egy tetszőleges web-szerveren, például Glassfish vagy Apache Tomcat application server segítségével.

## Eszközök

A keretrendszer amiben az alkalmazást fejlesztettem a Google Web Toolkit (GWT)<sup>1</sup>, ami roppantul leegyszerűsíti a webes alkalmazások fejlesztését, és elfedi a böngészők közötti különbséget. A keretrendszer biztosítja kliens és a szerver közötti egyszerű kommunikációt Remote Procedure Call (RPC)<sup>2</sup> segítségével, így ezt választottam az összekapcsolásukra. Sajnos ezt a technológiát nem használhattam ki a MÁV Elvira rendszerének lekérdezéséhez. Az Elvira egy Representational State Transfer (REST)<sup>3</sup> API-n keresztül kérdezhető le, így a tőle szükséges adatokat ezen keresztül szereztem be. A program egy saját Resource Description Framework (RDF)<sup>4</sup> adatbázist használ az útvonalak és városok meghatározásához, ezt bővíti a felhasználók által megadott adatokkal.

## Hozzáállás

*„ ... the most important way to be expressive is to **try**. ”*

*Robert C. Martin*

A szakdolgozat írásakor nagyon nagy hangsúlyt fektettem a kód minőségére és átláthatóságára. A munkámra nagy hatással volt Robert C. Martin, Steve McConnell és Martin Flower könyvei, amikben az olvasható, tiszta kód készítését írják le. Igyekeztem a szakdolgozatnak minden részét modulárisan, kicsi és könnyen érthető elemekből felépíteni, és bár nem sikerült mindenhol megvalósítani az elveket, de mindenhol törekedtem a kódom olvashatóságára és szépségére, az öndokumentáló változó, függvény és osztálynevek használatára.

---

<sup>1</sup> GWT: Google Web Toolkit – Keretrendszer webes alkalmazások fejlesztéséhez

<sup>2</sup> RPC :Remote Procedure Call – Elosztott rendszerek kommunikációjához használt technológia

<sup>3</sup> REST:Representational State Transfer – http protokoll segítségével elosztott rendszerek hálózati kommunikációjához használt technológia

<sup>4</sup> RDF: Resource Description Framework – Gráfolapú adatbázis ami XML formátumban tárolja az adatokat

## Kezdeti terv, nehézségek

A szakdolgozat témabejelentőjében az áll, hogy PDF-ből kinyert adatokat használok fel az alkalmazás RDF adatbázisának létrehozásához. A bejelentő megírásakor ugyan elég határozott elképzelésem volt az alkalmazásról, de nem tudtam sokat arról, hogy ennek megalkotásához milyen adatokra lesz szükségem. Tervezés és a kezdeti prototipizálás közben nyilvánvalóvá vált, hogy a MÁV honlapján található PDF-ek nem tartalmazzák az alkalmazáshoz elengedhetetlen információkat, például a megállók koordinátáit, vagy a közöttük lévő utak leírását, amik a térképes megjelenítést tették volna lehetővé. Így aztán más adatforrást kellett keresnem. A Google térképes tömegközlekedés útvonalainak leírásához létezik egy General Transit Feed Specification (GTFS)<sup>5</sup> nevű formátum, ami szimpla szöveges txt fájlokból, formáját tekintve Comma Separated Values (CSV)<sup>6</sup> adatbázisfájlokból áll. Ezek ugyan nem a kívánt formában, de minden adatot tartalmaztak arról, amire szükségem volt a térképes megjelenítéshez, a hiányzó adatokat pedig az elvirának egy nem hivatalos REST apijától beszerezve pótoltam ki. Úgy érzem, hogy a váltás nem könnyítette, inkább nehezítette dolgomat a program írása során, mivel több és különböző módon elérhető forrásokból kellett egyedi adatbázist felépítenem. A GTFS és a PDF fájlokat egyaránt egy külső program segítségével (CSVReader illetve IText) alakítottam olyan formára, amit fel tudtam dolgozni, és egyik sem a pontos adatokat tartalmazta; szűrésekre, adatok összekapcsolására és nagy adatmennyiség feldolgozására ugyanúgy lehetőségem nyílt.

Az adatbázis építése során a nagy adathalmaz miatt beleütköztem az adatok feldolgozásának lassúságába, mert az adatbázis első verziója körülbelül 4 óra alatt készült el. Ez természetesen optimalizálásra szorult, és emiatt a feladat ezen részét izgalmassá és roppant szórakoztatóvá tették.

Ugyan tervezetten, de nagyon nagy nehézségekbe ütköztem a technológiák elsajátítása közben. A szakdolgozat megírása és megtervezése során célom volt, hogy minél több új technológiát elsajátítsak elkészítése alatt. Ezt a célt sikeresen elértem. Az írás közben újonnan megismert technológiák között volt a Java, a Google Web Toolkit webes

---

<sup>5</sup> GTFS:Google Transit Feed Specification – A Google térképén megjelenő tömegközlekedés adatainak szerkezetét leíró specifikáció

<sup>6</sup> CSV: Comma Separated Values – primitív adatbázisfájl-típus, a nevéből kikövetkeztethető működési mechanizmussal

keretrendszer, az Ant build-script megismerése, a Git verziókövető rendszer, az Remote Procedure Call technológia, amivel a szerver és a kliens közötti kommunikációt lehetett könnyedén áthidalni, a REST és a hozzá kapcsolódó elmélet megismerése és ezzel számítógépek magasszintű, http protokollt használó kommunikációjának módjai a hálózaton, a JUnit tesztek megismerése és írása, az RDF adatbázis és a szemantikus web fogalmának megismerése.

A fejlesztés elején a használt technológiák megismerése, kipróbálása kapott kulcsszerepet. Mivel minden új volt, képtelen lettem volna előre tervezni a programot, így a kezdeti terv szándékosan elnagyolt volt, részletek nélkül. A szakdolgozat számomra egy olyan feladat volt, amit Steve McConnell Code Complete 2 című könyvében „Wicked Problem”-ként ír le: egy probléma, amit csak úgy lehet specifikálni, ha először megcsinálom (McConnell, 2004, old.: 75). Az útközben felmerült problémákra így mindig akkor reagáltam amikor felmerültek, és nem próbáltam meg kitalálni azt, amihez nem is értek még. Ebben rengeteget segített a Git verziókövető rendszer, ami nélkül a program a kísérletezések közben biztosan darabjaira hullott volna. Az eszközeim használatáról rengeteget olvastam, a velük járó program struktúrák és jó szokásokat pedig igyekeztem útközben magamévá tenni, így alakult ki a szakdolgozat végleges állapota.

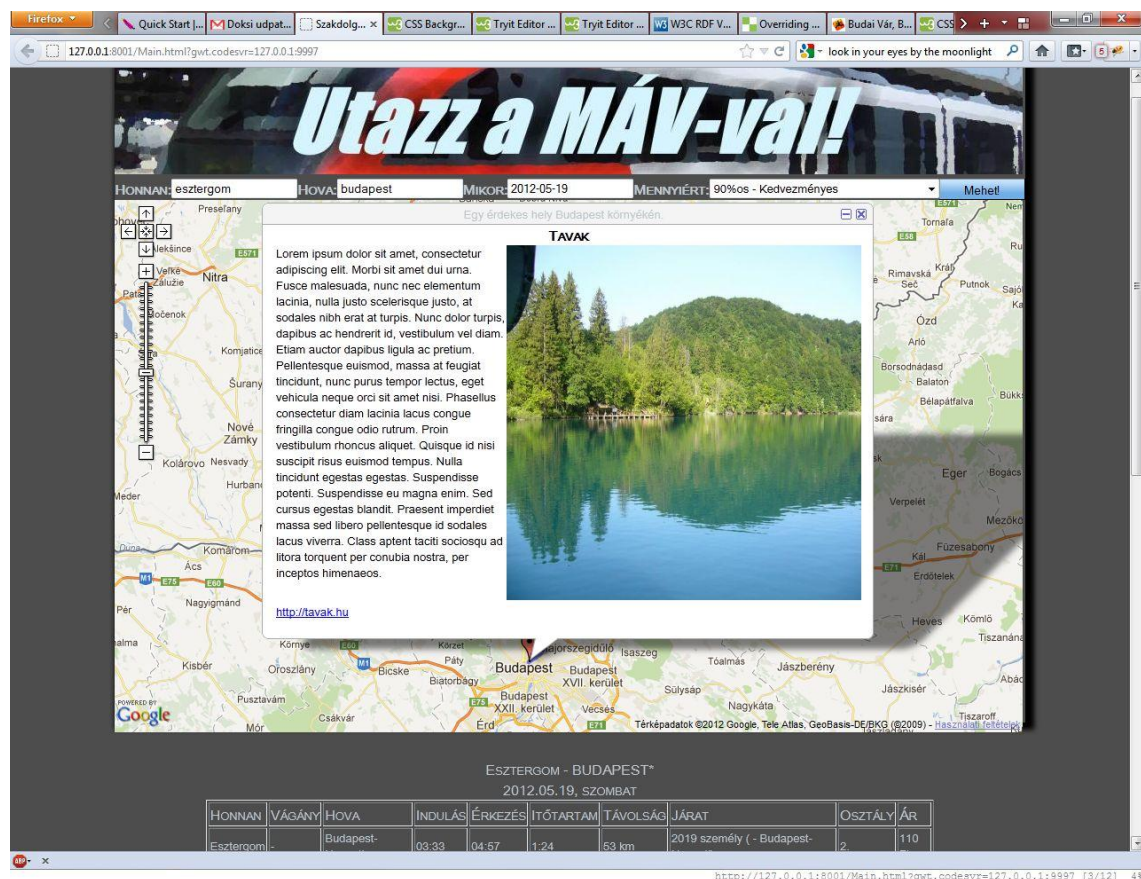


# Felhasználói dokumentáció

## Bevezető

Ez a program egy webes alkalmazás ami információkat biztosít olyan kirándulásokhoz, utazásokhoz, ahol az utazók vonattal terveznek menni. A honlapon a célállomások érdekességei közt lehet böngészni, és új érdekességeket hozzáadni.

Mivel a weblapnak sürgősszerű a felhasználók adatait eltárolni a feladat ellátásához, a honlap használata regisztrációhoz illetve bejelentkezéshez nem kötött, bárki szabadon használhatja. A széles felhasználói közönség miatt a felhasználói felületet a lehető legegyszerűbbre lett tervezve.



## 1. Felhasználói felület

## Futtatási környezet

### Kliens

A weboldal minden modern böngészőben működik, ahol a „modern” a jelenleg letölthető legfrissebb böngészőket jelentik. A program a Firefox 11 és a Chrome 18.0 verziójaival lett letesztelve és fejlesztve, így működése ezeken a böngészőkön garantált. Ezen kívül a böngészőben engedélyezve kell lennie a javascript kód futtatásának, és mivel honlapról van szó, természetesen szükséges az internet hozzáférés is.

A honlap megnyitásakor a következő felület jelenik meg:



### 2. A honlap megnyitáskor

Itt a térképen lehet navigálni, az egér görgőjével a nagyítási szintet állítani, az egér bal gombjának nyomvatartásával és arrébmozgatásával pedig arrébhúzni a képet.

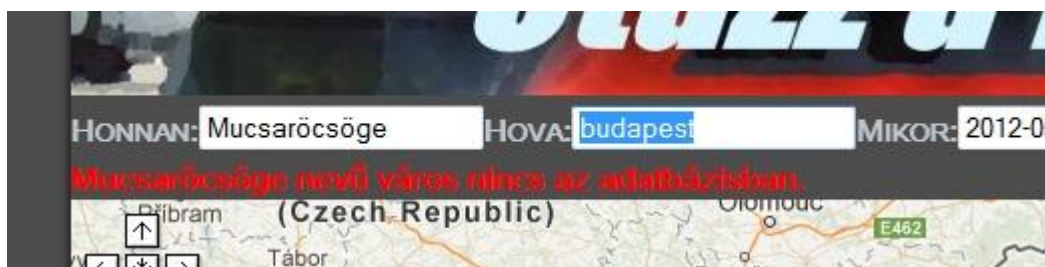


A térkép fölötti sávban találhatóak a szűréshez használandó elemek. Itt lehet megadni hogy honnan tervezzük az utazást, hova tervezünk utazni, mikor tervezünk utazni, és be lehet állítani az utazáshoz egy tarifát is, ami a MÁV vonatjegyek kiírt árát változtatja. A dátumot és a tarifát legördülő menüből választhatjuk ki. A dátum alapértelmezett értéke az aktuális nap, a tarifának az alapértelmezett értéke a képernyőn látható kedvezmény, így ezeket a paramétereket nem kötelező megadni.



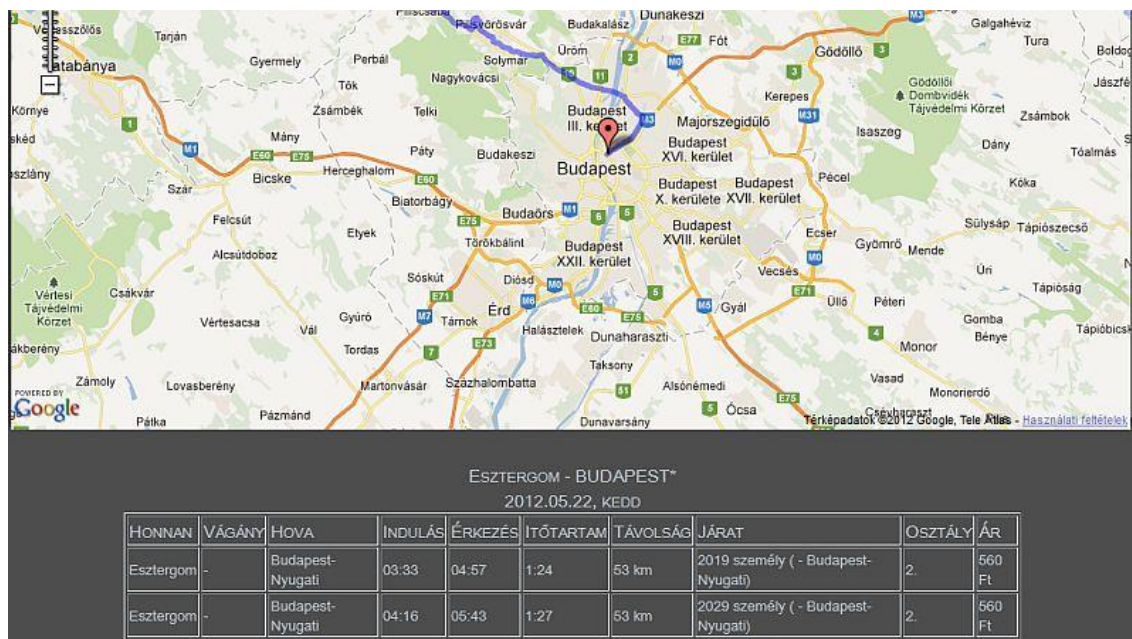
### 3. A megadható adatok

Ha rossz adatot adunk meg, vagy hiányosat, akkor egy hibaüzenet jelenik meg a mezők alatt, értesítve a problémáról:



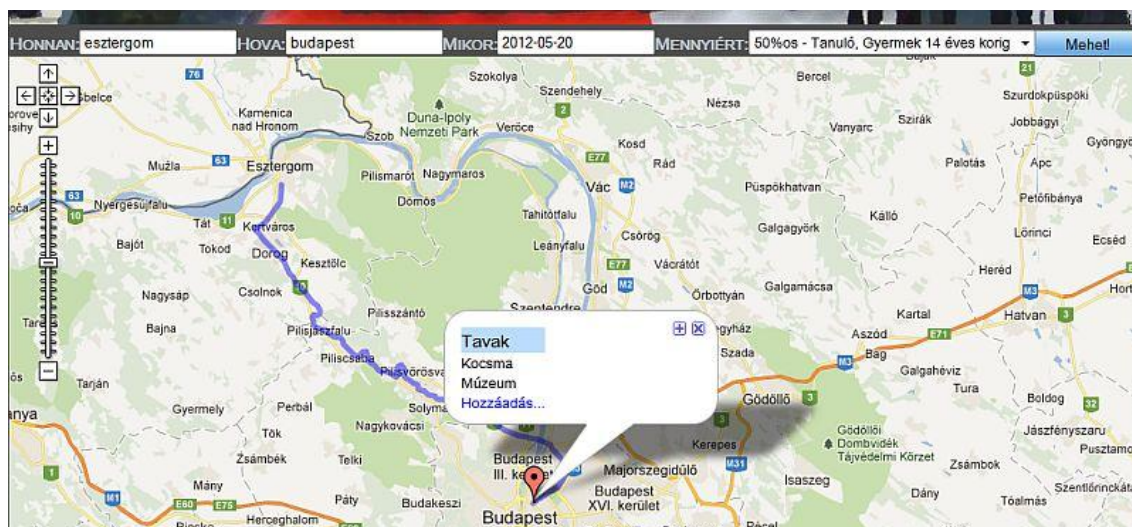
### 4. Hibaüzenet

Ha a mezők megfelelően ki vannak töltve, a mellettük lévő kék „Mehet!” gombra kattintva kérhetjük le az utazás információit. Ez két dolgot jelent. Egyrészt megjelenik az utazás útvonala és a célállomás helyén egy jelölő a térképen, valamint a térkép alatt egy táblázatban megjelennek az utazással kapcsolatos információk.



#### 5. A táblázat megjelenik a térkép alatt

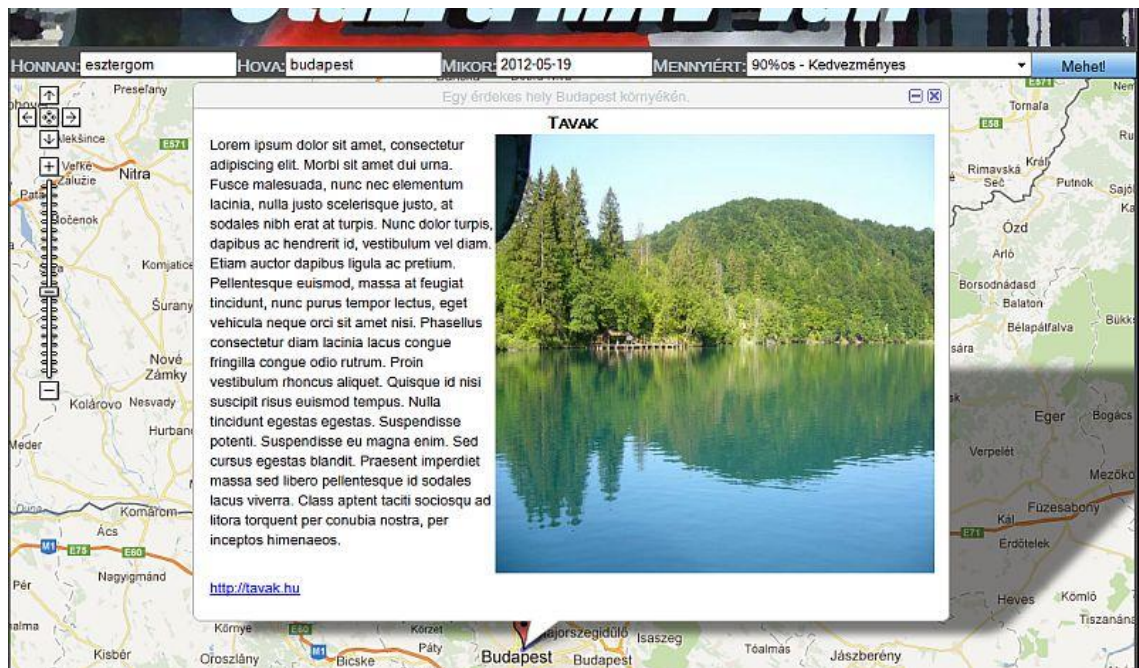
A térképen lévő jelölőre kattintva nyithatjuk meg a listát azokról a helyekről, amik a célállomáshoz kapcsolódnak.



#### 6. Érdekes helyek listája a térképen

Ha a lista üres, akkor egy üzenet a lista helyén közli velünk a szomorú hírt „*Itt nincsen semmi érdekes...*” felirattal. Ha nem üres, akkor a lista elemeire kattintva megnézhetjük részletesen egy-egy hely leírását.

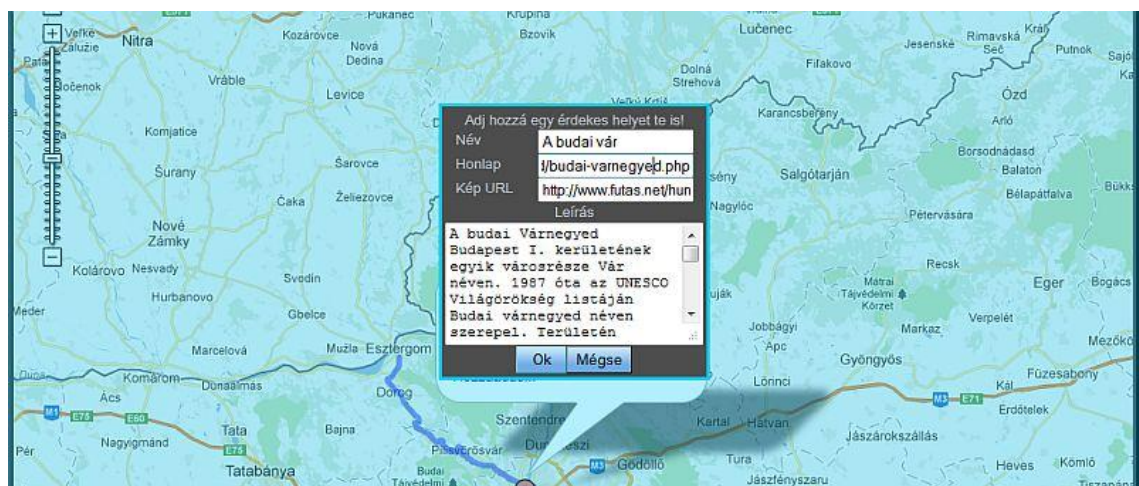




## 7. Egy hely részletei

Amennyiben vissza szeretnénk térni a listához, a jobb felső sarokban lévő vízszintes vonallal jelölt gombbal lehet az ablakot visszakicsinyíteni, és a listában tovább böngészni. A hely linkjére kattintva a weboldala a böngésző egy új tabjában nyílik meg.

A honlap lehetőséget biztosít új hely felvételére is az adatbázisba. Mind az üres, mind az elemeket tartalmazó lista legutolsó eleme a „Hozzáadás” gomb, amit megnyomva egy felugró ablak jelenik meg.



## 8. Új hely létrehozásakor felugró ablak

Ezen a felugró ablakon a Név, Honlap, Kép URL, és a leírás kitöltésével vehetünk fel egy új elemet a listába, amit később más felhasználók is megnézhetnek.

## Szerver

A szerver futtatási környezete egy tetszőleges Web Application Server(WAS)<sup>7</sup>, azaz alkalmazásszerver (például Glassfish vagy Tomcat), amire a war kiterjesztésű fájlban lévő alkalmazást fel kell telepíteni. Ennek a telepítésnek a részletei változóak, attól függően, hogy milyen WASra telepítünk. A szerveroldali program, mint minden Java program, egy Java Virtual Machine (JVM)<sup>8</sup>-ban fut, aminek virtuális memória mérete legalább 500 MB méretű kell hogy legyen.

## Üzembe helyezés

A szerver üzembe helyezése három pontban:

### 1) Fel kell telepíteni az alkalmazásszerverre a programot tartalmazó warfájl<sup>9</sup>

- a. Ennek menete az alkalmazásszervertől függ, telepítési segédletért annak a dokumentációjából kell ihletet meríteni.

### 2) Megfelelően fel kell paraméterezni az alkalmazás JVM-jét

- a. A JVM maximális memóriaméretét (és ajánlott a minimálist is) legalább 1000 MB nagyságúra kell állítani. Ehhez a JVM indítási paramétereként meg kell adni a következő parancsokat:

`-Xmn100M -Xms500M -Xmx1000M`

- b. Szintén a program indításakor JVM paramétereként meg kell adni a program konfigurációs fájljának helyét és nevét, relativ vagy abszolút elérési útvonallal:

`-Dconfig.location=<konfigurációs fájl>`

pl.: `-Dconfig.location=c:\work\Szakdolgozat\config.properites`

- c. A program a Java alap temp könyvtárát használja futás közben az ideiglenes fájlok tárolásához. Amennyiben ez nem megfelelő, vagy a programnak nincs joga ezt a könyvtárat használni, egy alternatív helyet kell megadni neki:

`-Djava.io.tmpdir=<abszolút elérési út a végén per jellel>`

---

<sup>7</sup> WAS – Web Application Server – Speciális Java alkalmazások futtatási környezetét biztosító alkalmazásszerver

<sup>8</sup> JVM – Java Virtual Machine – virtuális futtatókörnyezet, minden Java alkalmazás egy JVMben fut

<sup>9</sup> warfile – .war kiterjesztésű fájl. A web application archive rövidítése - Javában írt webes program

### 3) Megfelelően ki kell tölteni a program konfigurációs fájlt

- a. Ezzel a fájlal lehet beállítani a program különböző szerveren lévő erőforrásainak helyét, például az adatbázis fájl helyét. Ez egy properties fájl<sup>10</sup>, amiben jelenleg a következő paraméterek adhatók meg:
- pricing.properties.file=<kilistázott kedvezmények helye és neve>  
rdf.database.file=<az RDF adatbázis helye és neve>  
mav.source.zip.url=<forrásokat tartalmazó gtfs zip fájl URL-je>  
példafájl:

\*\*\*config.properties\*\*\*

```
1 pricing.properties.file=../resources/pricing.properties
2 rdf.database.file=/home/matyi/downloads/data.rdf
3 http://data.flaktack.net/transit/mav/latest/feed/gtfs.zip
```

## Rendszerüzenetek

A kliensen kétfajta hibaüzenet jöhet elő, információ-jellegű és kritikus.

Az információ-jellegű az oldal részeként jelenik meg, szöveges üzenetként. Például olyan városnév beírásánál ami nem található az adatbázisban, információ-jellegű hibaüzenet jelentkezik, ami jelzi, hogy rossz/nem található a beírt adat. Itt a hibaüzenet közli a teendőket a hiba elhárításához.

A kritikus hibaüzenetek a rendszer belső hibáját, vagy egyes erőforrások hiányát jelzik. Ezek a szervertől „HTTP 500 hibakódú, internal server error” felirattal jelennek meg a honlap bal felső sarkában egy felugró ablakban. A hibák hatékonyabb kiszűrése miatt ezek közül a hibák közül a legtöbb részletes információt is közöl a szerverben keletkezett problémáról. Ilyen hiba lehet például ha az adatbázis nem létezik, ha nem elérhető vagy nincs jogosultsága írni egy szükséges fájlt a szervernek, és egyéb belső hibák.

---

<sup>10</sup> properties fájl - .properties kiterjesztésű fájl ami kulcs érték párokat tartalmaz egyenlőségjellel elválasztva : key=value

## Fejlesztői dokumentáció

### Az alkalmazás dióhéjban

Ez a program egy webes alkalmazás amit GWT keretrendszerben Java nyelven készült, de nem kizárólag abban, előfordul HTML, CSS, illetve Javascript kód is. A programból egy war fájl készül, amit egy tetszőleges alkalmazáserver futtat, és néhány fontos paramétere megadható induláskor egy konfigurációs fájlban. A kliens és a szerver RPC-n kommunikálnak egymással, aszinkron hívásokkal hívja a kliens a szerver által biztosított szervleteket. A kliens egy weblapot jelenít meg, amin le lehet kérdezni és megnézni egy térképen egy belföldi MÁV-os vonatutazás útvonalát, a célállomás környékén lévő érdekes helyeket, valamint információkat(ár,időpont,stb.) az utazásról. A szerver feladatköre, hogy a kapott paraméterek alapján kikeressen adatokat egy RDF adatbázisból a kliensnek, ami tartalmazza az információt a térképen az utak, a városok és a helyek kirajzolásához. Ezen kívül feladata még nem létező adatbázis esetén annak létrehozása egy GTFS specifikációnak megfelelő adatbázisból úgy, hogy letölti ezt a forrás adatbázist tartalmazó zip fájlt, kitömöríti, beolvassa a CSV formátumban lévő adatokat, és legenerálja belőle az RDF adatbázist. A vonatindulással kapcsolatos lekérdezéseknél egy külső REST API-t hív segítségül, tőle kérdezi le a kliens által kért vonatindulás információit.





A program azon részeihez, ahol megkönnyítette a fejlesztés menetét, JUnit tesztek készültek. Elsősorban az adatbáziskezeléssel kapcsolatos szervlet, a szerver és a kliens között utazó osztályok és a REST API-t hívó osztály lett így letesztelve.

A program a fejlesztés kezdeti szakaszától verziókövető rendszer alatt van, forráskódja és forráskódja fejlődésének részletes történelme megtalálható a [www.github.com/croo/Szakedolgozat](https://www.github.com/croo/Szakedolgozat) cím alatt.



## Felhasznált technológiák

### Nyelvek:

-  Java
-  Javascript
-  HTML
-  CSS
-  XML
-  JSON

### Technológiák:

#### **GWT – Google Web Toolkit**

- A keretrendszer amiben az alkalmazás készült. Lehetővé teszi, hogy a kliens oldali kódot Javascript helyett Java nyelven lehessen megírni, amit aztán a GWT fordítója Javascript nyelvre fordít. A fordító nem csak a végtelenségig kioptimalizálja az írt kódot, de minden böngészőre külön permutációt készít, és később a csatlakozó kliensnek az ő böngészőjéhez illeszkedő kódot küldi el, így biztosítja, hogy a honlap mindenhol ugyanúgy működjön és nézzen ki.

#### **REST - Representational state transfer**

- Ezzel a technológiával készült az Elvira API amitől az alkalmazás egy utazás információit elkéri. Az alkalmazás ezt hívja, egyszerű GET kérésekkel.

A REST elsősorban nem egy technológia, hanem egy hozzáállás, hogy hogyan kellene hálózaton a gépek kommunikálniuk egymással. Az ötlet az, hogy ne találják fel a spanyolviaszt, hanem a jól bevált protokollt, a HTTP-t használják információcserére, ami már sokkal jobban megoldotta ezt. Az erőforrások/adatok uri-kon keresztül érhetőek el a HTTP különböző hívástípusaival : GET, POST, UPDATE, DELETE

#### **RPC – Remote Call Procedure**

- Ez egy másik módszer amivel gépek a hálózaton kommunikálhatnak egymással. A szerver és a kliens ennek segítségével beszélget. Ennek lényege, hogy az alkalmazásban minél jobban elfedje ezt a távolságot, lehetővé téve, hogy a kliensből közvetlenül lehessen hívni a szerver egy függvényét.

#### **RDF – Resource Description Framework**

- Speciális adatbázis, amiben az utak és városok megjelenítéséhez szükséges adatait tárolja a szerver. Az RDF a W3C egy szabványa, ami az adatokat gráfként avagy elemek és azok közötti relációkként tárolja.

#### **GTFS – Google Transit Feed Specification**

- Egy szabványos adatbázis formátum, amit a googlemaps fel tud használni. Ebben írhatja le egy közlekedési vállalat az útvonalait és az azokkal kapcsolatos információkat, ami aztán megjelenik a Google térképes szolgáltatásában. Ezek az adatok publikusak és mindenki számára elérhetőek.

#### **CSV – Comma-Separated Values**

- Ahogy a neve is mutatja, vesszővel elválasztott értékek (néha fejléccel). Az egyik legprimitívebb adatbázisfajta. A GTFS specifikációban ilyen táblákban vannak eltárolva az utazás adatai. A táblák formátumai a specifikáció része.

#### **Junit**

- Java osztályok automatikus függvény/osztályszintű tesztelését lehetővé tevő technológia. A kód néhány fontos osztálya illetve olyan interfésze aminek implementációja drasztikusan változott, ezzel lett tesztelve.

#### **Verziókövetés**

- A verziókövető rendszerben a kód készülés közbeni állapotait el lehet menteni, és később visszatérni hozzájuk, ezzel biztonságossá válik a kód változtatása, és nem probléma a kísérletezgetés vagy egy korábbi verzióra visszaállítás.

## Külső könyvtárak, eszközök:

### Junit4

- Ez a könyvtár biztosítja a Junit tesztelést.

### Apache Jena

- Az RDF adatbázisbeli lekérdezéseket lehetővé tevő könyvtár.

### GoogleMaps API v2

- Ennek a könyvtárnak a segítségével jelenik meg a honlapon a térkép illetve az azokon megjelenő funkciók – utak kirajzolása, felugró menük.

### OpenCSV

- Ez az egyszerű könyvtár megkönnyíti a CSV fájlok beolvasását és feldolgozását Java-ban.

### GWT

- Robosztus keretrendszer saját fordítóval, és rengeteg beépített webes technológiával. Néhány a fontosabb és a kódban használt funkciója:
  - Java kódból javascript kód generálás
  - Javába beilleszthető natív javascript kód támogatással
  - Grafikus felület összeállítás XML-ben
  - Egyedi eseménykezelés
  - RPC kommunikáció a szerver és a kliens kód között

### Ant

- A programot különböző módon kell futtatni fejlesztői módban és felhasználói módban, máshogy kell felparaméterezni a java illetve a GWT fordítókat, ezért szükség volt ennek az egyszerűvé tételére. Ebben segít az Apache Ant, ami egy java alapú build-automatizáló eszköz.

### Resty

- Kis könyvtár, ami leegyszerűsíti a REST API-k hívását.

## Tiszta Kód

A szakdolgozat témájának leadása idején ismerkedtem meg a programozás egy merőben új szemléletével. Provokatívan megfogalmazva arról szól, hogy ha a kód amit írtam nem érthető, akkor rossz kódot írtam, még akkor is, ha az megoldja a kitűzött feladatot. A programkódot elsősorban más embernek írom, és csak másodsorban a számítógépnek (Martin, Clean Code, 2009, old.: 6-12). Így a szakdolgozatom elsősorban és leginkább ennek a módszernek a tanulása és elsajátítása, mint bármi más.

Ebben a fejezetben vannak azok a szabályok, amikkel a fejemben írtam a programot. A forráskód külalakja, módosíthatósága, modularizáltsága, struktúráltsága, öndokumentálásának érdekében tett lépéseket, és az ehhez tanult módszereket mutatom be. Nem sikerült mindenhol betartanom ezeket a módszereket, de mindenhol törekedtem rá, hogy a kódom minél olvashatóbb és könnyedén módosítható legyen.

Az itt felsorolt szabályok sajnos *nem azok*, amik igazak a program kódjára. Az itt felsorolt szabályok azok, amiket legjobb tudásom szerint próbáltam betartani a kód írása közben.

## Java konvenciók

Az osztályok neve nagybetűvel, a függvények neve kisbetűvel kezdődik. A konstans változók neve CSUPA\_NAGYBETŰ, és alulvonások választják el a szavakat. A getter és setter metódusok get illetve set szóval kezdődnek (McConnell, 2004, old.: 276).

## Változók

A változók nevei mindig tükrözik azt, amire használják őket, kerültem a rövidítéseket, és a kiejthetetlen neveket (Martin, Clean Code, 2009, old.: 18), (McConnell, 2004, old.: 260). Kerültem az objektumszintű változók használatát, és a lokális változók élettartamát is igyekeztem rövidíteni a lehetőségekhez képest (McConnell, 2004, old.: 246). Előfordulnak változók a kódban, amik csak azért jöttek létre, hogy javítsák az olvashatóságot, vagy hogy tisztázzák, egy függvény pontosan mit is adott éppen vissza.

## Függvények

A kódban előforduló függvények mellékhatás-mentesek, és úgy vannak elnevezve, hogy azt csinálják, amit a neve alapján az ember elvárna tőlük. A nevükben benne van minden, amit csinálnak (McConnell, 2004, old.: 171). Egy függvény csak egy dolgot csinál, és azt jól csinálja, és semmi mást nem csinál (Martin, Clean Code, 2009, old.: 35). A függvény nevének hosszúsága attól függ, hogy mekkora a láthatósága a függvénynek. A publikus metódusok neve rövid, tömör, és lényegre törő, az osztályon belüli privát függvények neve olyan hosszú, ami megfelelően leírja, hogy milyen feladatot végeznek el. A paraméterek száma lehetőleg ne legyen háromnál több.

Példa:

```
public Route getRoute(String startTown, String endTown)
{
    if (bothLocationExists(startTown, endTown)) {
        saveTownsToSession(startTown, endTown);
        return database.getRoute(formatted(startTown), formatted(endTown));
    } else {
        return null;
    }
}
```

## Osztályok

Az osztályoknak a legfontosabb feladata, hogy csökkentsék a program bonyolultságát (McConnell, 2004, old.: 152). Mivel a programozó legfontosabb feladata úrrá lenni a káoszon (Fóthi Ákos előadásán hangzott el 2009-ben), ezek a leghatékonyabb eszközök a programozó eszközei közül. A szabályok amiket be próbálok tartani az osztályok készítésekor, segít a kód struktúráltságának alakításában, és abban, hogy egyszerre minél kevesebb absztrakciós színen kelljen gondolkodni. A szabályok a következők: Egy osztály legyen erősen kohézív, és az osztályok egymással legyenek lazán csatoltak. Minden fölösleges információt rejtse el, és jól definiált interfészen keresztül lehessen őket használni. Zárják el a komplexitást, és rejtse el az implementáció részleteit. Rejtse el a privát adataikat, és egy osztálynak legyen pontosan egy feladata.

## Adatbázis

Az adatbázis kezelése legyen egy feladat, és legyen egy központi helyen. Így a kód többi részének nem kell foglalkoznia az adatbázissal kapcsolatos alacsony absztrakciós szintű feladatokkal, így redukálva a kód komplexitását (McConnell, 2004, old.: 85). Az adatbázis egy interfészen keresztül érhető el, ami lehetővé teszi a módosítást, sőt az implementáció teljes lecserélését úgy, hogy sehol máshol nem kell módosítani a kódban.

## Demeter törvénye(Law of Demeter)

Ez a szabály röviden arról szól, hogy egy osztálynak nem szabad tudnia azoknak az osztályoknak a belső felépítéséről, amiket használ (Martin, Clean Code, 2009, old.: 97).

Például:

**Helyes:**

```
class A{  
    B b = new B();  
  
    void foo(){  
        b.hello();  
    }  
}
```

**Rossz:**

```
class A{  
    B b = new B();  
    void foo(){  
        b.publicFieldOfB.hello();  
        b.getPrivateFieldOfB().hello();  
    }  
}
```

Ezzel a módszerrel el lehet kerülni az úgynevezett „vonatszerencsétlenségeket”, amikor a függvényhívások feltorlódnak egy sorba. Ha ilyen függvényhívás fordul elő a kódban, az azt jelenti, hogy valamilyen funkcionalitást kifejeztünk a használt osztályból, vagy rossz osztálynak adjuk a feladatokat. Külső osztályok használata

mellett ezt a szabályt szinte lehetetlen betartani, de nagyon jól csökkenti a komplexitást a kódban, és kikényszeríti az osztályok jó használatát.

### „Extract Till You Drop” - szabály

Milyen hosszú legyen egy függvény? Ezt a kérdést válaszolja meg egyértelműen ez a szabály. Egy függvény akkor végzi jól a dolgát, ha egy dolgot csinál, csak azt csinálja, és azt jól csinálja. Ha egy függvény magyarázatakor használok az „és” kötőszót, ha tudok két különböző dolgot mondani ami a függvényen belül történik, akkor az a függvény nem egy dolgot csinál. Ekkor azt a függvényt szét kell bontani több függvényre, egészen addig, amíg egy függvény már csak egy dolgot csinál. Ez a szabály segíti a függvények jó elnevezését, és hogy egy függvény csak egy absztrakciós szinttel foglalkozzon a kódon belül (Martin, Clean Coder Code-casts, 2011).

### Kommentek

A kommentek hajlamosak nagyon gyorsan elöregedni, téves információt adni, hazudni. Minden kommentnek egy elismerésnek kell lennie, hogy hibáztunk. Abban hibáztunk, hogy nem sikerült a kódban jól kifejezni magunkat. Ha kommentet írunk valamihez, akkor annak adnia kell plusz információt a kódhoz, nem pedig ismételnie azt. Természetesen a komment itt nem egyezik meg a kód dokumentációjával, és van ahol szükséges sőt kötelező kommentet írni. Az általános szabály, hogy a kommentek legyenek ritkák, és amikor mégis írunk egyet, az adjon hasznos információt (Martin, Clean Code, 2009, old.: 53-74).

### Az indítás és futtatás elválasztása

A kódban külön kell választani a program elindítását, inicializálását, és a futó program kódját. Így jobban struktúrált és átlátható lesz a program. A másik igen lényeges elem, amit el kell választani a többitől, az a program fő osztálya, ahol a *main* található. Mivel a többi objektumot itt hozzuk létre, a programkódot a *main* metódusban már csak a létrehozott elemekből kell összeállítani, és ez a programfelépítés nagyban elősegíti a kód modularizáltságát (Martin, Clean Code, 2009, old.: 154).

Mindkettőre jó példa a szakdolgozat kódja, és részletes magyarázattal példa látható rá a dokumentációban is, a Fejlesztői Dokumentáció fejezetben, a kliens csomagjának leírásában.

## Refaktorálás

A fentebb leírt szabályokat követve próbáltam elkészíteni a kódot, de természetesen ezeket a dolgokat nem sikerült elsőre megcsinálni. Sokszor voltam kénytelen a kódomat újraírni, átstrukturálni. Ennek a módszernek a neve a refaktorálás. A refaktorálás definíciója a következő:

*Megváltoztatjuk a program belső működését illetve struktúráját a azért, hogy érthetőbbé és modulárisabbá tegyük, úgy, hogy közben a kód viselkedését nem változtatjuk meg (Flower, Refactoring, 2008, old.: 53).*

Írás közben nehéz eldönteni, hogy mikor jár a kód struktúrája jó úton, és gyakran eltéved. A legtöbbször egy nem látott probléma, vagy egy osztály ami egyre több feladatot lát el, túl nagyra nő. Ekkor kell a kódot refaktorálni, amíg ismét a kívánt minőségű nem lesz. Ezeknek a problémáknak rengeteg jele van. Ezek között szerepelnek a hosszú függvények, a túl nagy vagy sok mezővel rendelkező osztályok, a több helyen előforduló duplikált kód, a kommentek, sőt, még az olyan programozási elemek is mint a switch (Flower, Refactoring, 2008, old.: 75-87).

Ezeknek a problémáknak a felismerésével és megoldásával valamint a tervezési minták követésével lehet egységes felépítésű, átlátható, könnyen bővíthető kódot írni.

## Az úttörők szabálya(Boy Scout rule)

Ez az egyszerű szabály eredetileg tényleg az úttörőknek szólt, és így hangzott: „Hagyd a táborhelyet tisztábban ott, mint ahogy találtad.” Ezt a szabályt a programozásra is ki lehet terjeszteni (Martin, Clean Code, 2009, old.: 14). Mindig, miközben a kód más részét szerkesztve találkoztam valamilyen nem megfelelő elnevezéssel, kijavítottam. Ha megláttam valamit szerkezetbeli problémát, vagy egy lehetőséget az egyszerűsítésre, megtettem. Így a kód minősége minden alkalommal - még a dokumentáció e részének írása közben is, mikor a példákat kerestem a szabályokhoz – emelkedett.

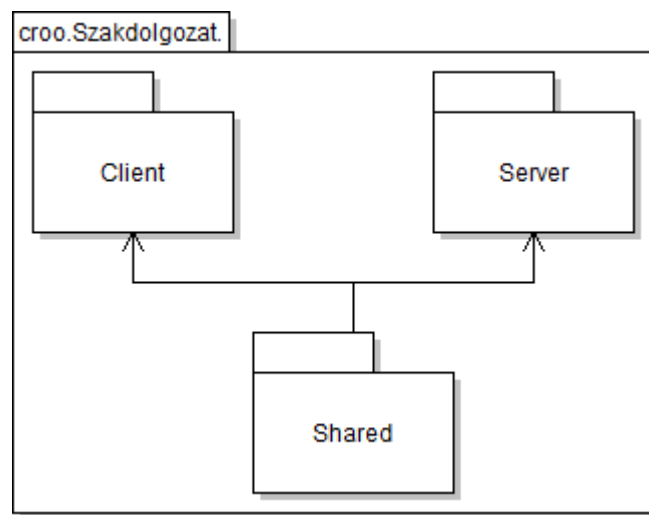


## Rendszerarchitektúra

A rendszer három nagy részre, Java csomagokra tagolódik:

- Client
- Server
- Shared

A Client és a Server értelemszerűen a kliens illetve a szerver kódját tartalmazó csomagok. A Shared csomagban azok az adatstruktúrák vannak, amik a hálózaton utaznak a kliens és a szerver között.



9. Magasszintű csomagdiagram

Az egyes csomagok szerkezeti felépítésének megértéséhez szükség van két tervezési minta elsajátítására. A program rendelkezik egy *Main.gwt.xml* nevű projektfájllal, ami a GWT keretrendszer számára szükséges információkat tartalmazza: az általánosan használandó stíluslapot, az alkönyvtár helyét amiből a Javascript kód készül, valamint a kliensben használt külső modulok(pl GoogleMaps API) helyét.

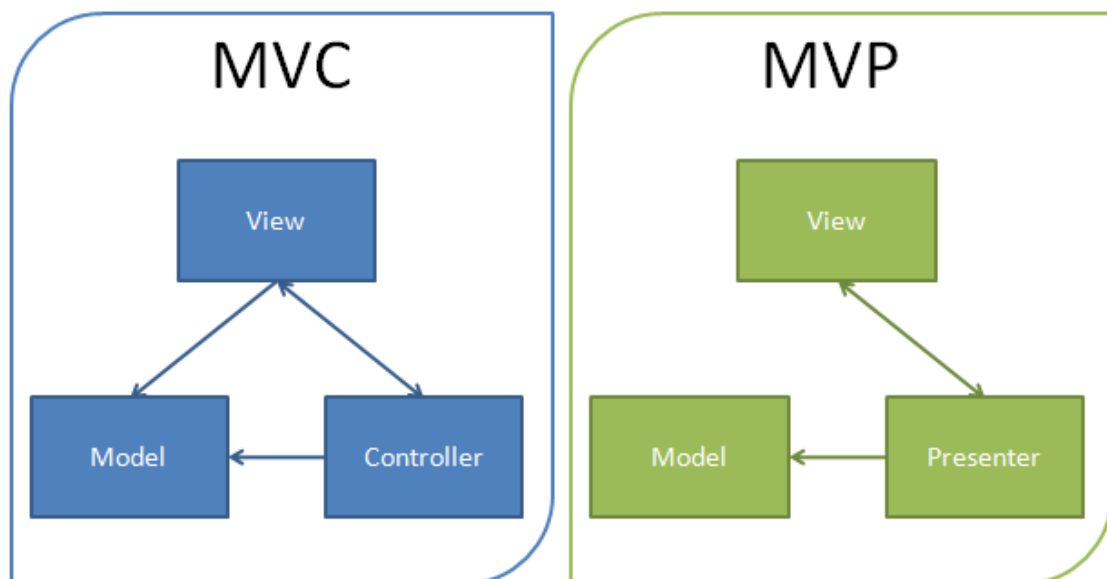
### Programtervezési minták a kódban

Két programtervezési mintát használtam a program megépítése során. Az első és legfontosabb az MVP tervezési minta. Ez a minta határozta meg a kód szerkezetét a kliensben. Megértése elengedhetetlen a kód struktúráltságának megértéséhez. A másik minta a Singleton tervezési minta, aminek segítségével elérhetővé tettem a program paraméter-fájl értékeinek lekérdezését az egész kliensben.

### *Model-View-Presenter(MVP) Pattern*

Az MVP pattern (Rask, 2008), (Flower, GUI Architectures, 2006), azaz programtervezési minta a GWT kliens fejlesztésében kap nagy szerepet. Feladata, hogy elválassza a grafikus felületet a logikától és az adatszerkezetektől. A View részt az alkalmazásban a view csomagban található xml felületleíró fájlok és a hozzájuk tartozó java osztályok alkotják. A view-ben nem található logika, minden olyan feladatot, ami nem a kinézettel kapcsolatos, a saját presenterének delegál. A presenter kommunikálhat a view-el egy display interfészen keresztül, ami biztosítja, hogy a grafikai felület teljesen elváljon az alkalmazás többi részétől, cserélhető és moduláris legyen. A presenter intézi a hívásokat az alsóbb rétegekbe, azaz a model-be, illetve kapja a válaszokat, amiket szükség esetén delegál a view-hez. A modelben az alkalmazáshoz tartozó erőforrásokat és adatokat érhetjük el, ami esetünkben a szerver interfészeit jelenti.

A jól ismert Model View Control(MVC) mintához nagyon hasonló. Eltérésüket az alábbi ábra szemlélteti:



10. MVC vs. MVP

Az MVP programtervezési minta nagyobb alkalmazásokban a struktúráltság mellett a tesztelést segíti elő; a GWT által generált javascript tesztelése ugyanis nehézkes, és a felületi elemek elválasztása lehetővé teszi, hogy a minden szükséges logikát javascriptre fordítás nélkül lehessen letesztelni. A kliens kód szerkezete ezen minta

köré lett felépítve. Ugyan a gyakorlatban a grafikát és a logikát nem mindig triviális elválasztani, ez a minta a legtöbb helyen jól biztosítja az olvashatóságot és a rétegek elkülönülését.

A dokumentáció következő szakaszaiban gyakran előfordul a *modul* kifejezés használata. A kliens egy *modulja* az elkövetkezendő leírásokban egy view és a hozzá tartozó presenter párost jelent.

### **Singleton Pattern**

A singleton tervezési minta (Wikipedia, 2012) biztosítja, hogy egy objektumból legfeljebb egy létezzen az egész kódunkban, ezzel biztosítva egy globális változót az objektumorientált nyelvekben. Használata ugyanúgy kerülendő, mint a globális változóé (Hevery, 2008), de természetesen itt is vannak olyan helyzetek, amikor alkalmazásuk hasznos és ajánlott.

A kódban egy singleton található, aminek neve SystemProperties. Ez az osztály olvassa be a kulcs-érték párokat tartalmazó properties fájlt ami felparaméterezi az alkalmazást.

### **Ant script**

A programhoz tartozik egy *build.xml* Ant<sup>11</sup> script. Ebben különböző parancsok segítik a program fordítását és egyéb feladatokat.

Az ant script fontos *target*jei, azaz utasításai:

#### build

- A kódot fordítja le a gwtc fordító segítségével Javascript, illetve a javac fordító segítségével bytekódra. A gwtc és a javac targeteket használja.

#### clean

- A generált kódokat, könyvtárakat, lefordított osztályokat, forrásokat törli ki, kitisztítva a projekt szerkezetet.

#### devmode

---

<sup>11</sup> Apache Ant: Build automatizáló eszköz, XML nyelven. Hasonló egy C++ make fájlhoz, de annál sokkal robosztusabb.

- Az egyik leggyakrabban használt parancs, ezzel lehet a GWT keretrendszer fejlesztői módját elindítani. Egy Jetty szerveret indít el, ami futtatja a szerveroldali kódot, figyeli a hozzá kapcsolódó klienseket, és kiírja a velük kapcsolatos logokat/hibákat.

#### deploy

- A build.xml fájlban beállított glassfish szerverre telepíti az alkalmazást. Csak helyi glassfish-el működik, elsősorban helyi tesztelésre jó.

#### war

- Újrafordítja az alkalmazást és elkészíti belőle az alkalmazásszerverre telepíthető .war fájlt.

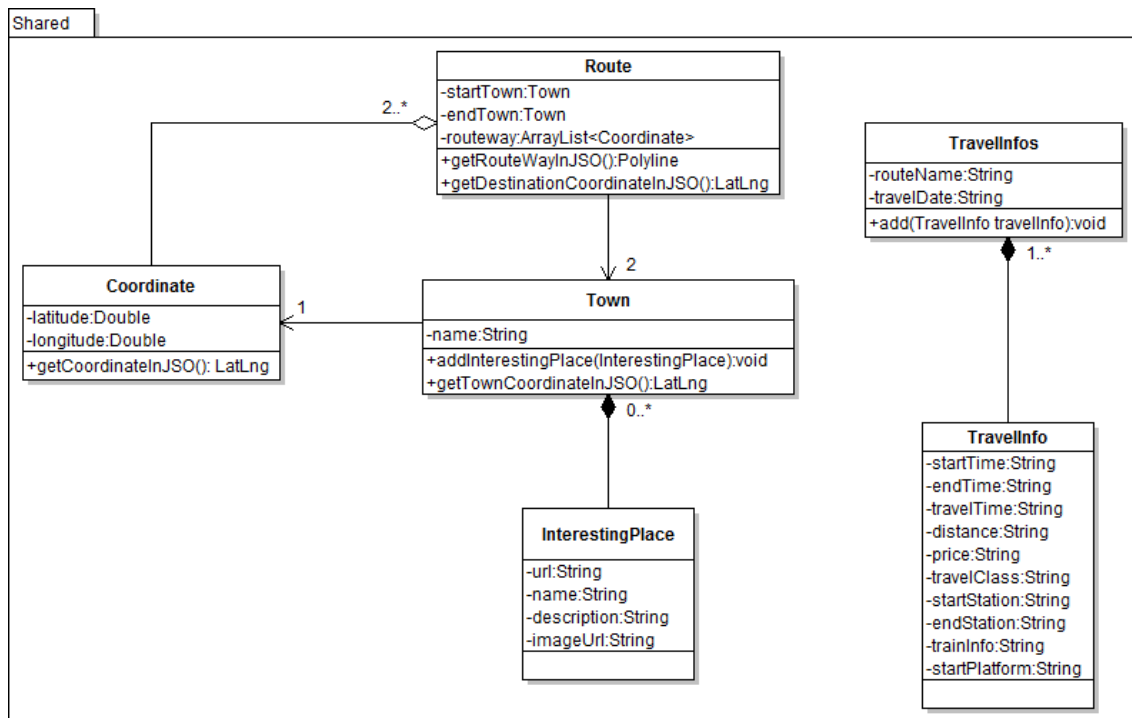
#### libs

- Az war fájl elkészítéséhez és működéséhez szükséges külső könyvtárakat másolja a build.xml-ben megadott helyekről a projekt megfelelő helyére.

## Csomag, Modul és Osztályszerkezet

### *Shared csomag*

Ez a csomag tartalmazza azokat az adatstrutúrákat, amik a szerver és a kliens között utaznak a hálózaton. Mindegyik osztály szerializálható, mivel örökli a GWT által biztosított `IsSerializable` interfészt. A feltüntetett függvényeken kívül gettereket illetve settereket tartalmaznak.



### 11. Shared csomag osztálydiagram

A TravellInfos adatstruktúra tartalmazza az Elvira REST API-tól lekérdezett táblázatot, valamint az utazás nevét és dátumát. A TravellInfo osztály ennek a táblázatnak egy sorát reprezentálja.

Az ábrán négy osztálynál is előfordul a `getCoordinateInJSO` függvény valamilyen formában. A JSO rövidítés azt jelenti, hogy JavaScriptObject. A GoogleMaps API koordináta-osztálya a natív Javascript kódban megírt `LatLng` osztály, és ezek a függvények egy ilyen objektummá konvertálva adják vissza az adatot. Így megjelenítésnél sokkal kényelmesebb használni az adatstruktúrákat.

### Client csomag

Ebben a csomagban található az a kód, ami a kliens oldalon, böngészőben fut. Csak a szerverrel áll kapcsolatban, az RPC által biztosított aszinkron hívásokon keresztül kommunikál vele. Ezeknek az interfészeknek a leírása a Server csomag fejezetben kerül részletesebb kifejtésre.

### Main Entry Point

A Main osztály `onModuleLoad` függvénye a program induló függvénye. Itt hozom létre a dinamikus elemeket, az RPC interfészeket és helyezem el őket a honlap statikus

felületén. Itt a `GWT.create(..)` parancs körülbelül megfelel egy `new` parancsnak. Ezt a létrehozási technológiát a GWT keretrendszere biztosítja, és Deferred Binding módszernek hívják. Az ezzel a parancsal létrehozott objektumokhoz készít a GWT fordító javascript kód permutációkat a különböző böngészőkhöz, illetve ez ugyanez a parancs hozza létre a generált osztályokat az RPC kommunikációs interfészhez fordításkor.

```
public class Main implements EntryPoint
{
    @Override
    public void onModuleLoad()
    {
        EventBus eventBus = GWT.create(SimpleEventBus.class);

        MapServiceAsync mapService = GWT.create(MapService.class);
        FilterServiceAsync filteringService = GWT.create(FilterService.class);
        TravelServiceAsync travelService = GWT.create(TravelService.class);

        TravelMapView map = new
            TravelMapView(eventBus, mapService, filteringService);
        TravelInfoView travelInfo = new TravelInfoView(eventBus, travelService);

        RootPanel.get("map").add(map);
        RootPanel.get("travelinfo").add(travelInfo);
    }
}
```

Az alsó két sor pedig a létrehozott dinamikus View elemeket rakja rá a statikus HTML oldalra, hozzárendelve az elemeket az HTML kódban megfelelő ID-vel ellátott div tagokhoz.

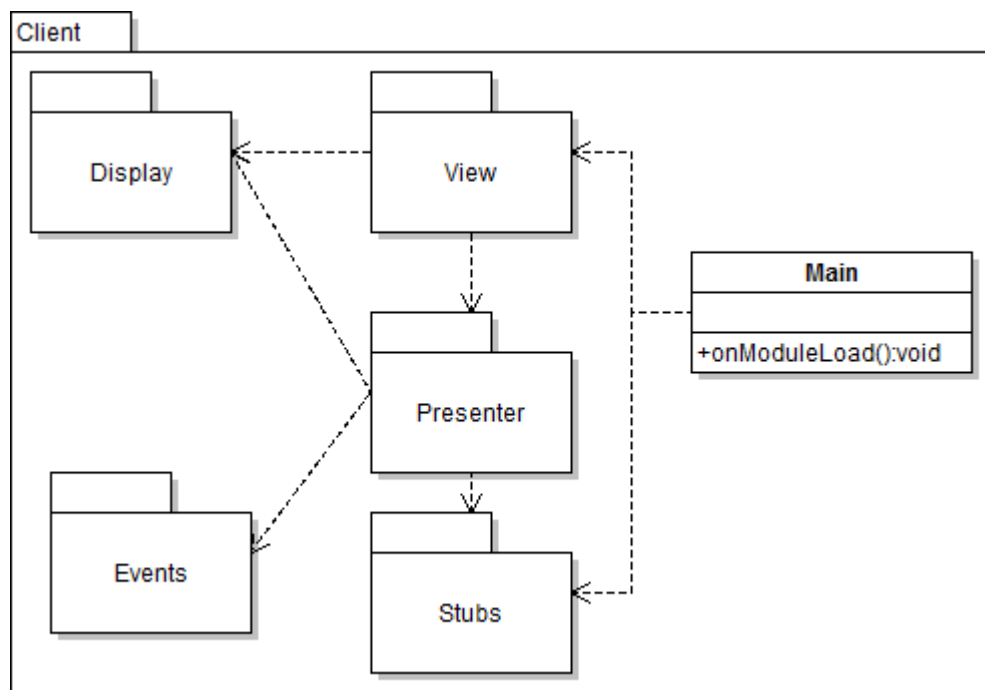
Ez a kód nagyszerű példa arra módszertanra, hogy a program inicializálása és a futása elkülönüljenek egymástól, valamint látszik belőle, hogy a program szükség esetén könnyedén bővíthető további funkciókkal, és modularizált (Martin, Clean Code, 2009, old.: 154).

## WEB-INF Mappa

A WEB-INF mappa tartalmazza azokat a lefordított class fájlokat, generált javascript kódot, képeket és egyéb forrásokat, amikből a war fájl készül. Két fontos kliens oldali fájl van benne amit szerkeszteni is kell. Ezek a honlap statikus részét írják le, és a mappa gyökerkönyvtárában találhatóak: A *Main.html* és a hozzá tartozó *Main .css* stíluslap. A két elem nem a HTML kódban, hanem a GWT projektfájlban van összedrótva.

## Felsőszintű csomagdiagram

A kliens sok kisebb csomagból épül fel, jól magyarázva kliens különböző részeit:

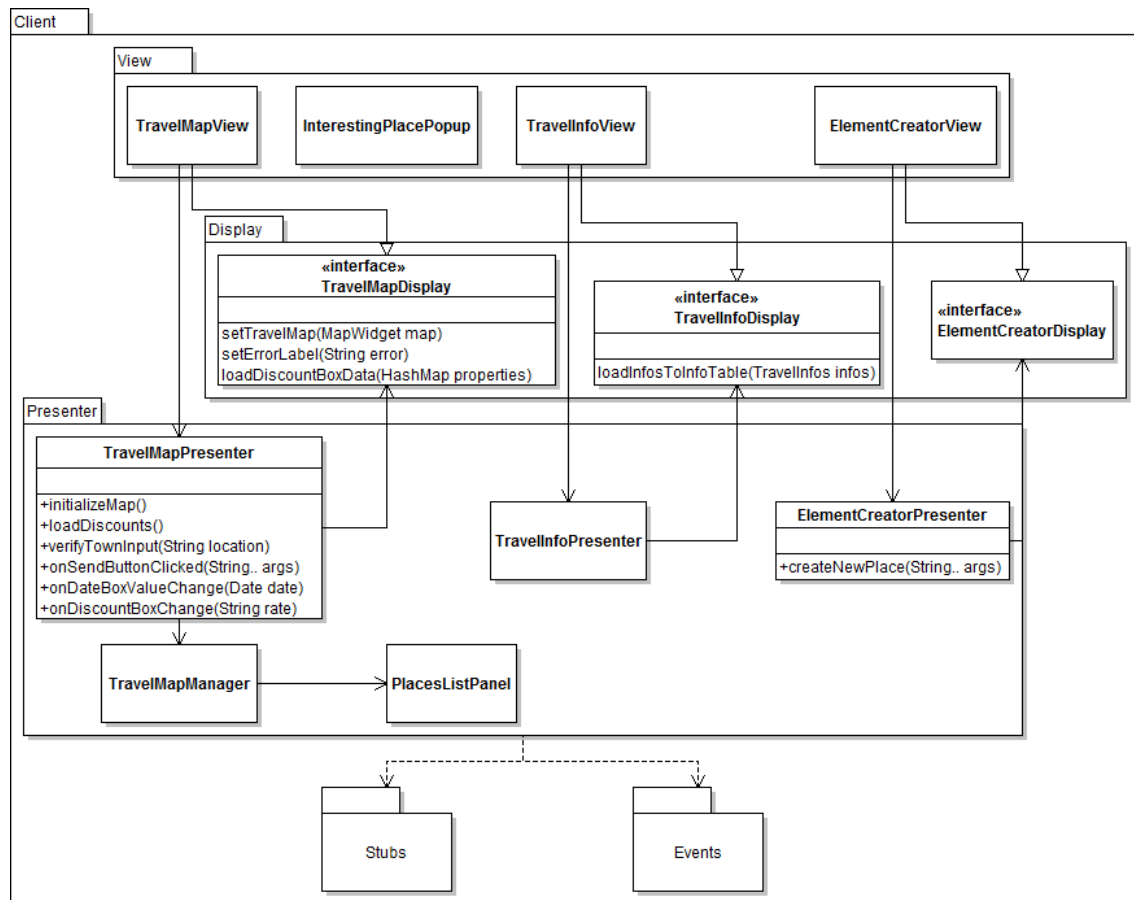


12. Kliens Csomagdiagram

A Stubs csomag tartalmazza a szerverrel kapcsolatos kommunikációs interfészeket, az Events csomag pedig az MVP egyes view-presenter párosai közötti kommunikáció eseményeit és eseménykezelőit tartalmazza.

A Display, View és Presenter osztályok az MVP minta részei, ezekről a tervezési minta leírásánál, valamint a kliens osztálydiagramjánál vannak részletezve.

Az ábrán jól láthatóan különül el a kód az MVP minta szerint:



### 13. Kliens osztálydiagram

A Modellt, az MVP harmadik rétegét itt elsősorban a szerverrel kommunikálás, a Stubs csomag alkotja. Ennek leírása részletesen a következő fejezetben, a „Server csomag és az RPC kommunikáció”-nál található meg.

A View csomagban lévő kódok tartalmazzák az oldalon megjelenő dinamikus elemeket. Mindegyik View a GWT keretrendszer UiBinder technológiája segítségével lett összeállítva. Ez azt jelenti, hogy a grafikus elemek egy XML plusz Java osztály párosok, amik együtt írják le a kinézetét az elemnek és kezelik az eseményeit a bennük lévő elemeknek. A View osztályai a megjelenítési logikán és az esemény továbbításon kívül semmilyen logikát nem tartalmaznak, minden feladatot az alattuk lévő csomagban a hozzájuk tartozó Presenternek adnak át.

A Presenter végzi a logikai feladatokat, ellenőrzi a bemenetelt, szólít meg más modulokat a kliensen belül( Events csomag) és kommunikál a szerverrel(Stubs



csomag). A hozzá tartozó View-et egy interfészen keresztül éri el, ez látható a Display csomagban. Ezek az interfészek szeparálják el teljesen a logikát a grafikus felülettől a kliensben, úgy, hogy laza kapcsolattá avanszálják a View és a Presenter közötti kapcsolatot. Ez a program javíthatóságán és modularitásán javít. Az osztálydiagramon feltüntetett függvények a Presenter és a View közötti kommunikáció interfészei, egyben az osztály publikus függvényei is.

### *Eseménykezelés modulok között*

Az egyes modulok közötti eseménykezelést a GWT egy speciális osztálya, az EventBus kezeli. Ez egy osztály, amire esemény-objektumokat lehet küldeni, illetve feliratkozni egyes eseményekre. Az események megírt osztályok példányai, így bármilyen adatot képesek tárolni. A modulok létrehozásakor megkapják az EventBus objektumot, és az Events csomagban lévő események használatával tudnak üzenetet küldeni egymásnak.

### *ErrorHandlingAsyncCallback osztály*

A kliens aszinkron hívásokkal hívja a szerveret, ahonnan néha hibaüzenetek jöhetnek, ha nem elérhető a szerver, vagy ha a szerver belső hibába ütközik. Ekkor a hibaüzenetek kezelését ez az osztály végzi.

```
public abstract class ErrorHandlingAsyncCallback<T> implements
AsyncCallback<T>
{
    PopupPanel popup = new PopupPanel(true, true);

    @Override
    public void onFailure(Throwable caught)
    {
        popup.clear();
        popup.add(new HTML(caught.getMessage()));
        popup.show();
    }
    @Override
    public abstract void onSuccess(T result);
}
```

Az eredeti AsyncCallback interfész két függvényét kell implementálni minden szerverhíváskor, az onSuccess, illetve az onFailure metódust. Az első akkor hívódik

meg, ha az eredmény sikeresen megjött a szervertől, az `onFailure` pedig akkor, ha valamilyen hiba vagy http error jön vissza eredményül, azaz nem jött használható eredmény. Az `ErrorHandlingAsyncCallback` absztrakt osztály az `onFailure` metódust implementálja, és egy popup ablakban értesíti a klienst a felmerült hibáról. Így a hibakezelés egy helyen van, és tényleges szerverhívásokkor csak az sikeres működést kell implementálni. Amilyen egyszerű ez az osztály, annyira hasznos. Rengeteg kódismétléstől szabadít meg, és szervertől jövő hibakezelést gyakorlatilag ő végzi.

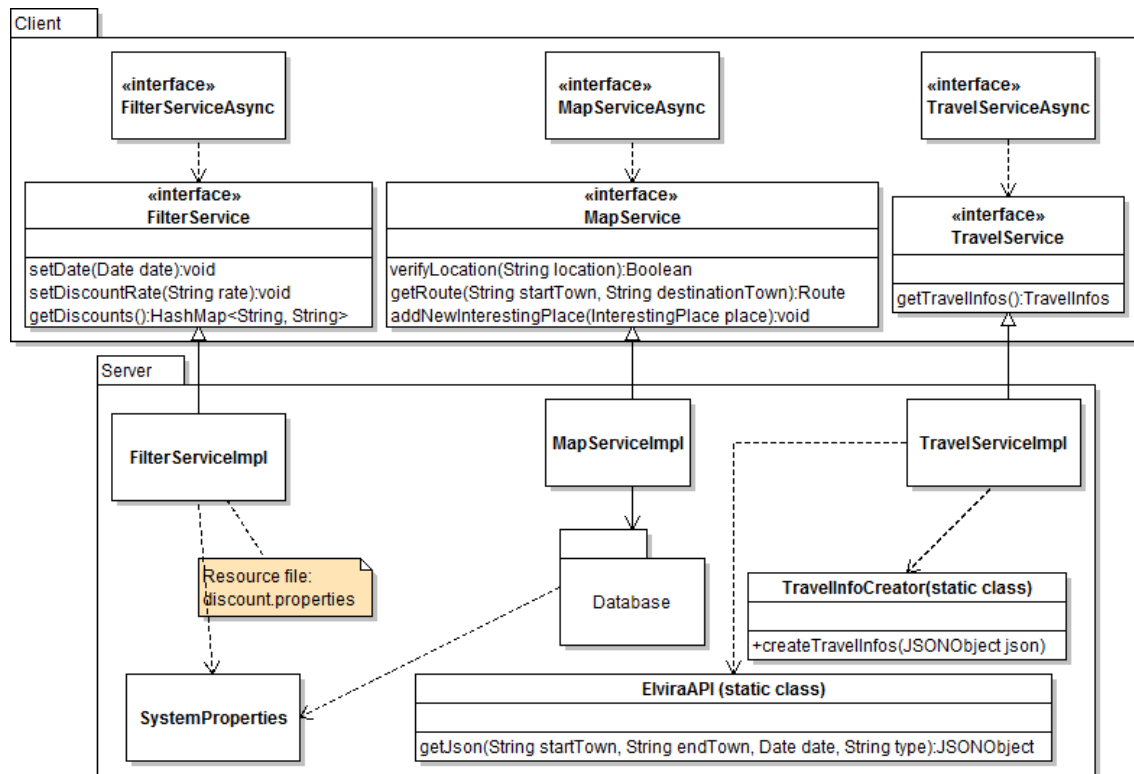
Mivel a szerverből jövő hibaüzenetek esetén a kliens nem sokat tehet, így ennél részletesebb hibakezelés implementálása nem szükséges.

### *Server csomag és az RPC kommunikáció*

A szerver három interfészen keresztül kommunikál a külvilággal. Adatokat biztosít a kliensnek, adatokat ír és olvas az adatbázisból, valamint egy távoli REST interfészen keresztül kérdez le adatokat.

Ebből Sever csomagban a klienssel kommunikáló szervletek vannak. Ezek a szervletek a `FilteringService`, a `MapService` és a `TravelService` implementációi.

Ezeknek az osztályoknak az interfészei, ún. stub-jai a kliens csomagban vannak, egy hozzájuk tartozó aszinkron interfész párral együtt. Ez a hármas: a kliensben lévő interfész és aszinkron párja, valamint a szerverben lévő implementáció együtt valósítja meg az RPC hívásokat a szerver és a kliens között. A kliensben a GWT keretrendszer segítségével példányosítani lehet az aszinkron interfészeket, amikhez fordításkor legenerálódik a kommunikációhoz szükséges kód.



14. Server osztálydiagram és RPC mechanizmus

A három szervlet három különböző feladatot lát el.

#### ✚ FilterServiceImpl

- Elmenti a szűrőként beállított adatokat a felhasználóhoz kötött session-ban, valamint biztosítja a szűrés megjelenítéséhez szükséges adatokat a kliensnek.

#### ✚ MapServiceImpl

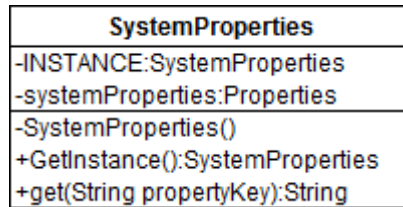
- A térképes műveletek futnak be hozzá, ő szerkeszti és olvassa az RDF adatbázist

#### ✚ TravelServiceImpl

- Ő kommunikál a külső REST API-val, kéri le az adatokat a szűrés alapján, és állítja őket össze RPC-n átküldhető adatstruktúrává, segédosztályok segítségével.

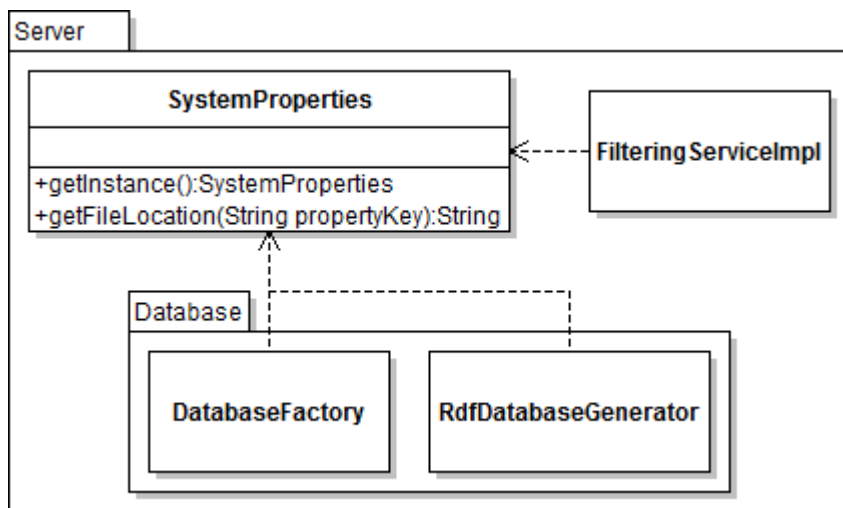
## SystemProperties

A szerver csomagban található a SystemProperties singleton osztály is. Ez az osztály a szerver konfigurációs fájljában megadott paraméterek lekérdezését teszi lehetővé, hasonlóan a Java System.getProperties() statikus függvényéhez hasonlóan.



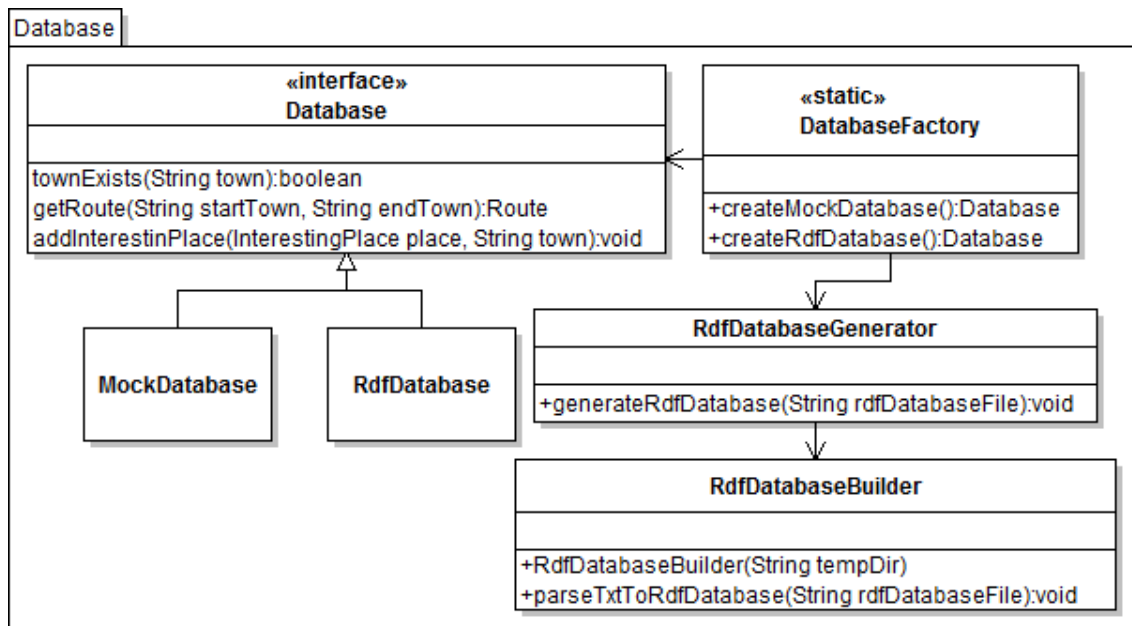
15. SystemProperties singleton

A konfigurációs fájl felhasználói pedig a szerver oldali osztályok. A Filtering szervlet, a Database csomagból pedig a DatabaseFactory és az RdfDatabaseGenerator osztályok használják.



## Adatbázis

A program adatait egy RDF adatbázisban tárolja. Az adatbázis egy rdf kiterjesztésű fájlban van tárolva egy tetszőleges, a szerver konfigurációs fájljában megadott helyen. Az adatbázis műveletek a szerveren a Database csomagban találhatóak. Ennek a csomagnak a felépítését a következő ábra szemlélteti:



16. Adatbázis osztálydiagram

A Database interfész definiálja az adatbázis szükséges műveleteit, amit egy teszteléshez létrehozott MockDatabase a valós adatbázist kezelő RdfDatabase osztály implementál.

A DatabaseFactory egy adatbázisgyártó osztály, amire azért van szükség, hogy az rdf adatbázis esetleges létrehozásának feladatát elvégezze. Az ezt használó MapService szervlet ennek a Factory osztálynak a segítségével gyártja le a használni kívánt adatbázist.

A Factory megvizsgálja, hogy létezik-e a konfigurációs fájlban megadott adatbázis, és ha nem létrehozza azt az RdfDatabaseGenerator és az RdfDatabaseBuilder segítségével. Előbbinek feladata a forrásadatok zip-fájljának beszerzése, a kitömörítés, az adatok RDF-é alakítása, és a feltakarítás. Ebből a feladatból az adatok RDF-é alakítása elég nagy falat, ezért ezt a feladatot delegálja az Builder osztálynak.

### Adatbázis létrehozása

A szerver elindításakor, a MapService servlet osztály inicializálásakor indul el az adatbázis generálás folyamata, amennyiben nem létezik a megadott adatbázis fájl. Ennek az adatbázisnak a létrehozása a forrásadatok méretétől és a szerver kapacitásától függően 10 perc is lehet, és program memóriaigénye is jelentősen megnőhet. Ezalatt a szerver letölti az előre megadott helyről azt a zip fájlt, amiben

található fájlok megfelelnek a GTFS (General Transit Feed Specification) specifikációnak, kitömöríti és beolvassa a benne található csv fájlokat, és felépíti belőlük az rdf adatbázist amelyben szerepelnek a megállók/városok, a közöttük menő útvonalak, valamint minden városhoz egy üres lista, ami annak a helynek a látnivalóit fogja tartalmazni. Ezután a zip-fájlt és az ideiglenes fájlokat törli.


### *Szerkezeti felépítés*

Az RDF adatbázis hármasokból áll, amit *statementnek* hívunk. A hármas két elem, és a közöttük lévő relációból áll. A *subject* az első eleme a hármasnak, ami relációban áll a hármas harmadik elemével, az *objectel*. A köztük lévő reláció a hármas középső tagja, ezt *predicatenek* nevezzük. Ezekből a relációkból sok előre definiált létezik, amiknek használata ajánlott, mert így egy reláción mindenki ugyanazt fogja érteni. A relációkat az RDFben felsorolt *névterek* azonosítanak egyértelműen. Például a Város – Neve – Budapest hármasban a Neve reláció már definiálva van egy sok helyen használt *névtérben*, az [xmlns.com/foaf/](http://xmlns.com/foaf/) oldalán, így ha ezzel együtt használjuk, egyértelmű lesz a reláció jelentése.


### *RDF névterek*

névterek és azok elemei segítségével írható le az egyes elemek közötti reláció.

Az adatbázis a következő információkat tartalmazza:

 **rdf**=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

- Az rdf adatbázis struktúrájában használt elemek névtere.(például **ID,about**) Minden RDF adatbázisban megtalálható.

 **dc**=<http://purl.org/dc/terms/>

- A Dublin Core névtere. A honlap(**references**) és a leírás(**description**) relációkat használja belőle az adatbázis.

+ **geo**=[http://www.w3.org/2003/01/geo/wgs84\\_pos#](http://www.w3.org/2003/01/geo/wgs84_pos#)

- A WGS84<sup>12</sup> specifikációnak megfelelően földrajzi helyek leírására alkalmas névtér. A szélesség(**lat**) és a hosszúság(**long**) koordináták leírását használja belőle az adatbázis.

+ **georss**=<http://www.georss.org/georss>

- Földrajzi koordináták, útvonalak tárolására alkalmas névtér. Az adatbázisban az útvonal leírására(**line**) van használva.

+ **foaf**=<http://xmlns.com/foaf/0.1/>

- foaf azaz „friend of a friend” nevű névtér, emberi kapcsolatok leírására. A nevek(**name**) és kép-urlek(**img**) tárolására.

+ **croo**=<http://example.org/croo#>

- Ez az adatbázis saját névtere amiben az adatbázisra egyedi elemek találhatóak. Azaz a városok(**town**) az útvonalak(**route**) és az érdekes helyek(**place**).

### *Az adatbázis elemei*

+ **Város(Town):**

- A város neve
- A város pontos helyrajzi koordinátája
- Latitude
- Longitude
- A városban található érdekes helyek listája

+ **Érdekes Hely(Place):**

- A hely neve
- A hely leírása
- A hely honlapja
- Egy, a helyhez kapcsolódó kép netes linkje

+ **Útvonal(Route):**

- Az útvonal kezdőállomása

---

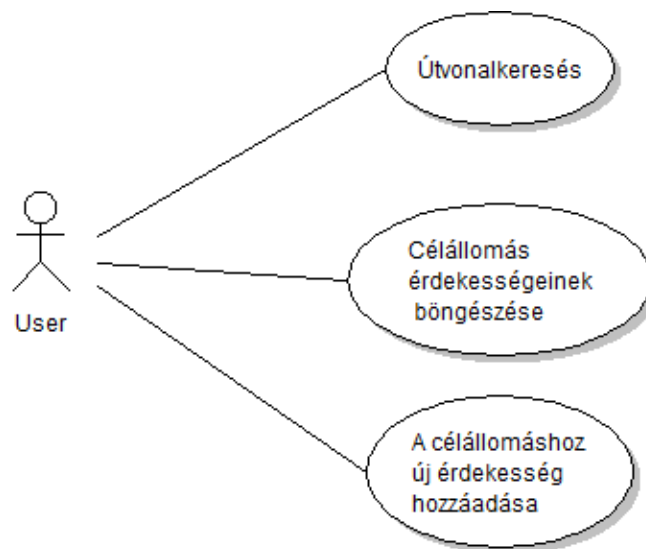
<sup>12</sup> WGS84 – Földrajzi fogalom: vonatkoztatási rendszer, a Földet globálisan közelítő ellipszoid-modell.

- Az útvonal végállomása
- Az útvonalat leíró koordináták listája

Ezek a köztük lévő relációk alapján egy gráfot alkotnak( függelék, 1. ábra).

## Felhasználói esetek

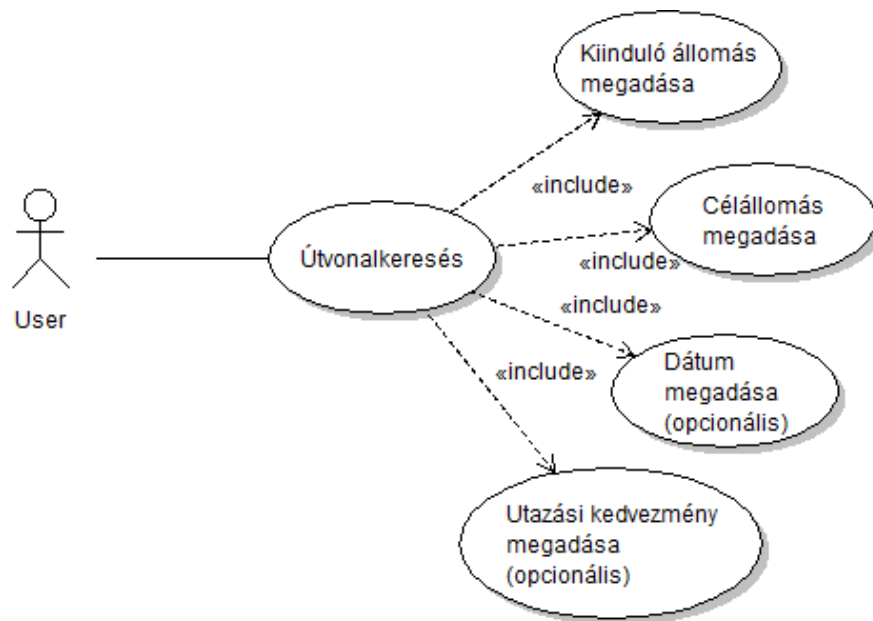
A felhasználói felület külleme és leírása a felhasználói dokumentumban található. Ez a fejezet a felhasználói eseteket veszi sorra. A felhasználó alapvetően három interaktív dolgot csinálhat a honlapon. Útvonalat keresni, a célállomáson lévő helyek listáját böngészni, illetve egy új helyet hozzáadni ehhez a listához.



### 17. Felhasználói esetek

Ebből az útvonalkeresés részletesen úgy történik, hogy a felhasználó kitölti a kereséshez szükséges mezőket. Az „honnán” és „hova” mezőket kötelező kitölteni, különben a keresés nem lehetséges. A dátum és a kedvezmények kiválasztása nem kötelező: ebben az esetben aznapi dátummal és kedvezmény nélkül, teljes árú vonatjeggyel érkeznek meg az adatok az oldalra.



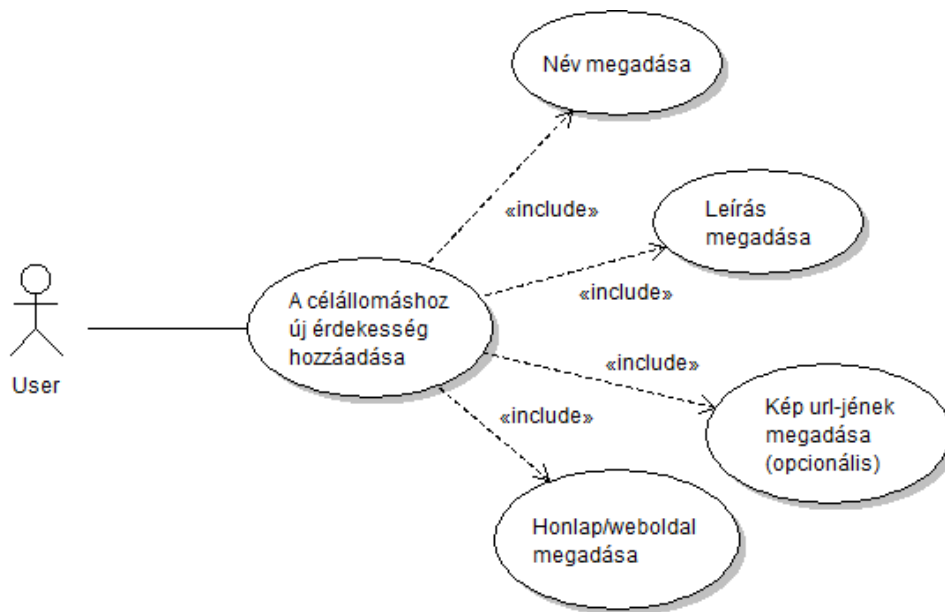


#### 18. Útvonalkeresés részletei

Ezután a keresés elindításával lehet a folyamatot elindítani. Sikeres útvonalkeresés után a térképen megjelenik a vonat által bejárt út. Ennek kezdete a kiinduló állomás, a jelölővel megjelölt vége pedig a célállomás.

A második, az „célállomások böngészése” felhasználói esetben a felhasználó a térképen a sikeres útvonalkeresés után a térképen felbukkanó jelölőre kattintva nyithatja meg annak a városnak a listáját, amin a jelölő áll. A felugró ablakban a listában szereplő elemekre kattintva nyithatja meg az egyes helykről szóló rövid leírásokat, valamint a leírásban szereplő linkre kattintva az adott érdekesség honlapját.

Ennek a listának a legalsó kék „Hozzáadás...” elemével lehet új érdekes helyet hozzáadni a listához. Ekkor ki kell tölteni a hely nevét, rövid leírását, az érdekesség honlapját, valamint opcionálisan egy neten található kép webcímét megadni.



19. Hely hozzáadásának részletei

Amennyiben ez a lista üresen érkezik a klienshez, akkor a hozzáadási funkción kívül csak egy felirat látszik, közölve a felhasználóval, hogy „Itt nincs semmi érdekes...”.

## Tesztelés

A legelémibb tesztelési fajta a manuális tesztelés. A fejlesztés során természetesen ez a fajta tesztelés dominált, mert a böngészőn a felhasználó imitálása, automatizálása nehéz feladat. Felmerült ez a lehetőség is, de ehhez speciális keretrendszerre (Selenium<sup>13</sup>) és használatának elsajátítására lett volna szükség, ami túlmutatott a szakdolgozat témáján. A manuális tesztek a program fejlesztése közben természetesen mindig előfordultak, de nem minden osztály teszteléséhez volt szükség erre, illetve elsősorban segédosztályoknál előfordult, hogy egyáltalán nem volt lehetőség a böngészőből tesztelésre. Ezekhez az osztályokhoz JUnit tesztek készültek.

## Manuális tesztek

### Fekete doboz

Fekete dobozos tesztesetek ennél a programnál azok, amik a grafikus felületen beadott input alapján valamit visszajeleznek, tehát amit bárki le tud ellenőrizni egy böngészőből, ha megnyitja az oldalt.

<sup>13</sup> Selenium: Web-böngészők grafikus felületének tesztelését automatizáló programcsomag/keretrendszer. <http://seleniumhq.org/>

#### Pozitív tesztek:

Előfeltétel	Feltéve, hogy elindult a program
<b>Teszt</b>	Az <i>honnan</i> és <i>hova</i> mezőket kitöltve helyes értékekkel, a <i>mikor</i> és a <i>mennyiért</i> mezőt alapértelmezetten hagyva
<b>Elvárt eredmény</b>	A térképe a program megjeleníti az útvonalat és a jelölőt a térképen, és megjeleníti a térkép alatt egy táblázatban az utazás információit.

Előfeltétel	Feltéve, hogy elindult a program
<b>Teszt</b>	Az <i>honnan</i> és <i>hova</i> mezőket kitöltve helyes értékekkel, a <i>mikor</i> mezőnek tetszőleges helyes dátumot választva, és a <i>mennyiért</i> mezőt alapértelmezetten hagyva
<b>Elvárt eredmény</b>	A térképen a program megjeleníti az útvonalat és a jelölőt a térképen, és megjeleníti a térkép alatt egy táblázatban az utazás információit a megfelelő dátummal.

Előfeltétel	Feltéve, hogy elindult a program
<b>Teszt</b>	A <i>honnan</i> , <i>hova</i> , <i>mikor</i> , <i>mennyiért</i> mezőket helyesen kitöltve
<b>Elvárt eredmény</b>	A program megjeleníti az útvonalat és a jelölőt a térképen, és megjeleníti a térkép alatt egy táblázatban az utazás információit a megfelelő dátummal, és a kiválasztott kedvezménynek megfelelő árlistával.

Előfeltétel	Feltéve, hogy a lekérdezés sikeres volt
<b>Teszt</b>	A térképen megjelenő jelölőre kattintva
<b>Elvárt eredmény</b>	Megjelenik a célállomáson lévő helyek listája.

<b>Előfeltétel</b>	<b>Feltéve, hogy a célállomáson lévő helyek listája meg van nyitva</b>
<b>Teszt</b>	Egy helyre rákattintva
<b>Elvárt</b>	Megjelenik a hely részletes információja.
<b>eredmény</b>	

<b>Előfeltétel</b>	<b>Feltéve, hogy a célállomáson lévő helyek listája meg van nyitva</b>
<b>Teszt</b>	A hozzáadás elemre rákattintva
<b>Elvárt</b>	Megnyílik az új hely hozzáadása panel.
<b>eredmény</b>	

<b>Előfeltétel</b>	<b>Feltéve, hogy az új hely hozzáadása panel meg van nyitva</b>
<b>Teszt</b>	A mezőket helyesen kitöltve és az OK gombra kattintva
<b>Elvárt</b>	Eltűnik a panel, és a főoldalon a mezők alatt megjelenik egy
<b>eredmény</b>	felirat, ami közli, hogy az új hely elmentése sikeres volt.

#### Negatív tesztek

<b>Előfeltétel</b>	<b>Feltéve, hogy elindult a program</b>
<b>Teszt</b>	Az <i>honnan</i> vagy a <i>hova</i> mezőben lévő név nem szerepel az adatbázisban
<b>Elvárt</b>	A főoldalon megjelenik egy felirat, ami szól, hogy a mező adatai
<b>eredmény</b>	nem találhatóak az adatbázisban.

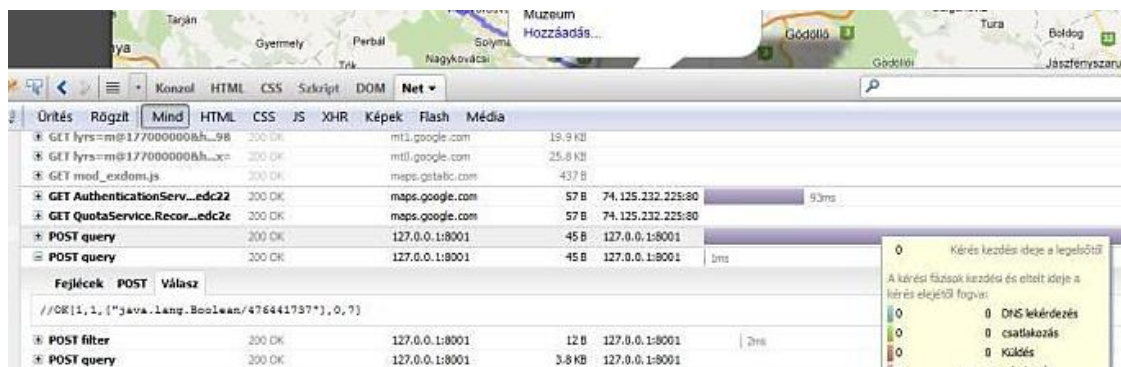
<b>Előfeltétel</b>	<b>Feltéve, hogy a célállomáson lévő helyek listája meg van nyitva</b>
<b>Teszt</b>	Ha a célállomáson lévő helyek listája üres
<b>Elvárt</b>	Akkor két elem jelenik meg a listában, az egyik egy nem
<b>eredmény</b>	megnyitható felirat, ami közli, hogy nincsen érdekes hely a célállomáson, valamint a Hozzáadás gomb.

<b>Előfeltétel</b>	<b>Feltéve, hogy az új hely hozzáadása panel meg van nyitva;</b>
<b>Teszt</b>	A mégse gombra kattintva;
<b>Elvárt eredmény</b>	Eltűnik a panel.

<b>Előfeltétel</b>	<b>Feltéve, hogy az új hely hozzáadása panel meg van nyitva</b>
<b>Teszt</b>	Ha nem töltjük ki a név vagy honlap vagy leírás mezőket
<b>Elvárt eredmény</b>	Egy hibaüzenet jelenik meg a panelen, ami közli, hogy hiányosan töltöttük ki az adatokat.

### Fehér dobozos

A fehér dobozos tesztelés tesztesetei megegyeznek a fekete dobozosokkal. A különbség az, hogy a tesztek működési mechanizmusát különböző eszközökkel meg lehet figyelni. Az egyik a tesztek közbeni debuggolás, a másik pedig a Firefox böngésző Firebug kiegészítője, amivel a szerverhez menő kéréseket és az onnan jövő válaszokat lehet nagyon részletesen és hatékonyan monitorozni.



### 20. Firebug plugin: hálózati kapcsolatok monitorozása

#### JUnit

A program kódja tartalmaz 36 darab JUnit tesztet. Különböző célokkal lettek létrehozva a fejlesztés egyes szakaszaiban, és különböző feladatokat is látnak el, de mind a belső működés különböző fontos részeit tesztelik. Az egyes függvények nevei a JUnit 4 elnevezési konvencióit követve hosszúak, és végig leírják az egyes tesztek elvárásait. Például az a teszt, ami egy Elvira Api-s lekérdezést tesztel, ami nem létező város nevét kérdezi le az távoli szervertől, és azt várja, hogy IOExceptiont dobjon a kód, így néz ki:

```

@Test(expected = IOException.class)
public void queryWithNotExistingStationShouldGiveIOException() throws
IOException{
    ...
}

```

Mivel a tesztek így tökéletesen dokumentálva vannak a kódban, ezeknek részletes leírására nem térek ki.

Az AllTest.java teszt csoport futtatja le a program összes tesztjét. Az osztályok amiket letesztelnek mind bizonyos szempontból kritikusak:

A Shared csomagból a Coorinate, a Town és az InterestingPlace osztályok a hálózaton az információ továbbítói, ezért kritikus, hogy jól épüljenek fel létrehozáskor, a felülírt equals függvényeik jól működjenek, és az esetleges logika ami bennük van, szintén hibátlan legyen.

Az ElviraApi osztályt tesztelő Junit nagyban segítette az osztály jó kialakítását, modulárisá alakítását, és segített megismerni a távoli REST interfész működését a különböző hibák esetén. Ez nagyon fontos volt, mivel az interfész hiányosan volt dokumentálva. Ezzel az osztályban szoros összefüggésben áll a TravelInfoCreator tesztelése, ami a lekérdezett adatokból állította össze a kliensnek küldendő adatszerkezetet. Ebben kellett az API esetleges hibáit kiszűrni, és letesztelése nagyban segítette, hogy az elkészült tábla tényleg az elvártnak megfelelően működjön.

A legfontosabb tesztek a MapService tesztjei, amik az adatbázis fejlesztése közben nyújtottak hatalmas segítséget, mivel az alkalmazás egy teszt-adatbázissal készült el, és a tényleges RDF adatbáziskezelés implementációja később került bele a kódba. Ekkor a tesztek pontosan definiálták az elvárt működést, nagyban megkönnyítették azt, hogy az új implementáció ugyanazokat a dolgokat és ugyanúgy lássa el, mint a régi.

## Skálázhatóság

A program kihasználja a jelenleg elérhető legújabb webes technológiákat, amik rengeteg problémát oldanak meg. Az egyik a skálázhatóság problémája, amit az alkalmazásszerverek vesznek át. Ezért a skálázhatóság problémájával, ebben az esetben azzal, hogy mennyi ember használhatja egyszerre az alkalmazást, nincsen

probléma. A kliens oldal a felhasználók oldalán fut, a szerver oldalon pedig az állapotmentes szervletek szolgálják ki az egyes kéréseket.

A szűk keresztmetszetet az adatbázis jelenti, ami közös erőforrás. Fejlesztésekor oda kellett figyelni a szálbiztonságra, és ez lassíthatja az kérések kiszolgálását. A másik probléma pedig a szerver hardveres teljesítménye, ami természetesen véges számú felhasználót tud kiszolgálni, erőforrásaitól függően.

## RDF Adatbázis

Az RDF adatbázis készítésénél merültek fel elsősorban problémák. A forrásadatbázis táblái nagyok voltak, és az adatok szűrése időkölséges művelet volt, mert a merevlemezről beolvasott adatsorok szűrése kivárhatatlanul hosszú ideig tartott egy erős gépen is. A probléma abból fakadt, hogy az adatokat a memóriába beolvasás helyett a merevlemezről olvasta a program. A nagy méretű forrásadatbázist a hatékonyabb használat érdekében be kellett olvasni a memóriába, ami a feladat lefutását 4 órára rövidítette, ami még mindig elfogadhatatlan volt az adatmennyiséghez képest. Ezeket a VisualVM<sup>14</sup> eszközzel készült mérések mutatják be:



### 21. Mérési eredmények:

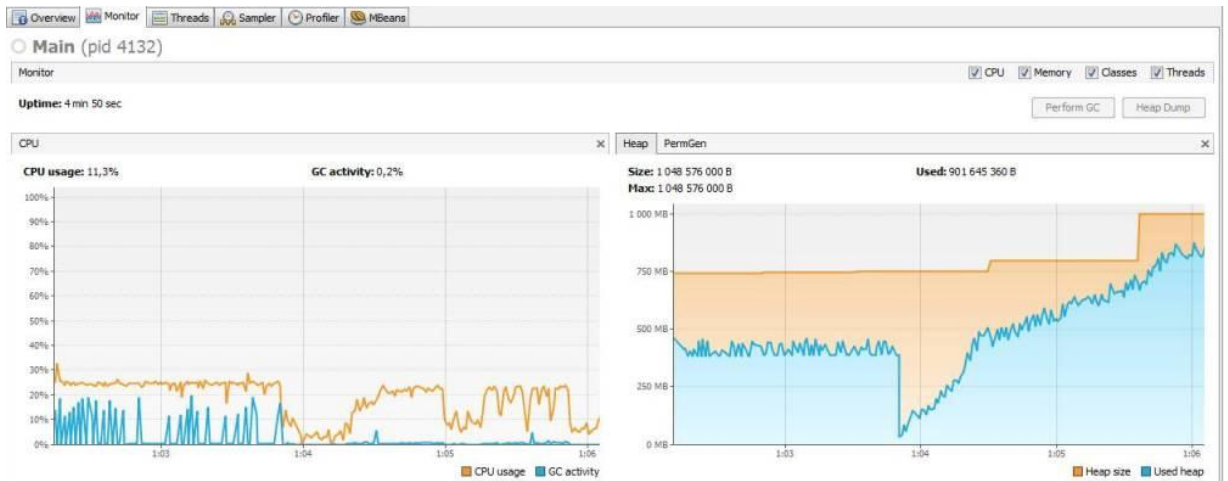
**Uptime 4 óra 9 perc 50 másodperc**

**CPU használat 37.5%**

**Memória használat: 400~500 MB**

<sup>14</sup> Visual VM: Java keretrendszer által biztosított eszköz, amivel a JVM paramétereit, memóiafogyasztás, GC, processzorhasználat és egyéb hasznos dolgokat lehet monitorozni.

Az beolvasott adathalmazok feldolgozási módszerében listákról hash táblák használatára tértem át, ekkor sikerült a kívánt sebességet elérnem, ami körülbelül 4-5 perc alatt végezte el az adatbázis legenerálását. Ennek hátránya, hogy a program memóriaigénye jelentősen megnőtt, de elfogadható volt.



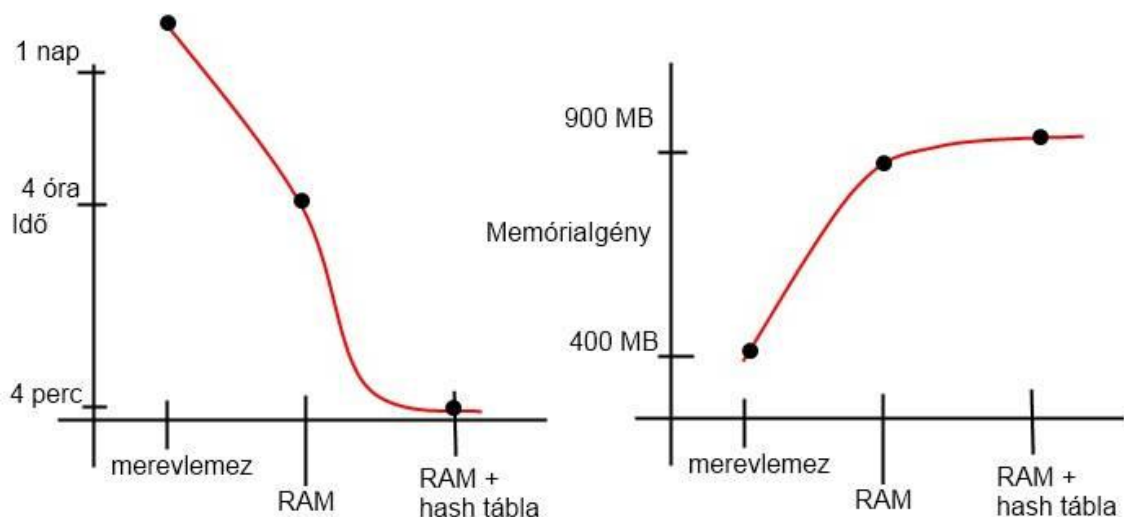
## 22. Mérési eredmények:

Uptime 4 perc 50 másodperc

CPU használat 11.5%

Memória használat: 500 – 1000 MB

A nyert sebesség jóval meghaladta a vesztett memória mennyiségét, így ezt a megoldást választottam a problémára.



## 23. Az idő és a memóriaigény az adathasználat módszeréhez viszonyítva



## Összegzés

### A kód tisztasága

A kód bemutatásáról nehéz beszélni, arról a kód beszél maga. De van néhány hasznos mérőszám, amivel képet lehet adni arról, hogy a kód mennyire sikeresen követte ezeket a szabályokat.

Ezeket a mérőszámokat az Eclipse CodePro plugin segítségével mértem le.

Kiemelném belőle az Average Block Depth mérőszámot, ami az egymásba ágyazott elágazások és ciklusok számát méri egy függvényen belül, és nagyon jól mutatja a függvények egyszerűségét. Kiemelném az Average Lines Of Code Per Method mérőszámot, ami értelemszerűen a függvényekben lévő sorok számának átlagát adja meg. Kiemelném az Average Number of Parameters mérőszámot, ami egy függvénynek által bekért paraméterek száma átlagosan. És végül kiemelném még az osztályok igen magas számát(Number of Types).

Ezek a számok bár nem adják át a lényegét, de adnak egy mutatót arról, hogy mennyire sikerült követni a felállított szabályrendszert.

A plugin nem csak mérőszámokat biztosított, hanem figyelmeztetett is, ha a valami a kritikus határon kívül ment. Az ábrán látható piros felirat két osztályra figyelmeztetett, amiben nagyon sok függvény volt: az egyik egy teszt-osztály volt, így ezt nem vettem figyelembe. Minél több teszt van egy osztályhoz, annál jobb, ez nem hiba. A másik osztály 17 függvénnel az RDF adatbázis építő osztálya. Ez az osztály valóban jóval meghaladja az átlagot, de függvényei túlnyomó többsége privát, erősen összefügg a feladattal, így ezt sem változtattam meg. Az összes többi érték a program szerint az elvárt, illetve megfelelő értékek között van.

Metric	Value
+ Abstractness	26.7%
+ Average Block Depth	0.83
+ Average Cyclomatic Complexity	1.22
+ Average Lines Of Code Per Method	6.53
+ Average Number of Constructors Per Type	0.46
+ Average Number of Fields Per Type	1.30
+ Average Number of Methods Per Type	3.22
+ Average Number of Parameters	0.75
+ Comments Ratio	1.7%
+ Efferent Couplings	55
+ Lines of Code	2,355
+ Number of Characters	86,046
+ Number of Comments	42
+ Number of Constructors	33
+ Number of Fields	127
+ Number of Lines	2,948
+ Number of Methods	229
Number of Packages	17
+ Number of Semicolons	1,197
+ Number of Types	71

### Kezdeti problémák

Az egyik baj a PDF-ről más forrásadatra átállással volt. Ez gondot jelentett a fejlesztés első szakaszában. Az adatok kinyerése nehézkes volt, és a térképes megjelenítéshez szükséges adatokból semmi nem volt meg. Az igazi probléma az volt, hogy nem tudtam alternatív forrásról, ami ezeket az adatokat tartalmazta volna. Ekkor sietett segítségemre egyetemista barátom, Welker Zsombor, aki hobiból évek óta készíti a GTFS specifikáció szerinti adatbázist a MÁV útvonalairól. Az általa összeállított adatbázis segítségével sikerült a térképes adatokat megszerezni.

A másik probléma a Google Maps API-val volt. Az alkalmazás elég sok mindent jelenít meg a térképen, változatos formákban, és nem mindenre volt egy a könyvtár felkészítve, így rengeteg belső hibába és nehézségbe ütköztem, amikből többet végül ki kellett kerülni, vagy bent hagyni a kódban, fejlesztés közben messze ez okozta a legnagyobb gondot.

## Kudarcok

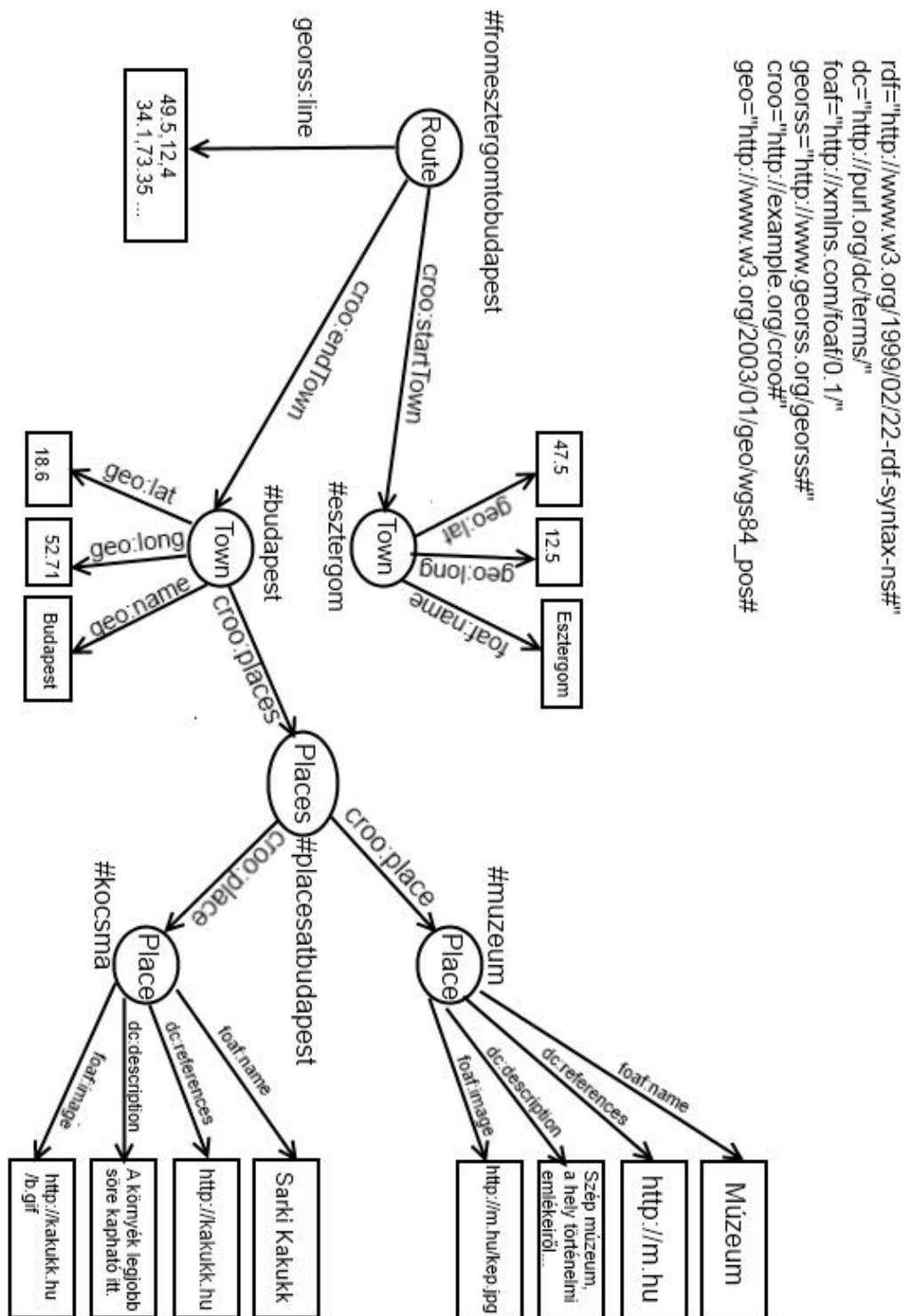
Bár a programban rengeteget foglalkoztam a kód struktúráltságával és szépségével, visszatekintve úgy érzem, csak a törekvést sikerült a kódba írni, de a szakdolgozat inkább nagyszerű példája refaktorálási lehetőségeknek, mint a tiszta kódnak.

A kód írása alatt többször kellett cserélnem külső eszközöket, kellett harcolnom hibás API-kkal, hiányos dokumentációkkal, néha eredménytelenül. A Java library ismeretének hiánya gyakran okozott olyan gondokat, amik amúgy elkerülhetőek lettek volna. Az adatbázis működik és jól ellátja feladatát, de a mérete indokolatlanul nagy, úgy érzem, nem sikerült a leghatékonyabb adattárolást a számomra új RDF technológiával megoldanom.

## Sikerek

A fejlesztés alatt a kitűzött személyes céljaimat sikerült elérnem. Megtanultam a Java programozási nyelvet, és rengeteg vele együtt járó technológiában tudtam elmélyedni. A tiszta kód írásának módszerének alapjait sikerült elsajátítanom, és eszméit magamévá tenni. Bár nem sikerült olyanra, mint amilyenre terveztem, de eredményként tudom felmutatni, hogy felismerem a (foglamazásban) hibás kódrészleteket, és pontosan el tudom mondani, milyen szabályokkal ütköznek. És úgy gondolom, ezzel hihetetlenül hatékony eszközre tettem szert, egy olyan eszközre, amit az egyetem órái között nem sajátíthattam volna el.

## Függelék



1. függelék

## Felhasznált irodalom

Flower, M. (2006. 6 28). *GUI Architectures*. Letöltés dátuma: 2012. 05 4, forrás: Martin Flower: <http://martinfowler.com/eaDev/uiArchs.html>

Flower, M. (2008). *Refactoring*. Westford: Addison-Wesley.

Hevery, M. (2008. 08 17). *Singletons are pathological liars*. Letöltés dátuma: 2012. 5 2, forrás: Miško Hevery: <http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/>

Martin, R. C. (2009). *Clean Code*. Stoughton: Prentice Hall.

Martin, R. C. (2011. Április 1). *Clean Coder Code-casts*. Letöltés dátuma: 2012. Január 12, forrás: <http://www.cleancoders.com>:  
<http://www.cleancoders.com/codecast/clean-code-episode-3/show>

McConnell, S. (2004). *Code Complete 2*. Redmond: Microsoft Press.

Rask, O. (2008. 1 1). *Model View Presenter*. Letöltés dátuma: 2012. 5 4, forrás: Wordpress: <http://mrrask.files.wordpress.com/2008/01/model-view-presenter.pdf>

Wikipedia. (2012. May 2). *Singletons*. Letöltés dátuma: 2012. May 4, forrás: Wikipedia: [http://en.wikipedia.org/wiki/Singleton\\_pattern](http://en.wikipedia.org/wiki/Singleton_pattern)