Jegyzőkönyv

Programozás 2. tárgyra

2018/2019-es tanév első félév

Készítette: Oláh Adél (RJDXPI)

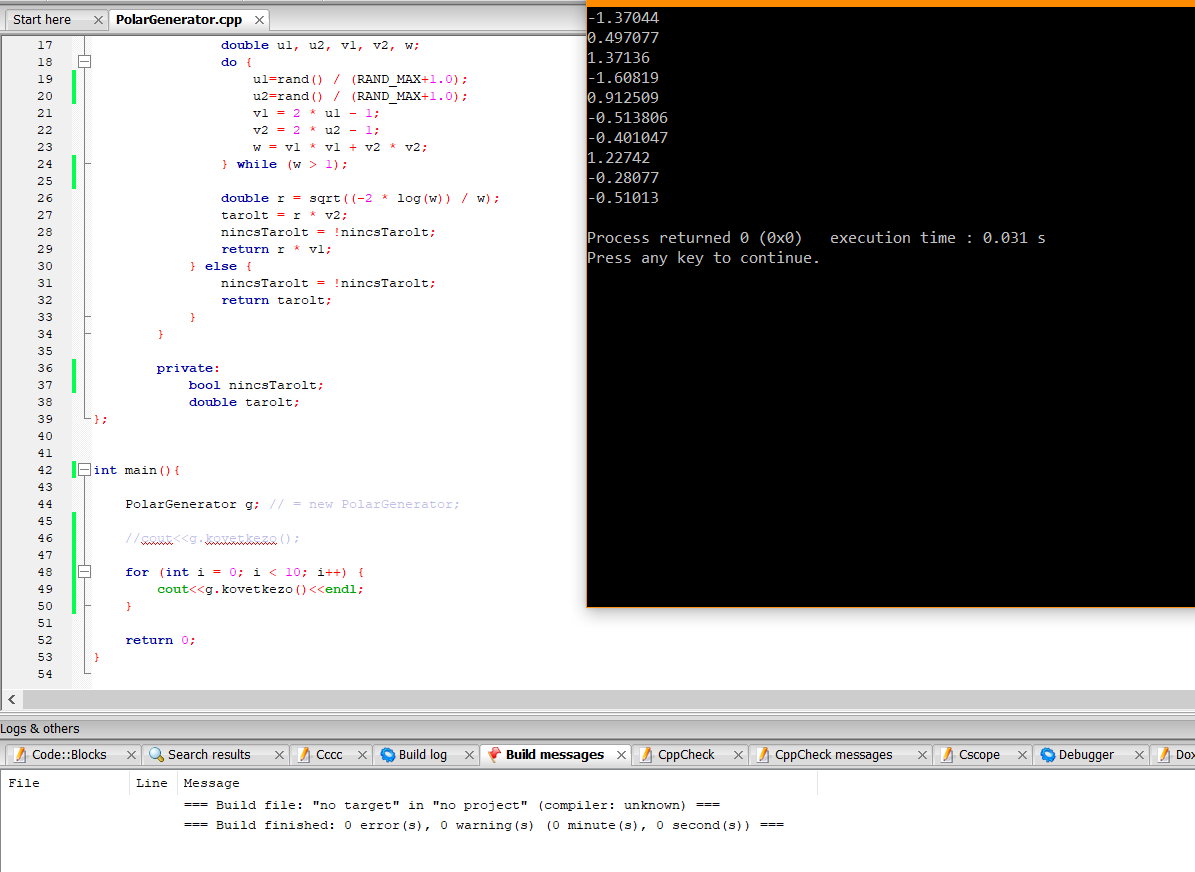
Forráskódok: *https://github.com/olahadel/UDPROG/tree/master/prog2*

Első hét:

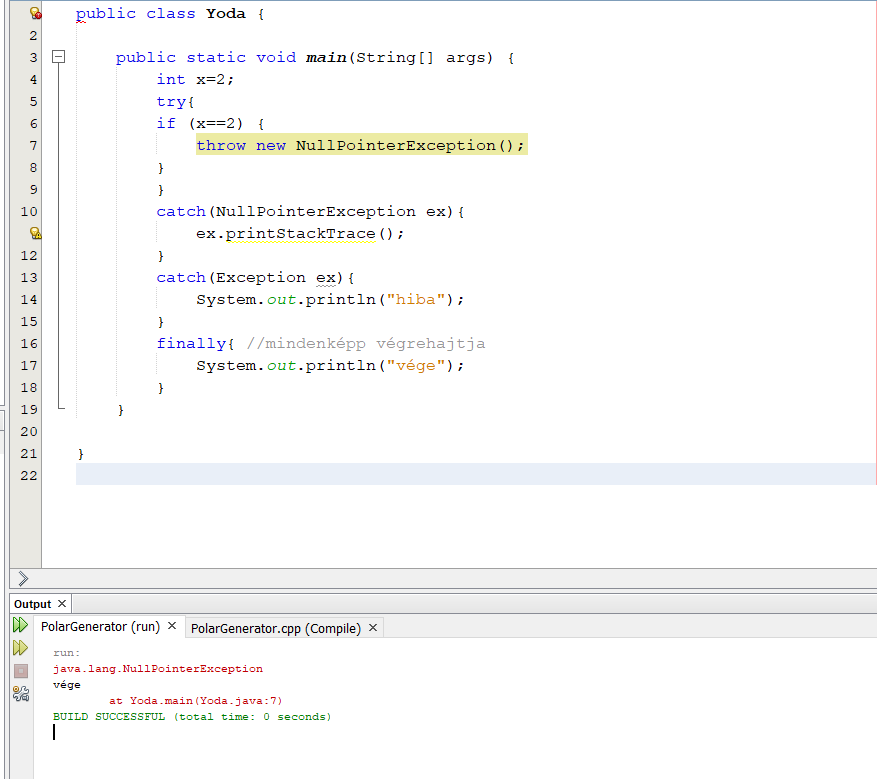
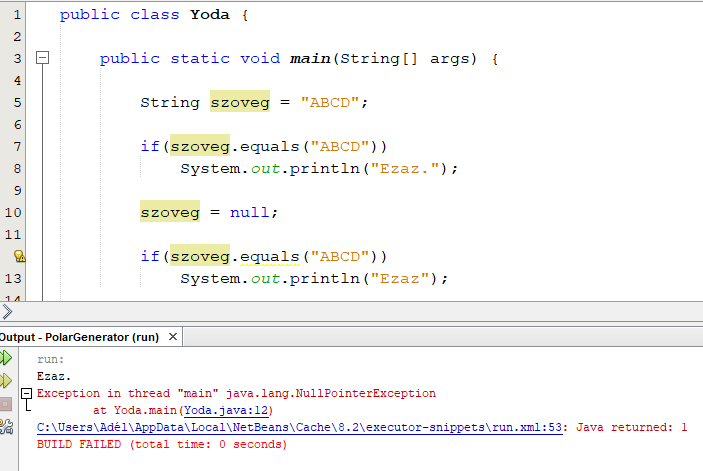
**OO szemlélet** (4. héten befejezve)  
**Homokozó**   
**„Gagyi”**   
**Yoda** (4. héten befejezve)  
**Kódolás from scratch** (2. héten befejezve)

2018.09.12.: Az első feladaton (**PolarGenerator**) dolgoztam, a c++ fájl még nem fut le.

2018.10.07.: Befejeztem az első heti **OO szemlélet** feladatot. Az eredeti Java változathoz hasonlóan ez is visszaadja a várt 10 számot.  
A Tanár Úr által belinkelt diasorban az OpenJDK (illetve az Oracle JDK) általi OO szervezés is ugyanaz, mint ebben a megoldásban. A diasorban szereplő kódban láthatunk egy nextNextGaussian és egy haveNextGaussian nevezetű változót, amely ebben a megoldásban sorrendazonosan a tarolt, illetve a nincsTarolt nevet kapta. Ezeknek megfelelően az eredetiben a nextGaussian() függvény megfelel a kovetkezo() függvénynek.  
A függvény alkalmazása esetén minden második (páratlan számú) híváskor fut le az algoritmus és először az r\*v1, majd az r\*v2 értéket adja vissza.   
Az eredeti kódban is ezt láthatjuk, csak míg ebben a logikai változó igaz értékről indul, az eredetiben hamisról, és ennek megfelelően cserélődik fel az if-else rész is.



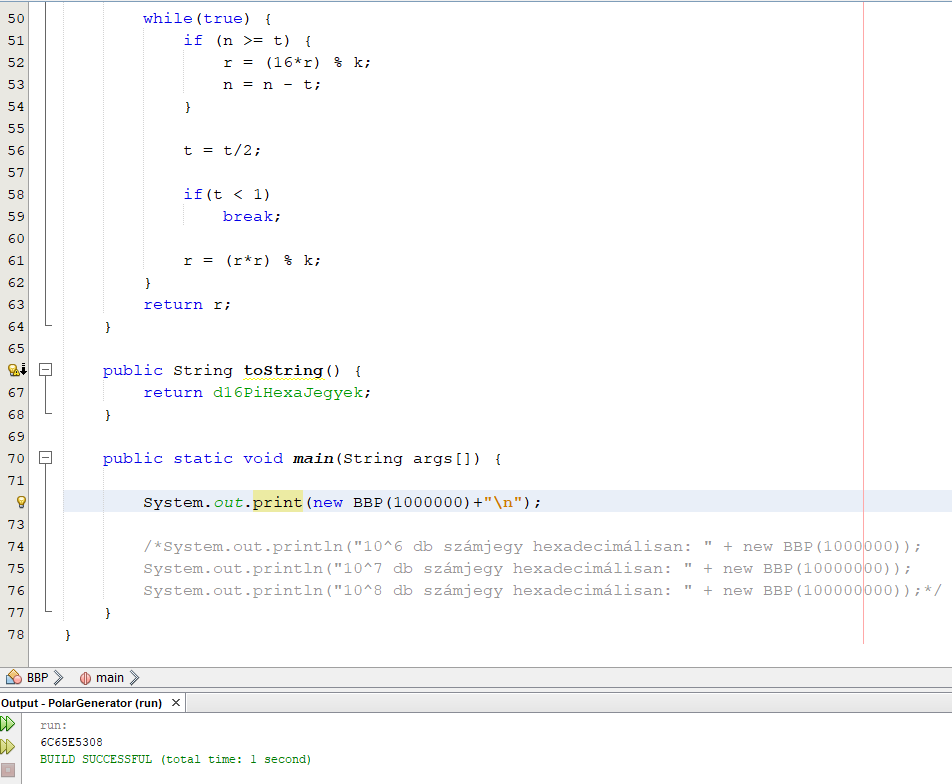
2018.09.14.: Kész a **Yoda** feladat, az én értelmezésemben.



2018.10.07.: Az első heti **Yoda** feladatot átírtam, így most már kikényszerítés nélkül dob kivételt. A Yoda conditions azt jelenti az én értelmezésemben, hogy a feltételvizsgálatban a kifejezés két részét a szokásos, hagyományos sorrendhez képest felcseréljük. A feladat szövege alapján kivételt akkor kell kapnunk, ha a szokásos sorrendet használjuk, nem pedig a megfordított Yoda sorrendet.   
Ennek megfelelően az én megoldásomban is kivételt kapunk a Yoda sorrend esetén.

A feladat kapcsán segítségemre volt: Ádám Szilárd

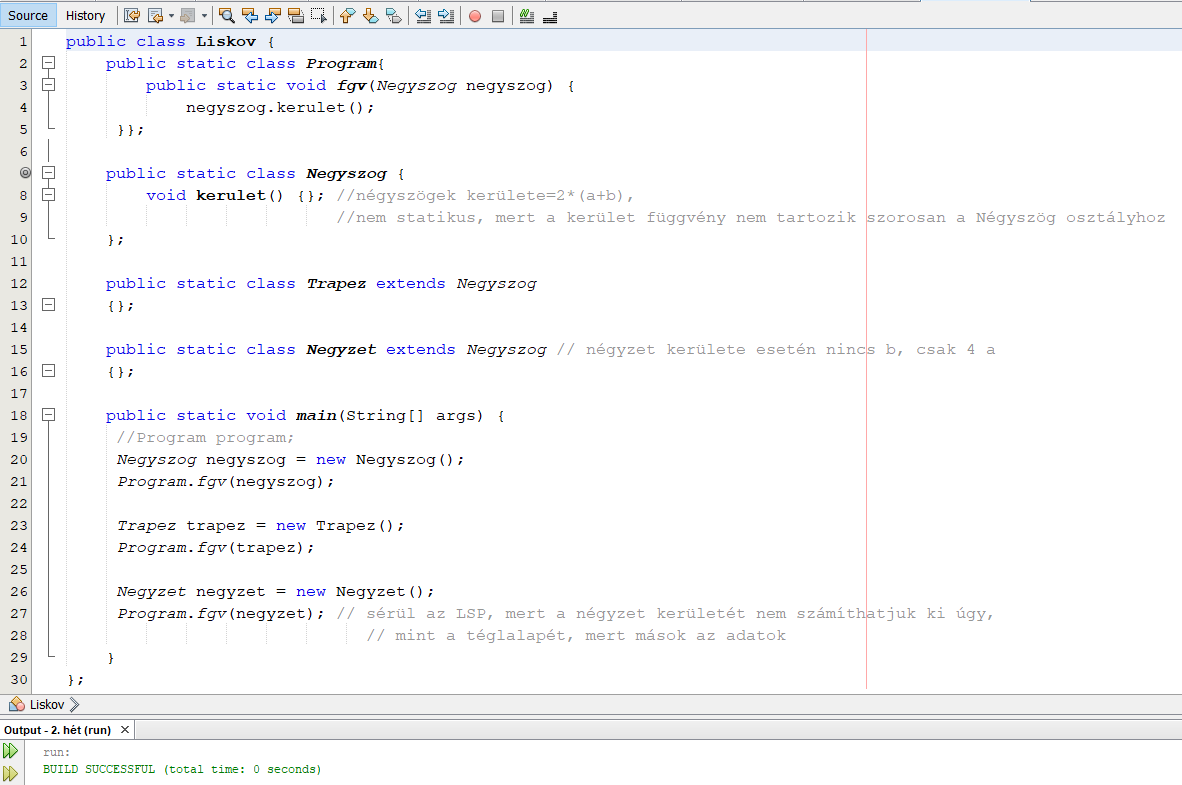
2018.09.16.: A **BBP algoritmust** tanulmányoztam a cikkből és elkezdtem dolgozni a Java kódon. (A második héten fejeztem be, mert akkor sikerült megérteni a forráskódot.)



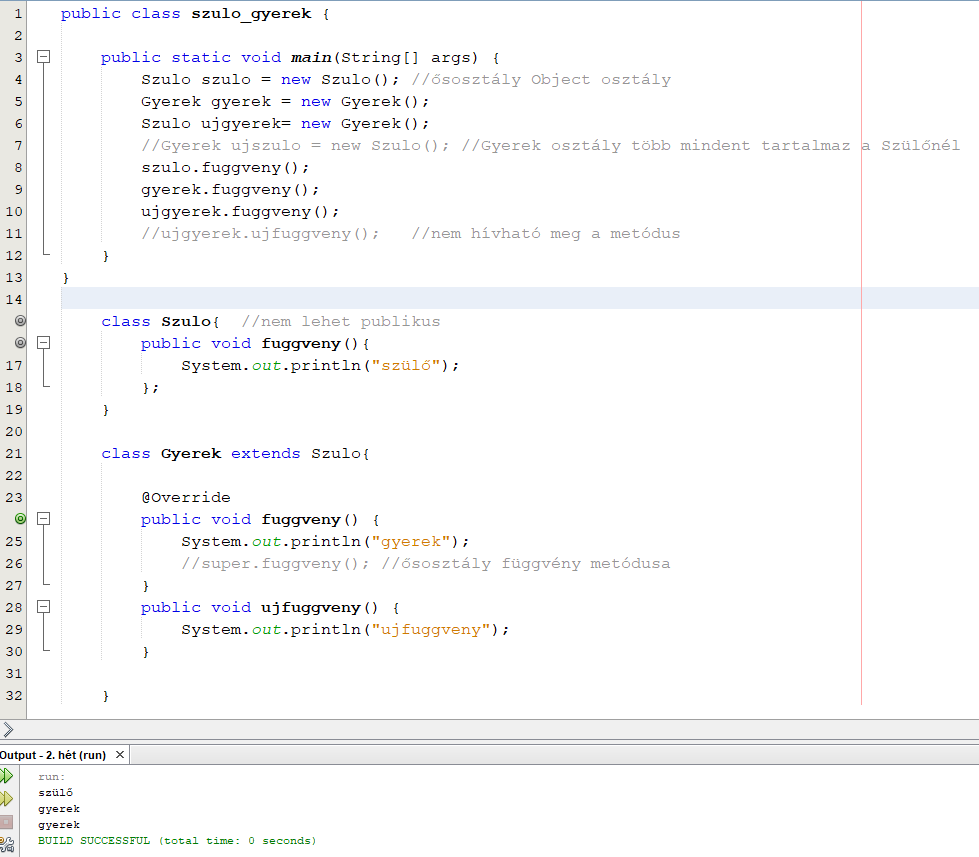
2. hét:

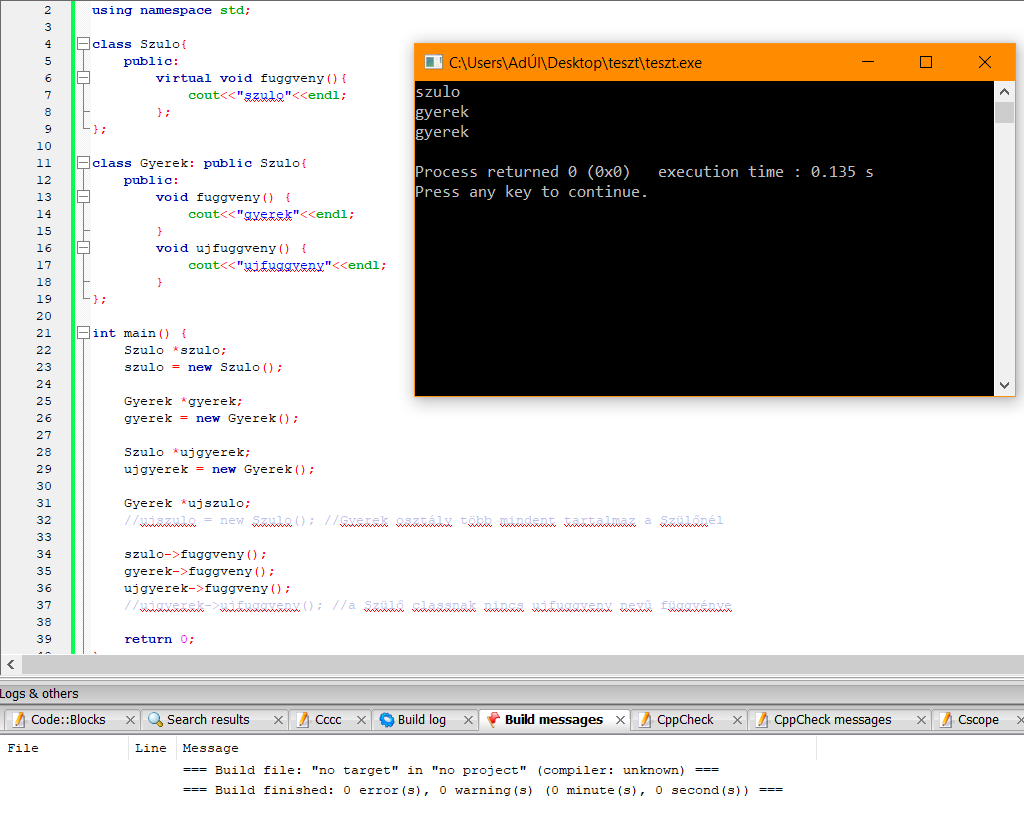
**Liskov helyettesítés sértése** (tárgyhéten befejezve) **Szülő-gyerek** (tárgyhéten befejezve) **Anti-OO** (tárgyhéten befejezve) **Hello, Android!   
Ciklomatikus komplexitás**

2018.09.19.: **Liskov elv megsértése** feladatra kész a C++, illetve Java kód. A feladatot a négyszögek kerületének kiszámításával szerettem volna bemutatni. Azért sérül az LSP a programban, mert a négyzet kerületét másképpen számítjuk ki, mint egy négyszögét; általános négyszög esetén A B C D bemenő paramétereink vannak, míg négyzet esetén csak A bemenő adatunk van. Általános trapéz esetén 4 különböző oldalunk van, ezért ott nem sérül az elv.



2018.09.22.: A **Szülő-gyerek** feladatban a kommentelt részekben látszik az, ami nem működik.   
A Szülő osztályt példányosíthatjuk a Gyerek osztály konstruktorával, de fordítva nem. A Gyerek osztály egy „bővebb” osztály lehet, mint a Szülő osztály, ezért az nem probléma, ha egy Szülő példánynak foglalunk le több helyet (amennyi egy Gyerek példánynak lenne szükséges). Ellenben a Szülő konstruktorával nem hozhatunk létre Gyerek példányt, mert ha a Gyerek osztály bővebb, akkor a Szülő konstruktora által lefoglalt hely nem feltétlenül biztosítaná a helyet a Gyerek elemei számára.  
Az ujgyerek.ujfuggveny() teljesíti igazából a feladatot. Az „ősön keresztül csak az ős üzenetei küldhetők” a feladat szerint. Az ős a Szülő osztály az üzenet pedig a fuggveny() metódus. Az ujgyerek.ujfuggveny() metódus esetén az ujgyerek a Szülő osztály egy példánya, míg az ujfuggveny a Gyerek osztály egy függvénye. Ez mutatja meg, hogy a szülő osztályon keresztül a gyerek osztály metódusa nem használható. (A Gyerek osztály példányán keresztül a Szülő metódusa használható bármilyen probléma nélkül, ahogyan ezt a program lefutása is mutatja.)



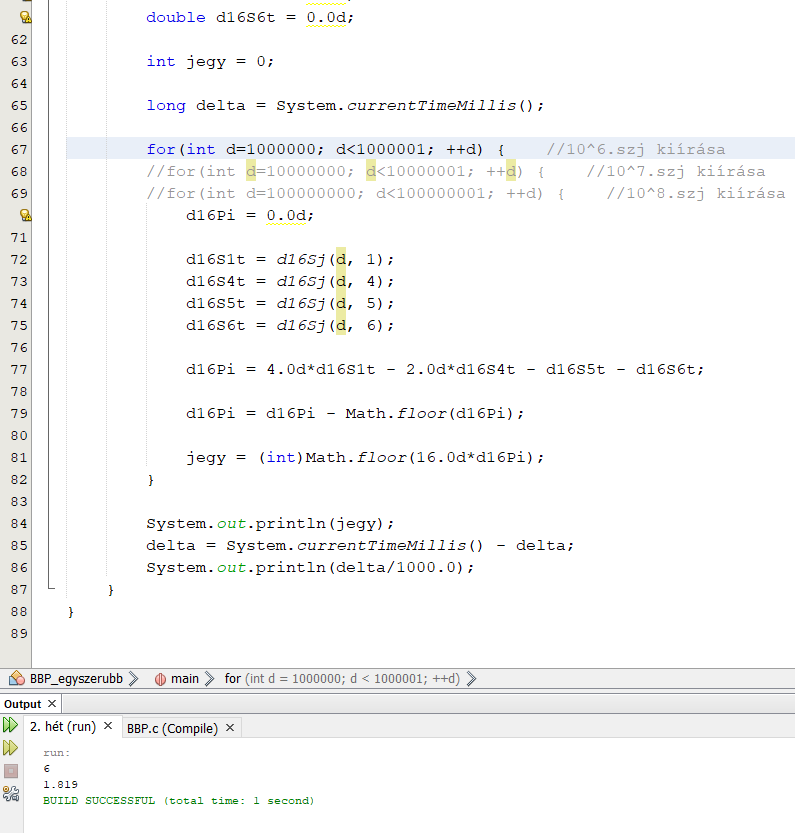
C++ -ban:

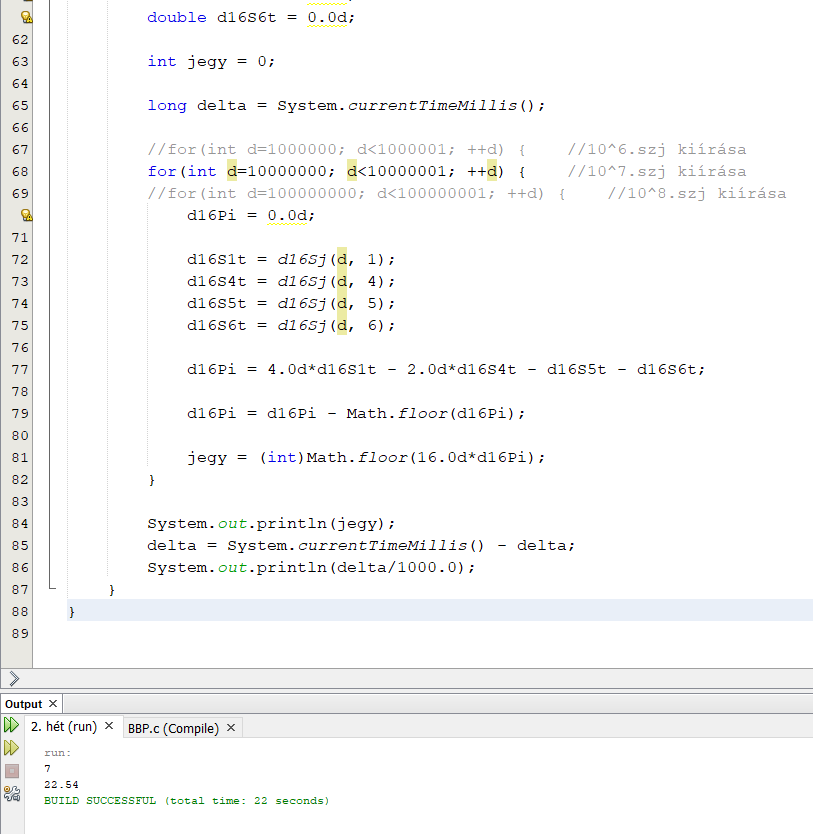
2018.09.25.: **Anti OO** feladat futási idők

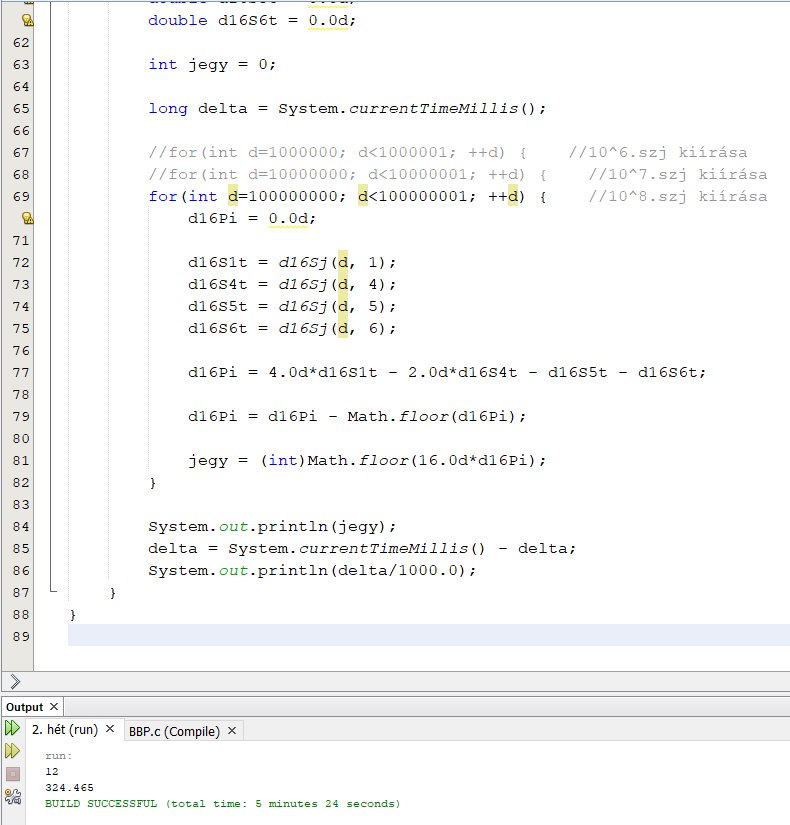
Összesítve:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 10^6. jegy | 10^7. jegy | 10^8. jegy |
| Java | 1,819 s | 22.54 s | 324,465 s |
| C | 1,313 s | 15,734 s | 228,237 s |
| C# | 2,846 s | 32,347 s | 564,417 s |
| C++ | 1,31 s | 15,765 s | 225,103 s |

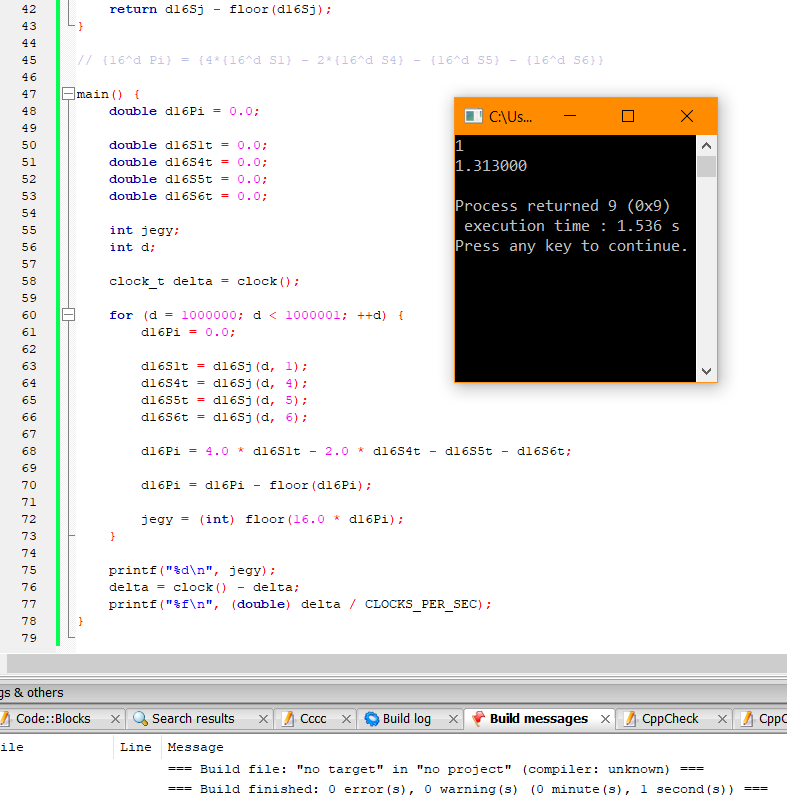
Javaban:

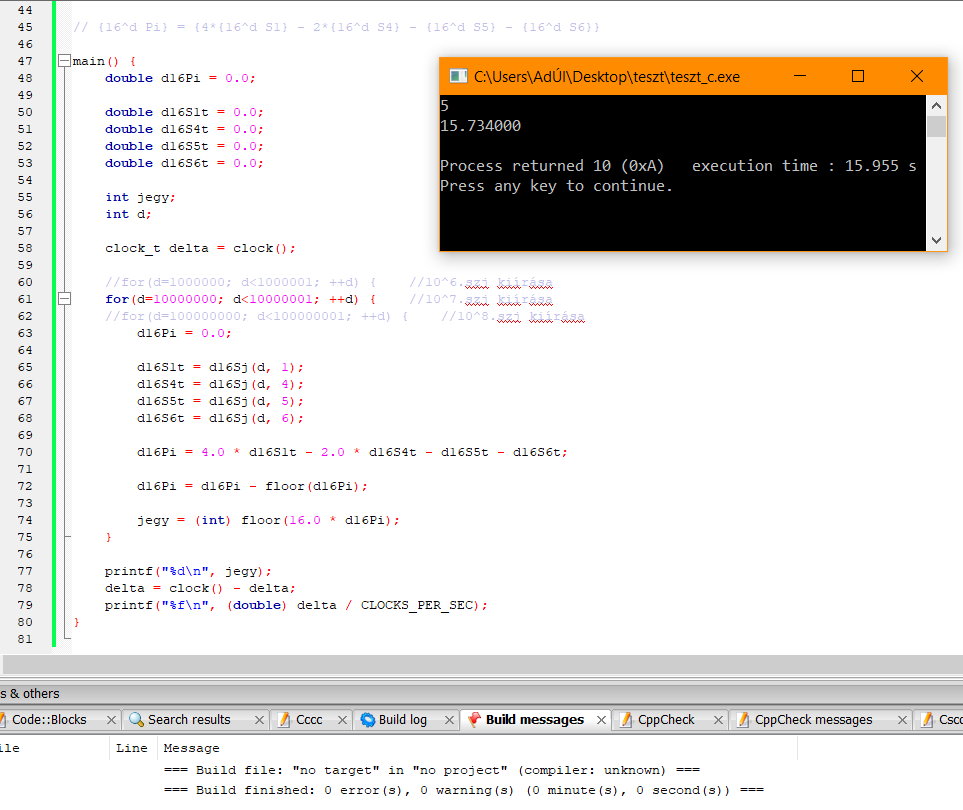
10^6. jegy:

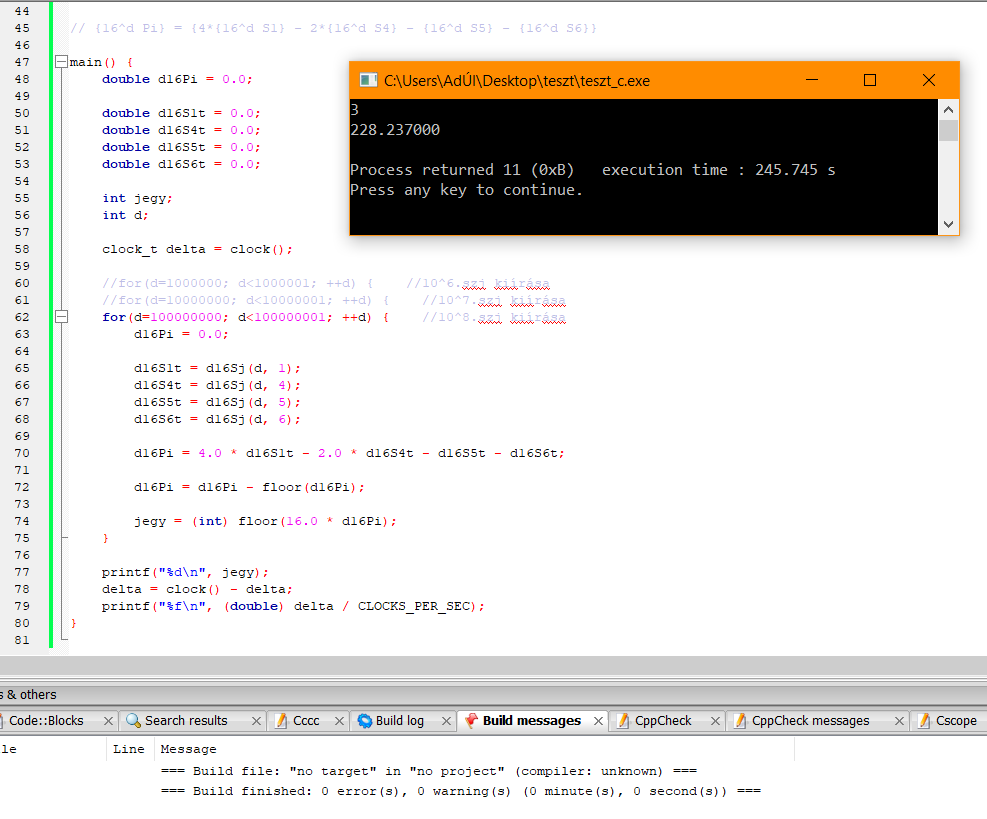
10^7.jegy: 

10^8. jegy:

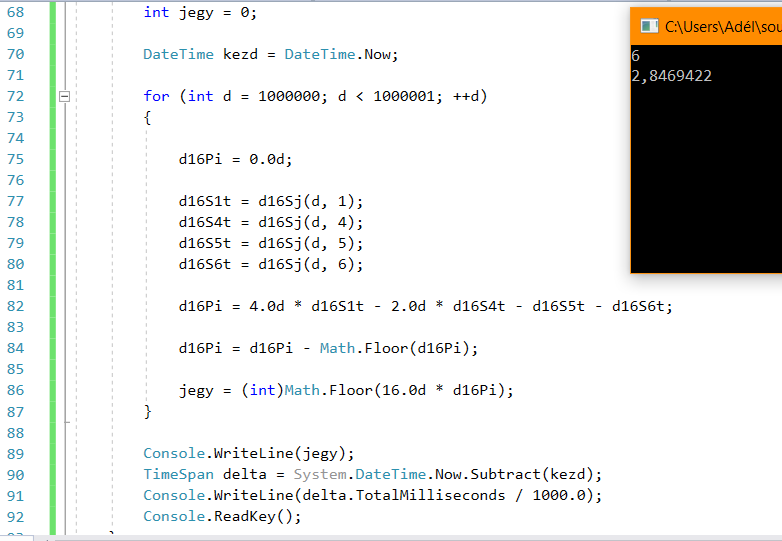
C-ben:

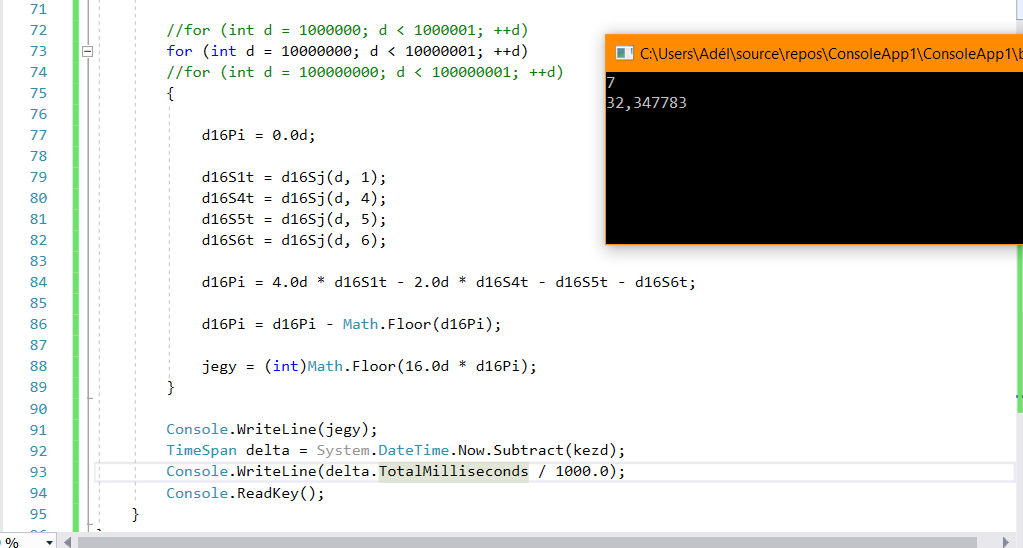
10^6. jegy:

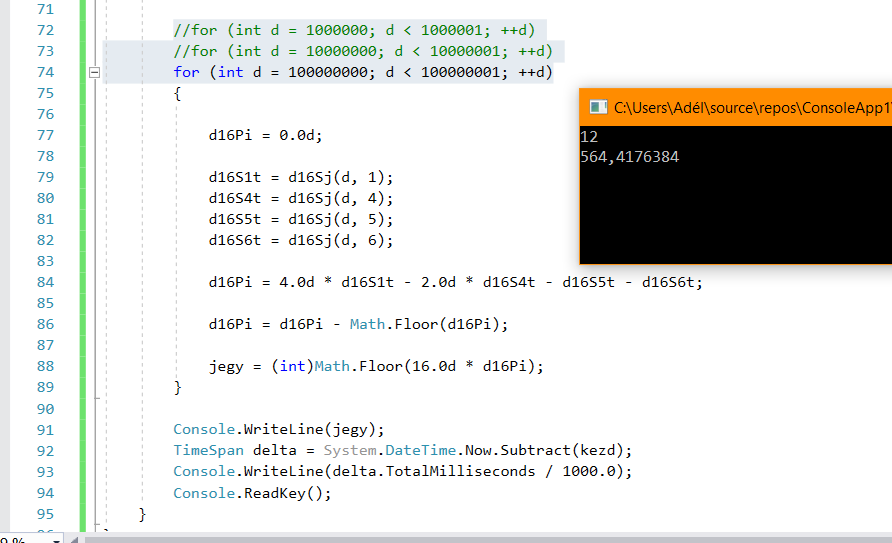
10^7. jegy: 

10^8. jegy: 

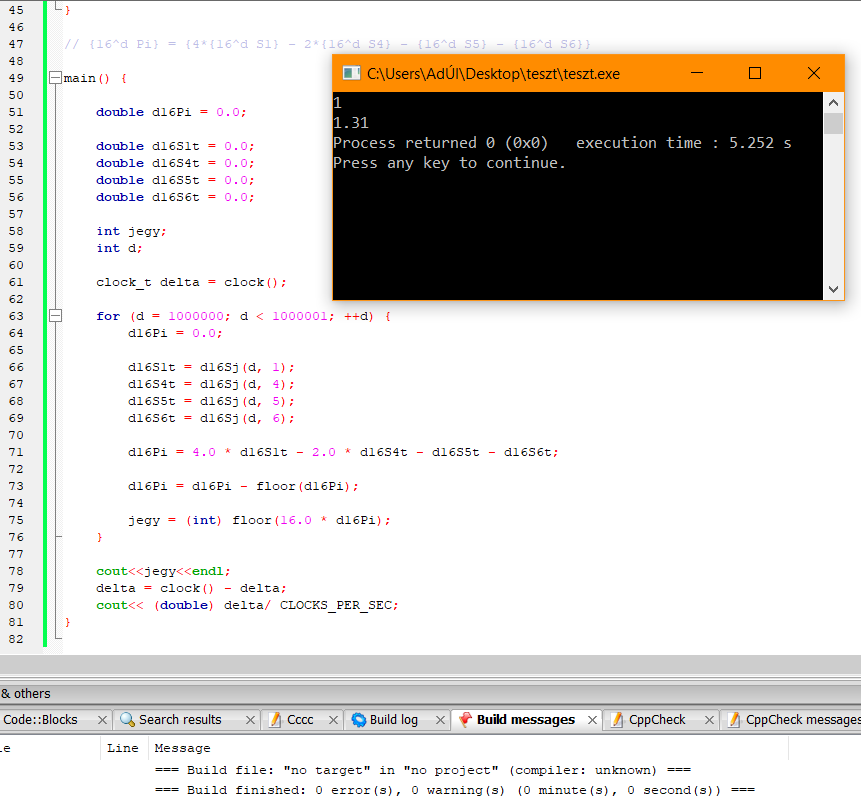
#C-ben:

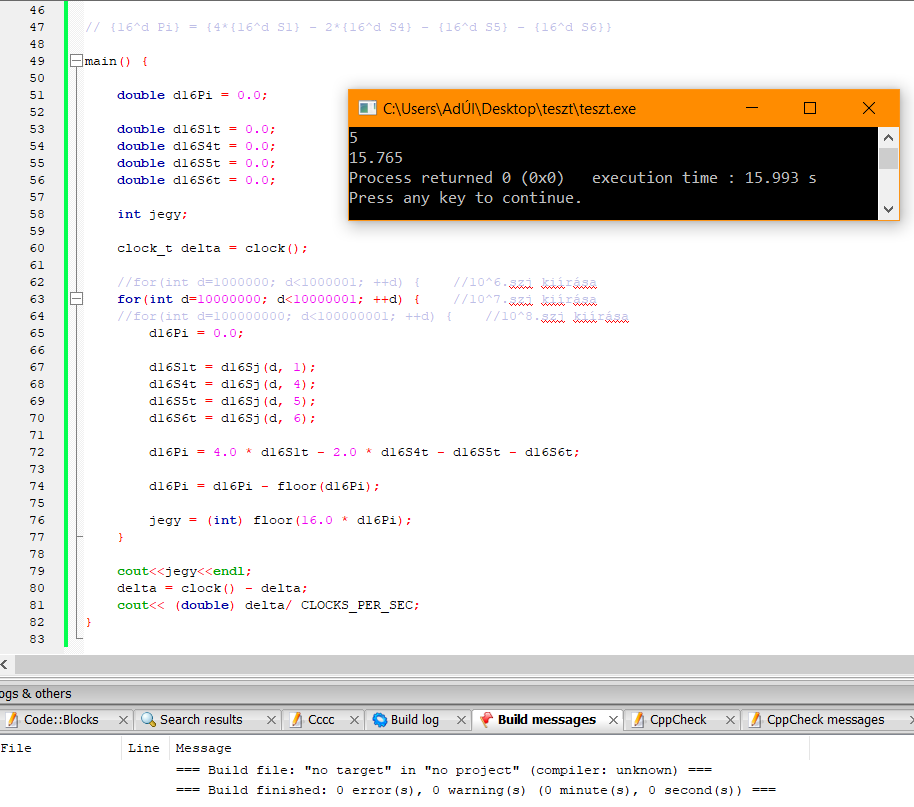
10^6. jegy:

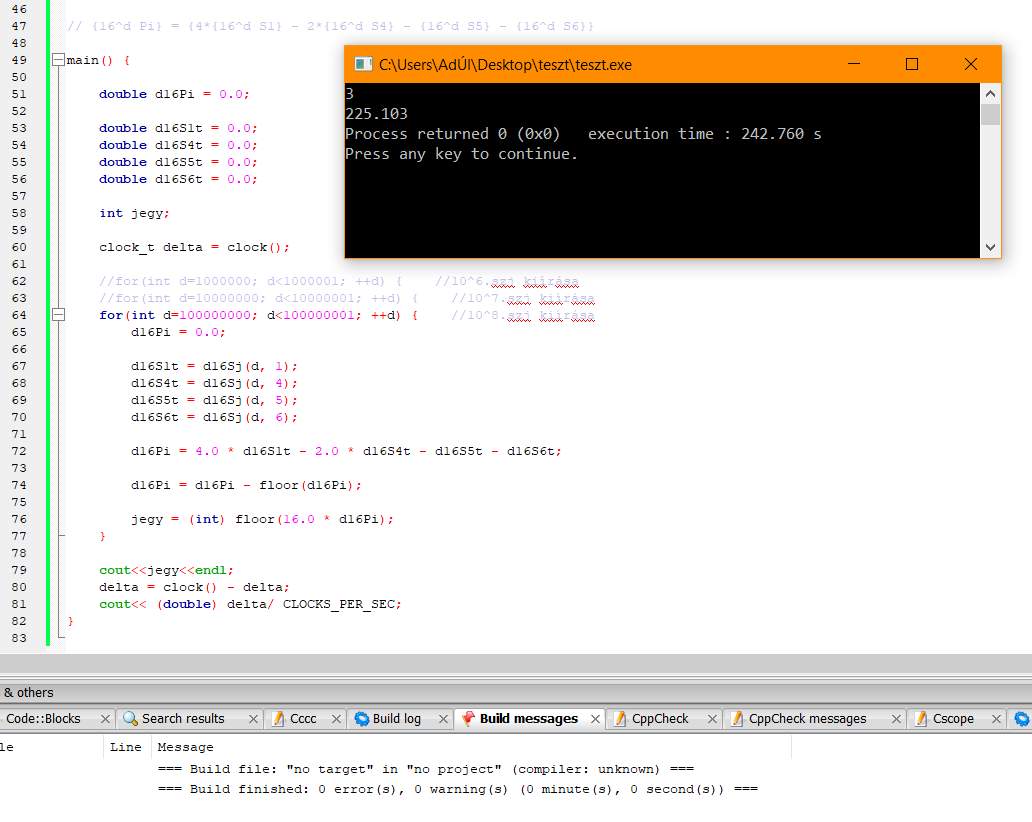
10^7. jegy: 

10^8. jegy: 

C++ -ban:

10^6. jegy:

10^7. jegy: 

10^8. jegy: 

3. hét

**Reverse engineering UML osztálydiagram** (8. héten befejezve)

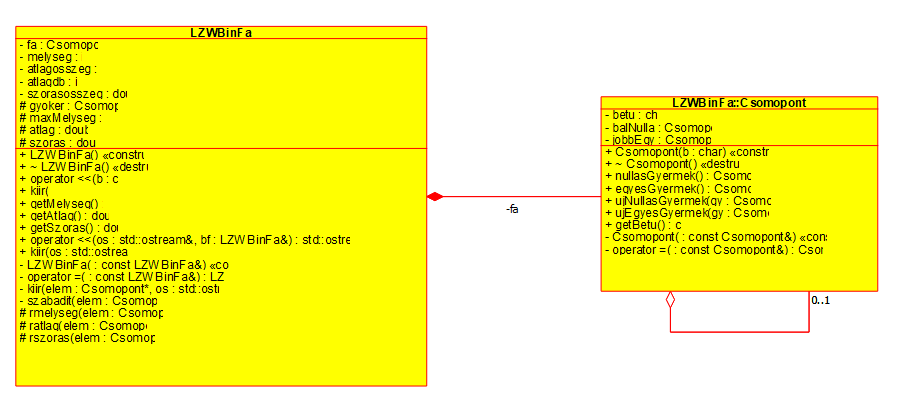
**Forward engineering** (8. héten befejezve)

**Egy esettan** (8. héten befejezve)

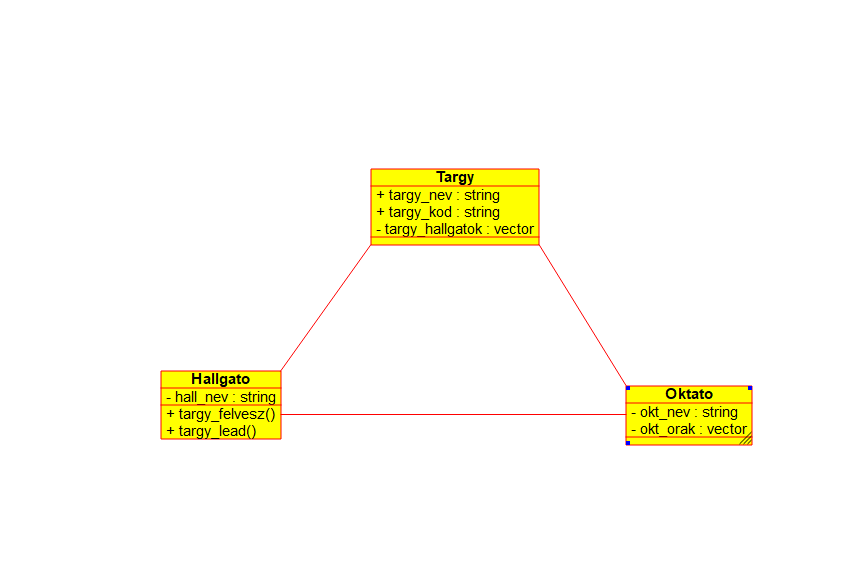
**BPMN**

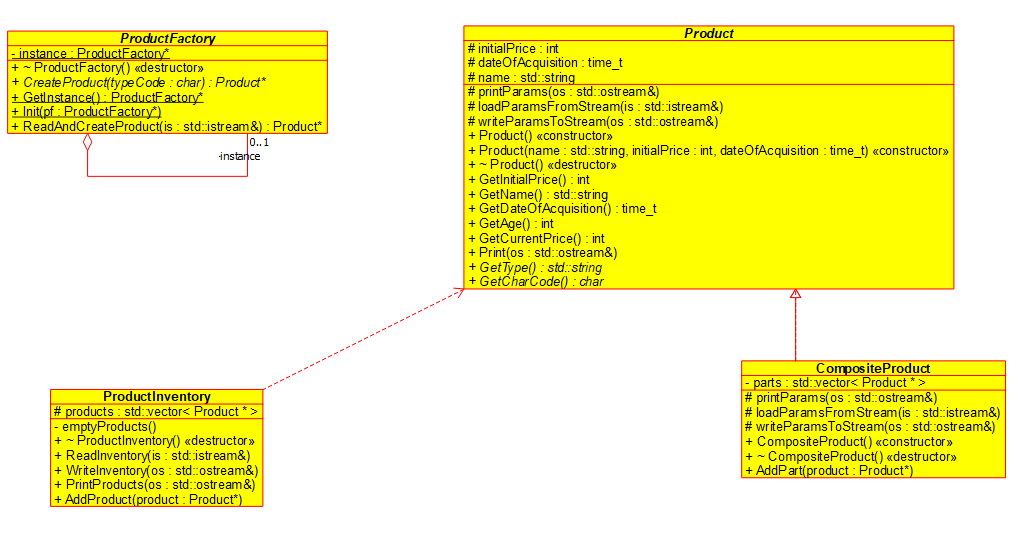
**BPEL**

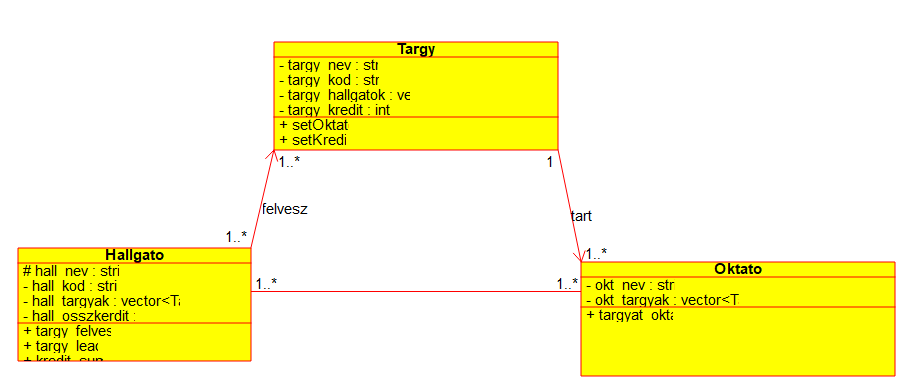
2018.09.29.-2018.11.11.: **Reverse** feladat: A kódot sikerült importálnom. Az eredeti binfa programban csak 2 osztály van definiálva, ezért a diagram is elég egyszerű.   
Az aggregáció és a kompozíció alapvetően azt fejezi ki, hogy egy „főosztály” tartalmaz egy „alosztály”-t. Az aggregáció olyan asszociáció, amely tartalmazást jelöl, jele a tartalmazó oldalán egy üres rombusz.   
Kompozíció esetén a tartalmazott objektum pontosan egy objektumhoz tartozik, vagyis a tartalmazott objektum nem lehet megosztva. Emellett a tartalmazó és a tartalmazott objektum együtt jön létre és együtt szűnik meg. A tartalmazott objektumért, mint erőforrásért a tartalmazó felel, így a felszabadításért is.   
(Forrás: Benedek Zoltán-Levendovszky Tihamér: Szoftverfejlesztés C++ nyelven c. könyv)  
Jelen esetben a fő osztály az LZWBinfa osztály és ezen belül van definiálva a Csomopont osztály. Ezek között áll fenn a kompozíció, mivel amennyiben a főosztály példánya felszabadításra kerül, úgy az alosztály példányai is, hiszen nincsen rájuk tovább szükség. A kompozíciót jelölő összekötő vonal alatt a „-fa” felirat azt jelöli, hogy a fa privát attribútum a – jel miatt.  
A diagramon még megjelenő aggregáció azt jelöli, hogy van-e a Csomopontnak további gyermek eleme (nullas, azaz bal oldali gyermek eleme).



Az **Esettan** feladatot elkezdtem, a könyvből az első esettanulmányt elolvastam, de a mintafeladatok kódját kigyűjtöttem, de még nem importáltam UML-be.

A **Forward** feladatnak a vázáig jutottam el. Az egyetemi rendszer leegyszerűsített változatát szeretném modellezni. Az oktatók, a hallgatók és a tárgyak kapcsolatát nagy vonalakban készítettem el eddig.  


2018.11.11.: Az **Esettan** feladat abból állt, hogy tanulmányozzuk a tankönyvben a 14-es esettanulmányt és a hozzá tartozó elméleti részt. Az ebből generált UML diagram:

**Forward** feladat: A kódok Githubon megtalálhatóak, a diagram pedig itt látható:  


4. hét

**Encoding** (tárgyhéten befejezve)

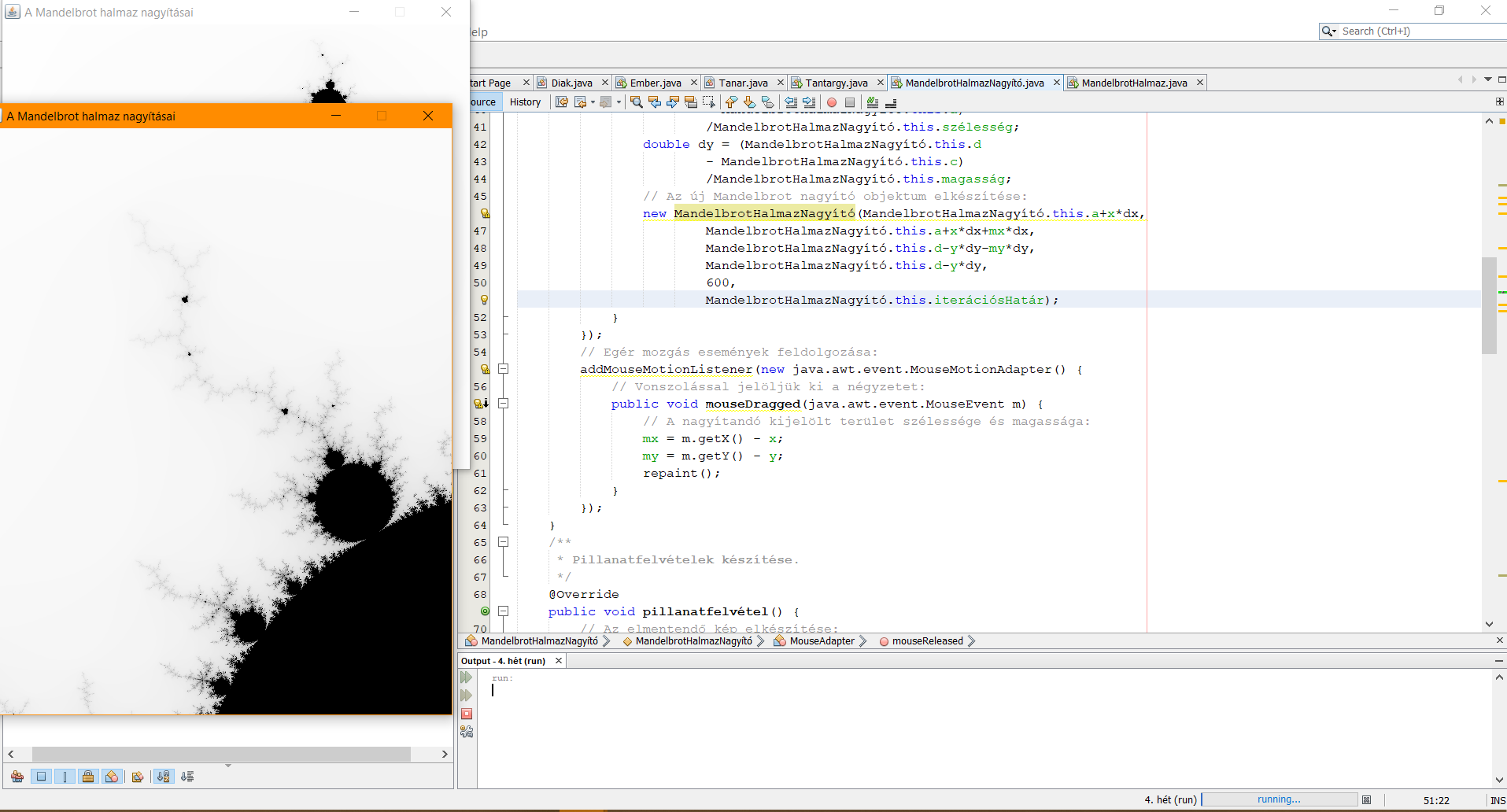
**OOCWC lexer**

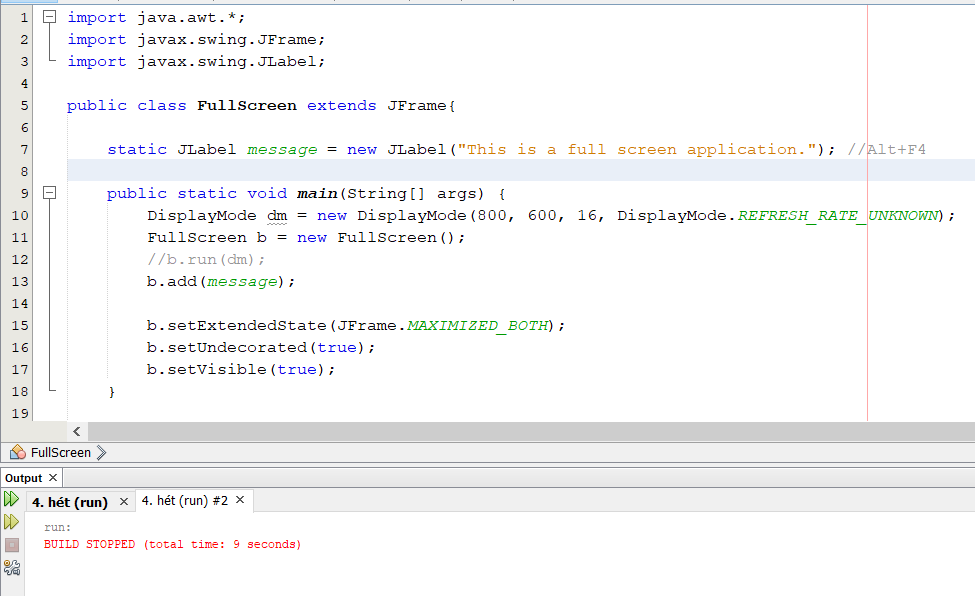
**l334d1c4** (tárgyhéten befejezve)

**Full screen** (tárgyhéten befejezve)

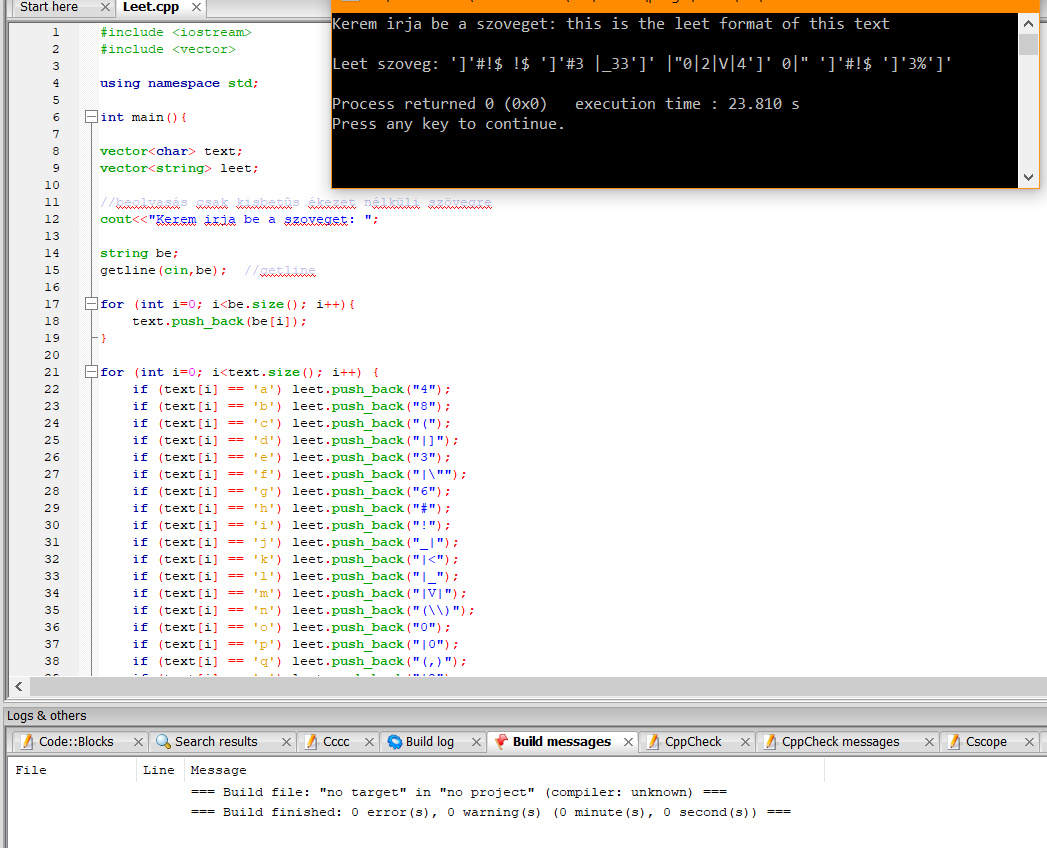
**Perceptron osztály**

2018.10.03.: Az **Encoding** feladathoz letöltöttem a forráskódokat, majd fordítottam, futtattam a kódokat. Minden probléma nélkül futottak az ékezetes betűkkel is. Ahogyan a programnak kell, nagyítás esetén az új ablakokat megnyitja.  
A nagybetűk használatát azért tehetjük meg probléma nélkül, mert a Java UTF-8 kódolást, illetve Unicode szabványt alkalmaz. Ez magába foglalja sok más mellett az ékezetes betűinket is.



2018.10.07.: **Full Screen** feladat, miszerint írjunk egy Java programot, amely teljes képernyős.   


**l334d1c4** feladat az volt, hogy írjunk egy programot, ami a normál szöveget leet szöveggé alakítja át. A leet szöveg olyan, mint a szleng, nincs rögzített, meghatározott formája. Ezért a leet karakterhelyettesítés az én változatomban:



5. hét

**JDK osztályok** (7. héten befejezve)

**Másoló-mozgató szemantika** (9. héten befejezve)

**Hibásan implementált RSA törése**

**Változó argumentumszámú ctor**

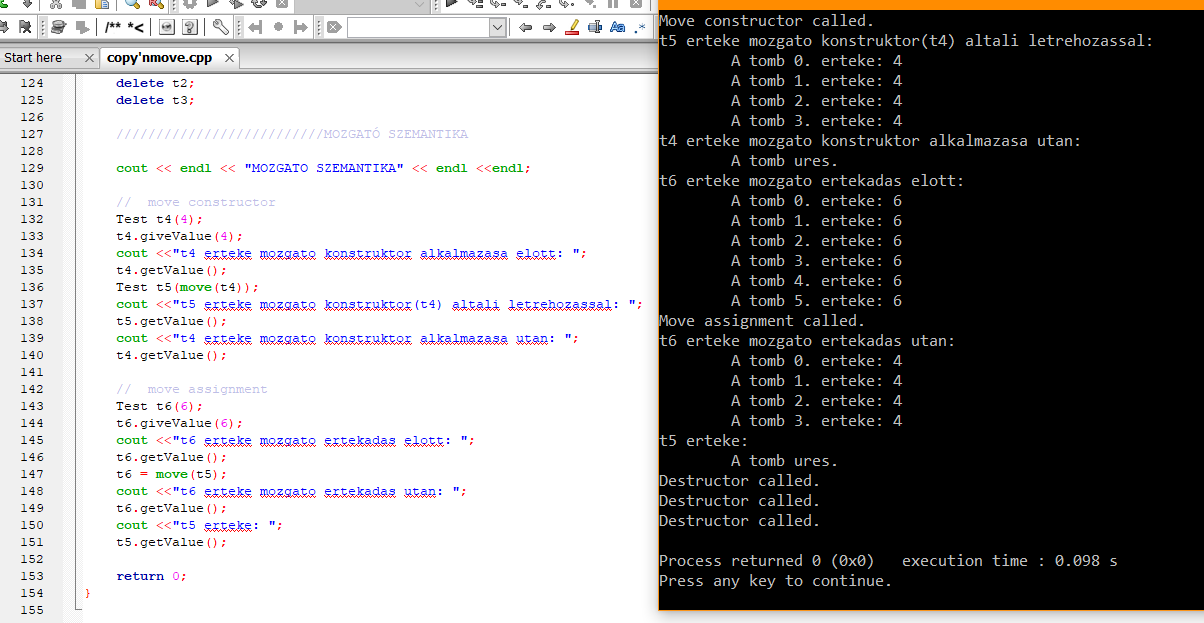
**Összefoglaló** (8. héten befejezve)

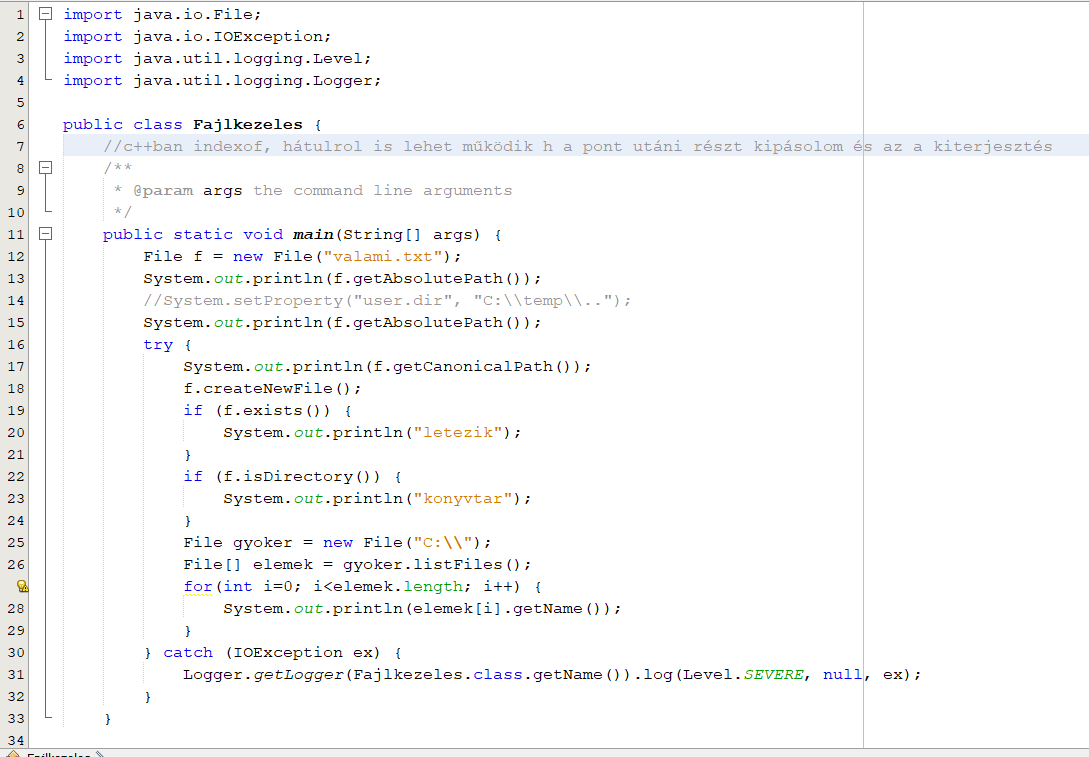
2018.10.10.: **Másoló-mozgató szemantika** feladat összegzése:

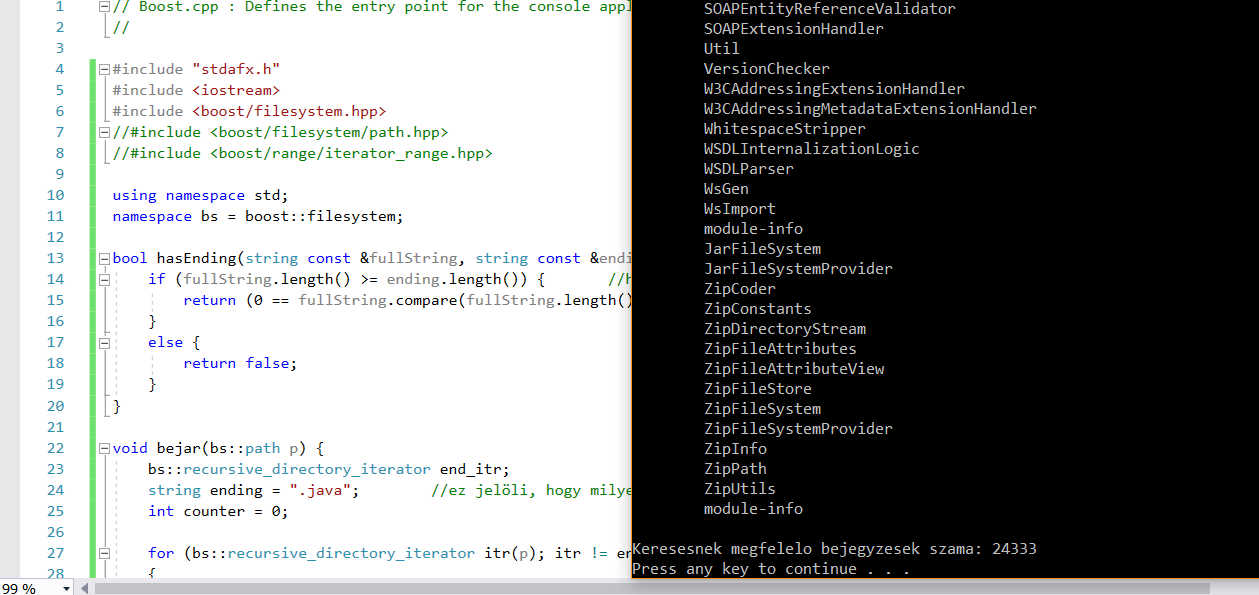
Ebben a feladatban a másoló és mozgató szemantikát hasonlítjuk össze példaprogramon keresztül.   
Másoló konstruktor és másoló értékadás:  
A másoló konstruktort leginkább arra használják, hogy egy osztályt egy már meglévő példány másolásával példányosítsanak. Ebben az esetben az eredeti és a másolt példány is azonos osztályba tartozik. A másoló értékadás ezzel szemben arra való, hogy egy már meglévő objektumnak új értéket adjunk, amely (az új érték) egy már létező objektum. A C++ egy osztály létrehozásakor alapértelmezetten generál másoló konstruktort és másoló értékadó operátort, ha ezeket nem definiáltuk direkt módon. Ezek a fordító által generált metódusok „alacsony” másolatokat készítenek, amely problémát okozhat, ha dinamikus memóriakezelést alkalmazó osztályokkal dolgozunk. Ezért azon osztályok esetében, amelyek dinamikus memóriakezelésűek, érdemes ezeket a metódusokat felülírni, hogy mélymásolatokat készítsünk.  
A shallow copy vagy alacsony másolás esetében arról beszélhetünk, hogy nem történik konkrét másolás, csak az objektum referenciája lesz megosztva, a változó csak a memória címet kapja meg. Máshogy megközelítve a dolgot, ha egy objektumot alacsony másolással másolunk, a lemásolt objektum lényegében csak egy hivatkozás (referencia) lesz arra az objektumra, amiről másoltuk. Mélymásolás vagy deep copy esetében viszont konkrét másolás történik, új memóriaterület foglalás megy végbe a háttérben. A mélymásolás értelemszerűen lassabb az alacsony másolásnál, viszont hasznos az esetlegesen felmerülő problémák kiküszöbölésére.

Mozgató konstruktor és mozgató értékadás:  
A C++11 két új függvényt definiál a mozgató szemantikával kapcsolatosan: a mozgató konstruktort és a mozgató értékadó operátort. Míg a másoló konstruktornak és a másoló értékadásnak az a célja, hogy másolatot készítsen egy objektumról egy másik objektumba, addig a mozgató konstruktor és a mozgató értékadás célja már az, hogy az erőforrások tulajdonjogát átruházza egyik objektumról a másikra (ez jóval kevésbé erőforrás igényes, mint egy másolat készítése).   
A mozgató konstruktorok és a mozgató értékadás definiálása hasonlóképpen működik, mint a másoló változataiknál. Habár míg a másoló függvényeinknél konstans referencia a paraméter, addig a mozgató függvényeknél nem konstans referenciánk van. Ez gyakorlatban úgy működik, hogy az eredeti objektum mélymásolása helyett egyszerűen átruházzuk (eltulajdonítjuk) az eredeti objektum erőforrásait. Ez magába foglalja az eredeti pointer alacsony másolását az objektumunkba, majd az eredeti pointer nullára állítását.  
Legtöbbször a mozgató konstruktor és a mozgató értékadás nem lesz definiálva alapértelmezetten, hacsak az adott osztály nem tartalmazza. Az alapértelmezett esetben a mozgató konstruktor és a mozgató értékadás ugyanúgy működik, mint a másoló konstruktor és értékadás, azaz másolatokat készít, de nem mozgat semmit sehova. Ebből következik, hogy ha azt akarjuk, hogy legyenek mozgató metódusaink, akkor ezeket meg kell írnunk magunknak.  
A kulcsa a mozgató szemantikának az l-value és az r-value használata. Ha létrehozunk egy objektumot vagy értéket adunk egy objektumnak l-value által, akkor az egyetlen dolog, amit tehetünk, hogy készítünk egy l-value másolatot. Nem feltételezhetjük, hogy biztonságos az l-value-t megváltoztatni, mivel lehetséges, hogy a későbbiekben még szükség lesz rá a programban. Hogyha van egy ” a=b ” kifejezésünk, akkor nem feltételezzük, hogy a b változni fog bármilyen módon.  
Ezzel ellentétben, ha létrehozunk egy objektumot vagy értéket adunk neki r-value formájában, akkor tisztában vagyunk vele, hogy az r-value csak egy ideiglenes objektum. Az ideiglenességből következik, hogy valamilyen formában szükség lehet ezen értékek megtartására. L-value referenciával megkötni csak abban az esetben lehet egy r-value-t, ha az konstans (az lvalue referencia használata esetén nem változtatható meg az rvalue). A másik lehetőség az r-value referencia alkalmazása, amely megköti az r-value-t, és ez a későbbiekben változtatható is lesz. Egy rvalue referencia is lehet konstans vagy nem konstans, de ritkán fogunk látni konstans rvalue referenciát, hiszen ennek pont az a lényege, hogy módosítható legyen.  
Visszatérve az r-value-k használatára, adott értékek másolása helyett (ami költséges lehet), a mozgató szemantika segítségével egyszerűen csak átadjuk az erőforrásait (ami olcsó) egy objektumnak, amelyre szükségünk van. Ez biztonságos, mivel az ideiglenes érték mindenképp törölve lenne a kifejezés végén, tehát biztos, hogy később ez már nem lesz használva. Összegezve a mozgató szemantika lehetővé teszi, hogy egy r-value erőforrásait átadjuk egy l-value-nak másolás nélkül.  
A C++11 az r-value referenciák által lehetőséget biztosít arra, hogy különböző viselkedéseket kreáljunk r-value és l-value alkalmazása esetén. Ez biztosítja, hogy az objektumok kezelése egyre hatékonyabbá váljon.

A feladathoz kapcsolódó esszé mellett egy mintaprogramot is elkezdtem, de még a mozgató konstruktor nem a feladatnak megfelelően funkcionál.

2018.11.18: A feladathoz kapcsolódó mintaprogramot is sikerült befejezni. Ebben láthatjuk a másoló és mozgató szemantika alkalmazását.  


2018.10.15: A **JDK osztályok** nevű feladathoz a fájlkezelés megismerése a programokban elengedhetetlen. Ennek láttam neki először egy Java programmal, majd ez után tervezem ezt átírni C++-ra.  


2018.10.30: A feladat megvalósítása C++ban. A program kilistázza az összes .java kiterjesztésű állományt a megadott könyvtárból.  


6. hét

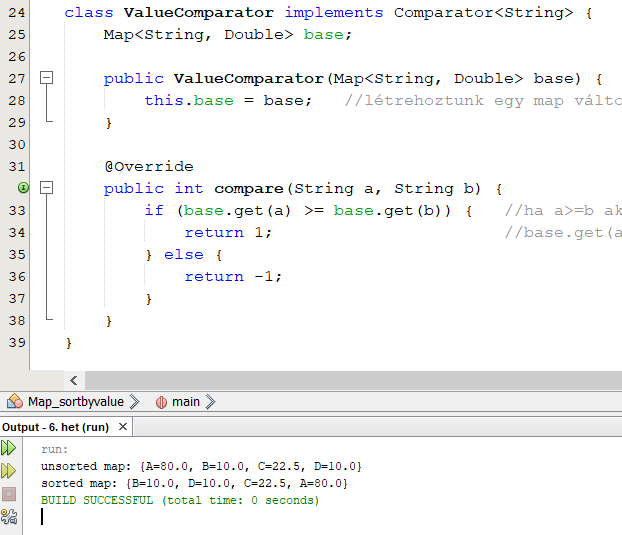
**Gengszterek**

**C++11 Custom Allocator**

**STL map érték szerinti rendezése** (tárgyhéten befejezve)

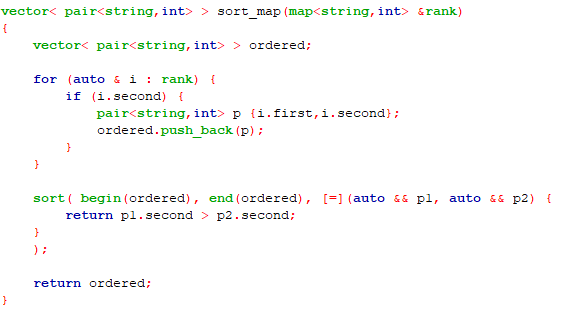
**Alternatív Tabella rendezése** (tárgyhéten befejezve) **Összefoglaló** (tárgyhéten befejezve)

2018.10.17.: Az **STL map érték szerinti rendezése** feladatot először másként értelmeztem, ezért az összegzés egy java programról készült.   
Ebben a feladatban a Map típusú adatok érték szerinti rendezése merült fel problémaként, erre szeretném bemutatni az én mintaprogramomat a probléma egy megoldásaként.  
Először is azzal kezdeném, hogy mi is az a Map? A Map egy olyan kollekció, amely kulcs-érték párokat tartalmaz. A Map nem tud tárolni duplikált kulcsokat, ezért minden esetben egy kulcshoz csak egy érték rendelhető. Alapértelmezetten a Map adatstruktúra nem rendezett, azonban vannak továbbgondolt, rendezett változatai. Ilyen például a HashMap és a TreeMap, amelyek a kódban is megjelennek.  
A HashMap adatstruktúra a Map-et veszi alapul és ezt kapcsolja össze egy hash táblával. Ebben az esetben is kulcs-érték párokat tárolunk, ahol a kulcsoknak egyedinek kell lennie. A kulcs lehet nulla, de csak egy esetben, míg az értékek tetszőleges számban vehetnek fel nulla értéket. Sajátossága, hogy a kulcsok sokkal könnyebben megtalálhatóvá válnak a hash függvény segítségével (a Map-pel szemben) annak ellenére is, hogy az adatok nincsenek rendezve.   
A rendezés kérdését a TreeMap adatstruktúra fogja megoldani. Ahogyan a neve is utal rá, ebben az esetben a kulcs-érték párokat fába rendezzük a kulcsok segítségével. Ez a HashMap-nél is gyorsabb keresést tesz lehetővé az adatok között.  
A mintaprogramra visszatérve, az alap ötlet az, hogy létrehozunk egy rendezetlen és egy rendezett map-et, feltöltjük a rendezetlent tetszőleges értékekkel, majd ezeket valamilyen módszerrel rendezzük a rendezett map-be. Ahogyan a kódban is láthatjuk, ez gyakorlatban úgy néz ki, hogy létrehozok egy map nevű HashMap-et és egy sorted\_map nevű TreeMap-et, majd ezeket rendezem egy bvc nevű Comparator segítségével. Azért választottam ezeket az adatstruktúrákat, mert a bemenő adatok rendezetlenek, így hasznos, hogy a HashMap-ben az adatok a beviteli sorrendben lesznek tárolva. A rendezett kulcs-érték párok esetében pedig a TreeMap struktúrának van egy alapértelmezett rendezési elve (kulcsok szerint rendez), ezért itt látványos lesz, hogy sikerült-e a rendezés.   
A Comparator a Java Collection osztály része, amely azt a célt szolgálja, hogy egy saját komparátor segítségével a Collection.sort metódust személyre szabhatjuk egyéni igények szerint. Ez különösen előnyös, mivel bizonyos adatstruktúrák rendezését így teljes mértékben mi irányíthatjuk, ha szeretnénk; illetve olyan kollekciók esetében is létrehozhatunk rendezéseket, amelyeknek alapértelmezetten nincsen. Ezen feladat esetében is ez a helyzet, hiszen a HashMap nem rendezett, viszont a komparátor segítségével rendezni tudjuk. A komparátort a konstruktorával létrehozzuk, amely során átadjuk neki azt a változót, amelyhez kapcsolódóan a komparátort használni szeretnénk. A rendezett kollekció eléréséhez még szükség van arra, hogy a definiálásánál átadjuk a TreeMap struktúrának mi alapján szeretnénk rendezni az adatokat, ezt meg is tesszük (átadjuk bvc komparátort), ahogyan a kódban is látható. A programban ezek után már csak át kell adnunk a teszt értékeket a map változónak és kiíratni a map, illetve a sorted\_map tartalmát.   
A rendezés azonban ennél kicsit több magyarázást, illetve a működéséhez egy további osztályt igényel. A ValueComparator osztályban létrehozunk egy base nevű Map típusú változót, amelybe később a konstruktor által átadott Map-et tesszük. Ezek után a kód leglényegesebb pontja, az összehasonlítás következik. Az eredeti compare metódust felülírjuk, így ez két String objektumot fog kapni bemenő adatként. A két átadott String a két összehasonlítani kívánt Map elem kulcsa lesz. A get() metódussal megkapjuk a Map kulcsaihoz tartozó értékeket és ezeket hasonlítjuk össze. Ha az első érték nagyobb-egyenlő a másodiknál, akkor -1-et kapunk vissza, egyéb esetben pedig +1-et, ezáltal az értékeket növekvő sorrendbe rendezzük.  
Kimenetként láthatjuk a rendezés végeredményét és azt, hogy a map kulcs-érték párjai most már érték szerint növekvő sorrendbe rendezettek a sorted\_map kollekcióban.

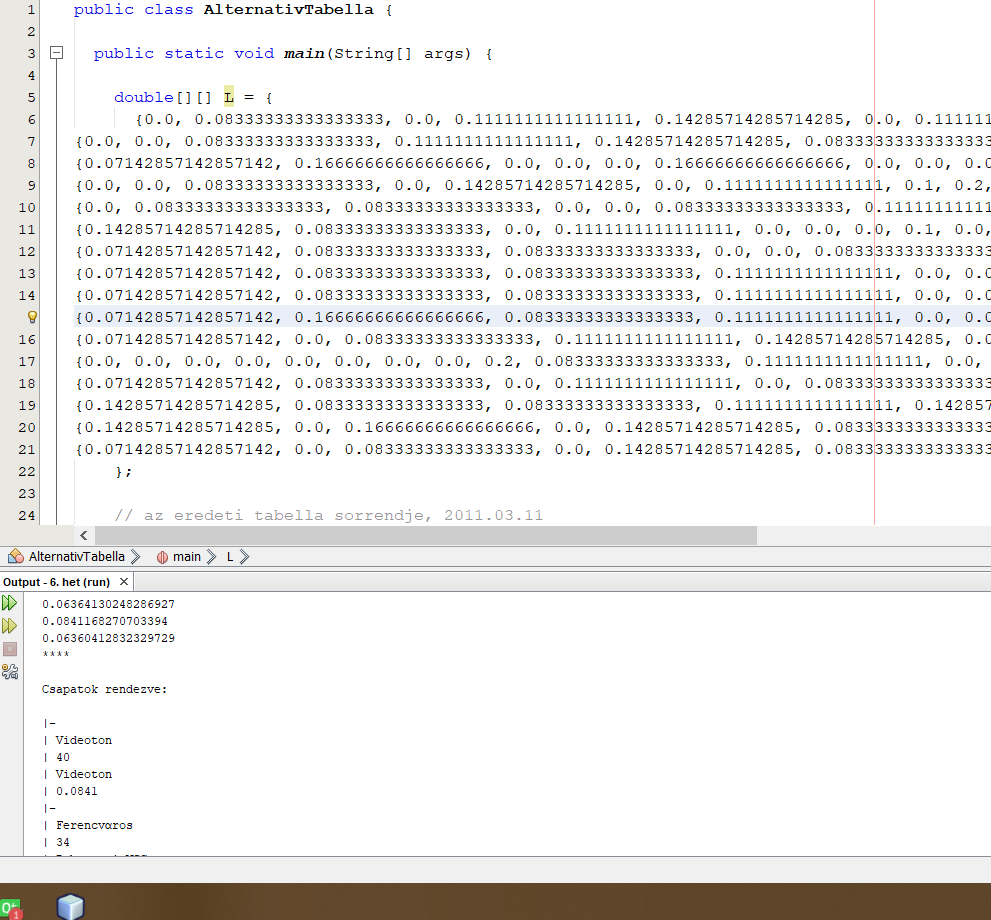


A feladat helyes megoldása a következő lett volna: A feladat kapcsán segítségemre volt: Ádám Szilárd

A mintaprogramban a rendező algoritmus egy map-et, azaz string-int kulcs-érték párokat kap meg bemenetként, majd ezeket egy string – int pair-ekből álló vektorba rendezi. A programban először is a rendezni kívánt adatok számára létrehozunk egy string – int pair-ekből álló vektort, ez lesz az ordered nevezetű vektor. A mintaprogram szerint amennyiben a bemenetként kapott map elemeknek van második, azaz egész szám értéke, akkor ezeket belerakjuk az ordered vektorba, mint pair típusú adatok. Maga a sort függvény lesz az, ami rendezi a vektort. Itt a paraméterekkel megadjuk, hogy honnan kezdve, illetve meddig rendezze a vektort, majd egy lambda kifejezést adunk meg még paraméterként. A lambda kifejezés lényege, hogy az ez által megadott függvényt nem definiáljuk külön, mivel csak egy esetben van rá szükségünk. Ennek felépítése úgy néz ki, hogy []-ben megadjuk, hogy a későbbiekben megadott külső változókat referencia (&) vagy érték(=) szerint szeretnénk kezelni, jelen esetben az érték szerinti kezelést választottuk. Ezek után pedig a bemenő változókat definiáljuk, itt a p1 és p2 az ordered vektor egy-egy pair-ei lesznek; az auto kulcsszóval azt határozzuk meg, hogy a p1 és p2 változók típusát nem mi adjuk meg, hanem a program automatikusan ismerje fel. A && karakterek azt jelölik, hogy ebben az esetben a p1 és a p2 rvalue-k lesznek. Ez azt jelenti, hogy a két változó nem kötődik semmilyen azonosítóhoz, ezek csak ideiglenesen tárolt értékek lesznek. A sort függvény a return-ben megadott reláció alapján rendezi a vektort, majd a függvényünk ezt vissza is adja visszatérési értékként.



2018.10.18.: Az **Alternatív tabella** feladat magyarázata:

A Csapat osztálynál implementált Comparable<Csapat> interface miatt a Csapat osztályban kötelező implementálnunk a public int compareTo(Csapat csapat) metódust. Ennek segítségével a „rang” oszlop értékei szerint kerülnek rendezésre a csapatok a tabellában. Ezt a metódust használja majd fel a java.util.Collections.sort(rendezettCsapatok) utasításban a Collection osztály sort metódusa. Ez a kollekciót a compareTo() metódusban meghatározottak alapján rendezi. A java.util.Collections.reverse(rendezettCsapatok) metódus segítségével pedig az így kapott rendezett kollekcióban található elemek sorrendjét a fordítottjára állítjuk.  


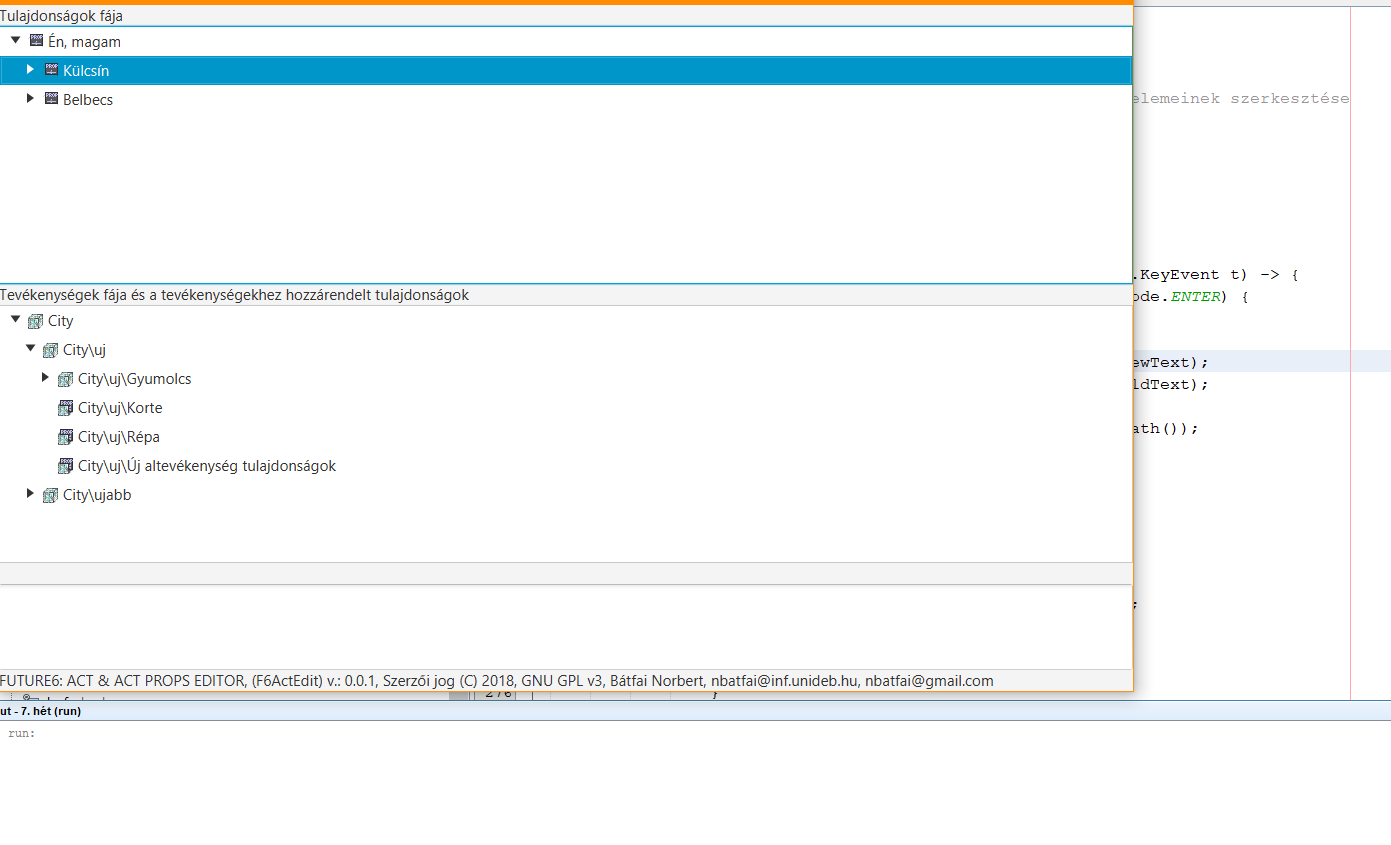
7. hét

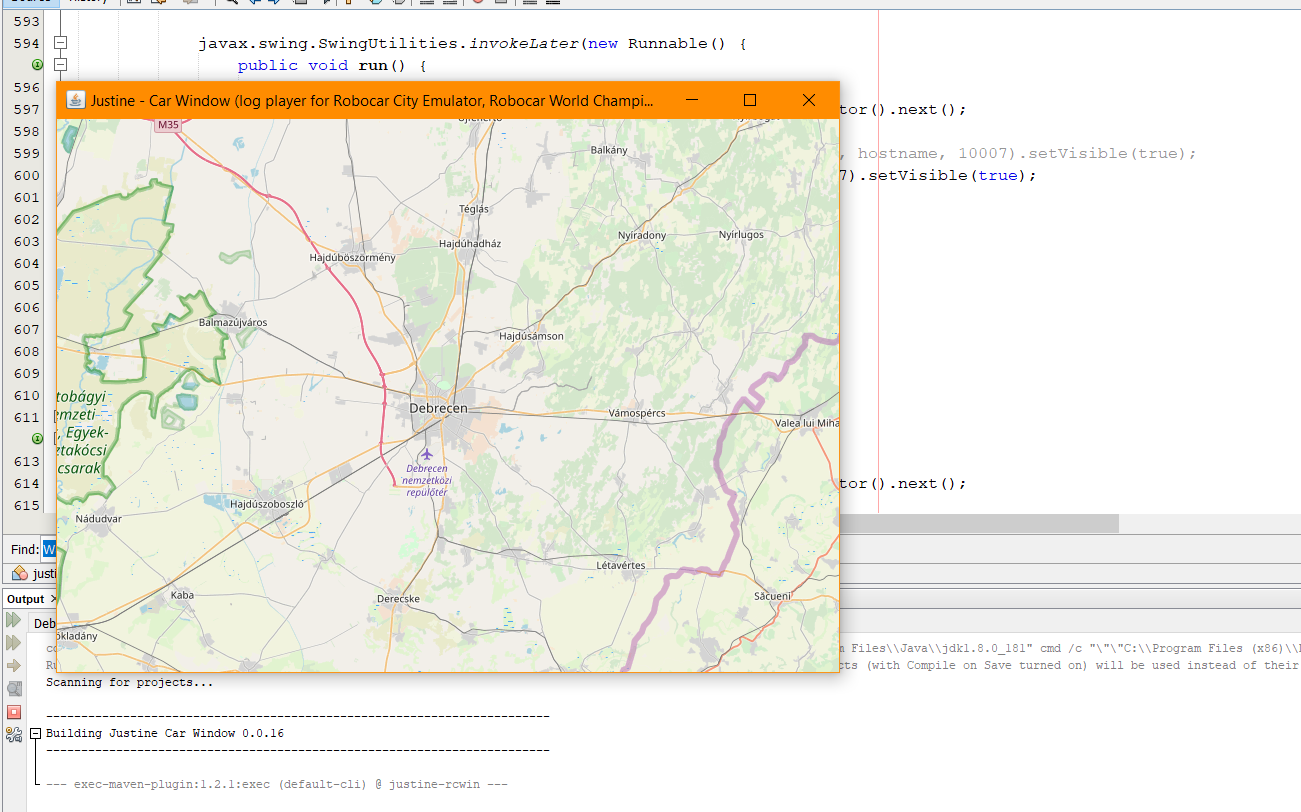
**FUTURE tevékenység editor** (tárgyhéten befejezve)

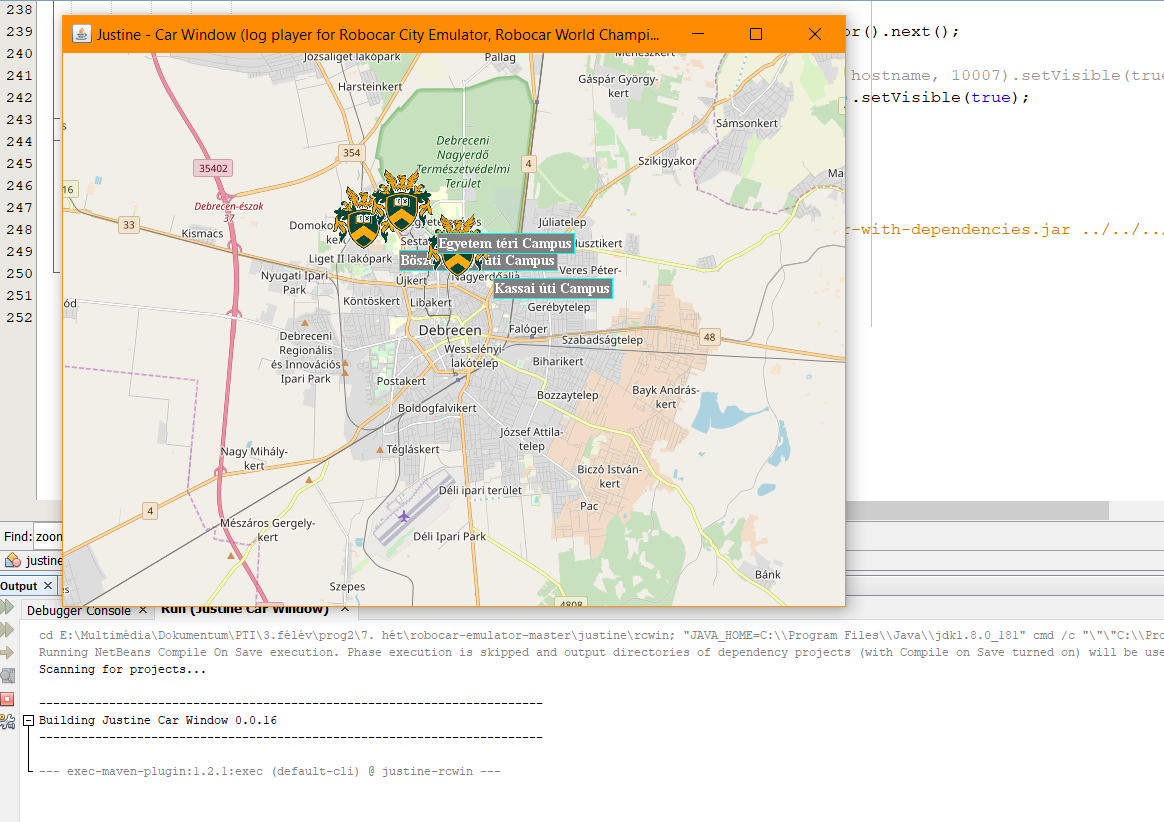
**OOCWC Boost ASIO hálózatkezelése**

**SamuCam**

**BrainB** (tárgyhéten befejezve) **OSM térképre rajzolása** (8. héten befejezve)

2018.10.30: **FUTURE tevékenység editor**: A feladatnak megfelelően változtattam a programon, az én javításom az, hogy a tevékenységfában egy elem módosítása esetén nem egy új fájlbejegyzés kerül létrehozásra, hanem a régit nevezi át.

2018.11.03: **OSM térképre rajzolása** feladat kapcsán a térkép megjelenítéséig jutottam. A megadott webcímet nem lehetett elérni, ezért az OSM honlapját használtam. A térképen a desztinációk megjelenítése még nem működik.  


2018.11.08: Sikerült a térképen a megjeleníteni a DE campusainak helyét. Hostként a [www.openstreetmap.hu](http://www.openstreetmap.hu) weblapot használtam, illetve mivel ezen helyek koordinátái fixek, nem fog változni a helyük, ezért ennek megfelelően töltöttem fel a waypoints Set-et.  


2018.10.4: A **BrainB** feladat az volt, hogy mutassuk be a Qt slot-signal mechanizmust ebben a projektben. A Qt slot-signal mechanizmus feladata alapvetően az, hogy widgetek, folyamatok egymás változásáról értesüljenek. Erre a problémára eddig több megoldást is használtak, pl. pointereket,azaz callback-eket, de a Qt-n belül ezt máshogyan oldották meg. A slot-signal mechanizmus úgy működik, hogy egy esemény bekövetkezésekor az egyik folyamat signal-t küld, amelyet egy másik folyamat slot-ja fog elkapni. A slot az a függvény, amely a signal érkezésekor kerül meghívásra válaszként. A connect() az a metódus, amellyel signal-slot párokat definiálhatunk.  
A BrainB feladat esetében a BrainBThread nevű szál heroesChanged függvényének meghívásakor érkezik a signal a BrainBWin folyamat slot-jára, az üzenetre a válasz pedig az updateHeroes metódus lesz. A heroesChanged függvény által indítjuk a signal-t, ez a draw() metódus meghívása esetén megy végbe, amely pedig a run() metódus része. A run() a brainBThread->start(); által van meghívva. A brainBThread szál a BrainBWin példányosításával jön létre, mivel a szál létrehozása a BrainBWin konstruktorában történik meg.  
A másik signal a brainBThread endAndStats metódusának meghívásakor jön létre, amely a BrainBWin slot-jára érve meghívja az endAndStats függvényét a BrainBWin osztálynak. Ezzel néhány debug üzenetet váltunk ki, elmentjük a képet, majd bezárjuk az alkalmazást.

8. hét

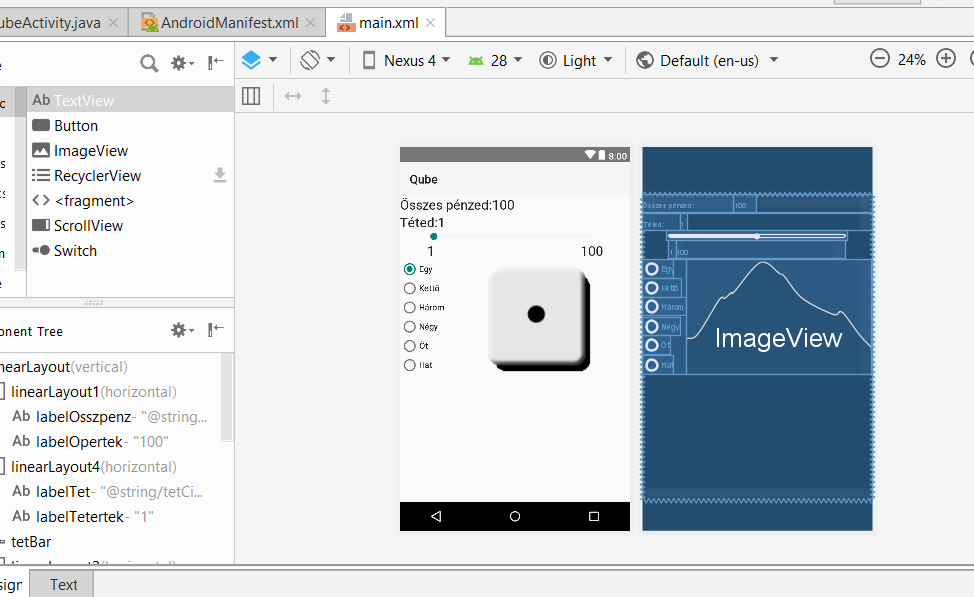
**Port scan** (tárgyhéten befejezve)

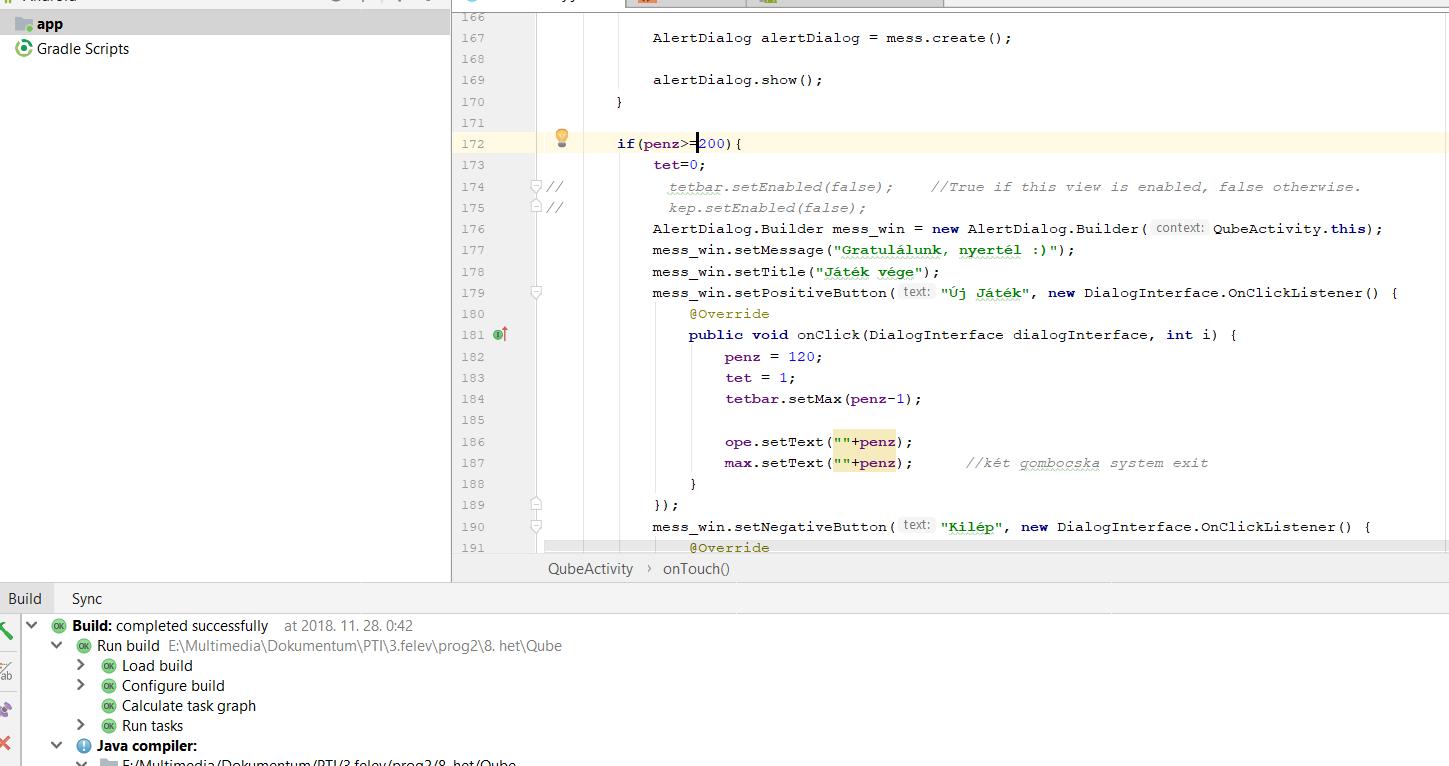
**AOP**

**Android Játék** (tárgyhéten befejezve – 10. héten módosítva)

**Junit teszt  
Összefoglaló** (10. héten befejezve)

2018.11.08: A **Port scan** példában socketeket hozunk létre a megadott paraméterek szerint. A domain nevet a megadott argumentumból, a portszámot pedig a ciklusból kapja meg. Abban az esetben, ha nem tud létrejönni a socket, akkor egy UnknownHostException vagy IOException keletkezhet, ami azt jelenti, hogy vagy a hostot nem tudja elérni vagy az adott port nincs nyitva a szerveren. Ebben az esetben lesz végrehajtva a kivételkezelés catch ága. A példában mindenfajta kivételt egységesen kezelünk, de akár külön is kezelhetnénk őket a kivétel típusa szerint. Összegezve ez egy port scannelő alkalmazás, amely kihasználja, hogyha nem sikerül socketet létrehozni az adott szerver adott portjára, akkor kivételt dob, ezáltal feltérképezi a host nyitott portjait.

2018.11.12: Az **Android Játék** feladat kapcsán én egy kockadobós játékot készítettem.



Az eheti feladatok egyike egy egyszerűbb Android játék megírása volt. Nekem is Androidos telefonom van, ezért nagyon sok ilyen alkalmazást láttam és használtam már, de ez volt az első, amelyet nem csak készen láttam, hanem én készíthettem el.

Manapság nagyon elterjedtek az Android operációs rendszerű telefonok. A NetMarketShare felmérései szerint 2018-ban az emberek 70% Android operációs rendszerű telefont használ. Az Android alapvetően Linux kernelt használó mobil operációs rendszer, amelyet elsősorban érintőképernyős mobil eszközökre (okostelefon, táblagép) terveztek. Abból eredően, hogy az Android a Google tulajdonában van, nagy előnye, hogy a Google fiókba való bejelentkezés után a telefont szinkronizálhatjuk a Google szolgáltatásokkal, ilyen például a névjegyzék, a levelezés vagy a naptár. A Google az Androidos alkalmazások készítéséhez kifejlesztette az Android Studiot, amely nagy előrelépést jelentett az Android alkalmazások írásában.  
Egy Android alkalmazás több komponensből épülhet fel, ilyen az Activity, amely úgymond a programnak egy ablaka, a Service, amely egy háttérben futó alkalmazás, a Content Provider, amely az adatok kezeléséért felelős, és a Broadcast Receiver, amelynek feladata a rendszerszintű eseményekre reagálás.   
Az Android operációs rendszer multiprocessz rendszer, azaz egy alkalmazás egyidejűleg több komponenst futtathat (Activity vagy Service) és az operációs rendszer több alkalmazást futtathat. Az alkalmazásoknak a háttérben megváltozhat a viselkedése, telefonhíváskor vagy más eseményre felfüggeszthetik működésüket, stb. Minden komponensnek különböző szerepe van az alkalmazáson belül. Bármelyik komponens önállóan aktiválódhat. Akár egy másik alkalmazás is aktiválhatja az egyes komponenseket.  
Az Activity-k jellemzője, hogy egyszerre mindig csak egy látható, de egy alkalmazáshoz több ilyen képernyőkép tartozhat, amelyeket futás közben – események hatására – szabadon cserélhetünk. Minden programnak kell legyen egy belépési pontja, amely az az Acitvity, amelyet először mutat meg a felhasználónak. Minden Activity az android.app.Activity osztályból származik le. Egy Activity-nek több állapota lehet, ilyenek az onStart(), onPause(), onResume() és az onStop() többek között.  
Sok alkalmazás igényli, hogy bezárt ablakkal is képes legyen futni a háttérben, ezt szolgáltatásként megteheti, egyszerűen kell egy Service osztályból leszármazott példány, így marad a programunknak olyan része, amely a felfüggesztett Activity esetén is fut (gondoljunk például egy média lejátszóra).   
Általában minden alkalmazás tárol adatokat két futás között, hiszen ha felfüggesztés helyett bezárnánk az alkalmazást, akkor elvesznének az addig összegyűjtött adatok. A Content provider (tartalom szolgáltató) komponens feladata egy megosztott adatforrás kezelése. A platformon lehetőségünk van állományokba vagy SQLite adatbázisba menteni adatokat, és ezt segíti a Content Provider, illetve lehetővé teszi, hogy két alkalmazás adatokat cseréljen egymással. A Content provider-en keresztül más alkalmazások hozzáférhetnek az adatokhoz, vagy akár módosíthatják is azokat.   
A Broadcast receiver komponens a rendszer szintű eseményekre reagál. Ilyen például, ha kikapcsolt a képernyő, alacsony az akkumulátor töltöttsége, elkészül egy fotó, bejövő hívás, stb. Az alkalmazás is indíthat saját „broadcast”-ot, például ha jelezni akarja, hogy valamilyen művelettel végzett (pl. letöltődött a torrent). Nem rendelkeznek saját felülettel, inkább valamilyen figyelmeztetést írnak ki például a status bar-ra, vagy elindítanak egy másik komponenst (jeleznek például egy service-nek).

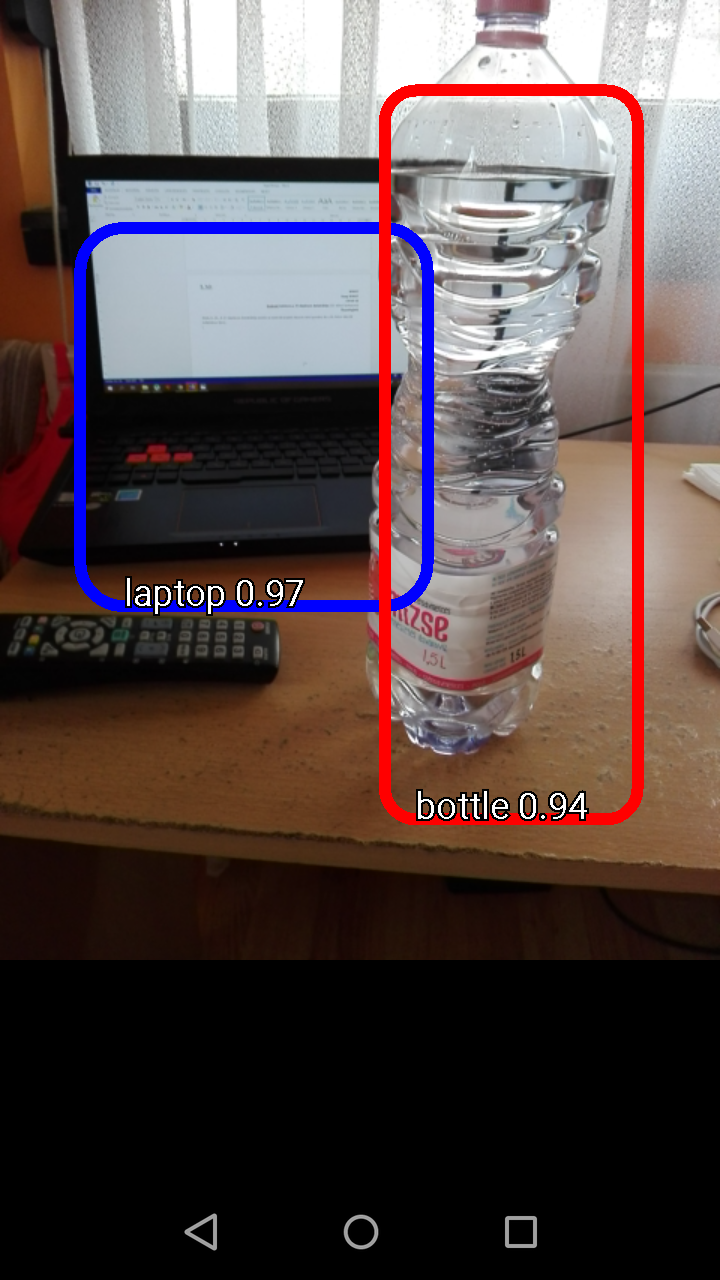
Az általam elkészített alkalmazás egyetlen Activityből áll. A játék egy kockadobálós játék, amelyben a játékos megtippelheti, hogy hanyast fog dobni. Helyes találat esetén a tét kétszeresét kapja a játékos, ellenkező esetben pedig a tét levonásra kerül a pénzéből. A játéknak akkor van vége, ha a játékosnak elfogyik a pénze. Ennek a játéknak a megvalósítása a következőképpen alakult.  
Elsőként az aktuális pénz megjelenítése került sorra, ezt ahogyan a képen is láthatjuk „Összes pénzed: 100” formátumban írja ki a játék. A játék kezdetén a játékos 100 pénzzel kezd, amely jelen állapotban akármeddig gyarapodhat a játék során.  
A következő a tét kiválasztása. Ehhez egy SeekBar-t használtam, amelyen megadhatja a játékos, hogy mennyi pénzt szeretne feltenni az általa választott tippre. Ennek értékét a csúszka felett szövegesen is láthatjuk. A maximálisan választható tét a játék során mindig a játékos pénzével egyenlő. A tét beállítása mellett egy setOnSeekBarChangeListener is beállításra került a csúszkán történő változások figyelése érdekében.  
A játékos a tippet, amelyet meg szeretne játszani RadioButton-ok által prezentált gombok segítségével választhatja ki. Az összes gomb egy RadioGroup-on belül van, ez azt a célt szolgálja, hogy egyszerre csak egy gombot lehessen kiválasztani.  
Ezek után jön a játék lényege, maga a dobás. A dobás értékének megfelelően megtörténik a pénz növelése, illetve csökkenése.  
Amennyiben a pénz nullára csökkent, megjelenik egy üzenet, hogy a játékos vesztett, majd választhat, hogy új játékot szeretne vagy kilépni. Ezt AlertDialog formájában oldottam meg, amely esetén a positive és a negative button-hoz kapcsoltam a kilép és az új játék lehetőségeket.

9. hét

**MNIST**

**Deep MNIST**

**CIFAR-10  
Android telefonra a TF objektum detektálója** (10. héten befejezve) **Összefoglaló** (10. héten befejezve) **Beszámítható kisfeladat**

2018.11.24.: A **TF objektum detektálója** esetén az Android projekt okozott némi gondot, de a 10. hétre sikerült működésre bírni.

2018.11.26:   
Az eheti feladatok középpontjában a TensorFlow megismerése és ennek használata áll. A feladatok kapcsán a Python nyelv alkalmazása került előtérben, mivel a Python interfésszel szemben a TensorFlow C++ interfésze fejletlen. Az összefoglalómban a Python, illetve a TensorFlow szerepét, előnyeit részletezném kicsit, illetve a TF Objektum detektálójára térnék ki néhány gondolat erejéig.

Először is a Pythonnal kezdeném. A Python a leggyorsabban fejlődő nyíltforrású,  általános célú objektum orientált programozási nyelv. A programozók 14 %a használja. A Python nyelv jelenleg Tiobe Index alapján a 4. legnépszerűbb programozási nyelv. A nyelv legfőbb ismérve és előnye, hogy szintaxisa egyszerű, emiatt könnyen olvasható és könnyen tanulható. Vállalati projektek esetén lényeges szempont, hogy Open Source forráskódú, tehát bárki szabadon használhatja, módosíthatja és terjesztheti. Egyaránt működőképes Linux, MacOS és Windows rendszereken is, így a programozók bármely operációs rendszer alatt dolgozhatnak Python projektjeiken. A többi nyelvhez – főként C/C++, Java – képest háromszor-ötször rövidebb kódokat kaphatunk a nyelv által, a fejlesztési időt akár tizedére csökkentheti és lényegesen egyszerűbb a karbantartása is a Python nyelven írt programoknak. Fejlesztői szepontból tekintve a kérdést, nagy könnyebséget jelent, hogy nincs szükség jelen esetben erőforrás kezelésre külső beavatkozás által, nincsenek pointerek, pozitívum viszont, hogy a Python többszálú folyamatok létrehozására is alkalmas. A nyelv objektum orientált, támogatja a többszörö öröklődést és az operátor túlterhelést is. A Pythonba, mint a Javaba vagy a C++ újabb verzióiba, egy kivételkezelő rendszer van beépítve, ami lényegesen leegyszerűsíti a hibakezelést. Többek között ezen előnyök miatt lehetséges, hogy a Python egészen a 4. helyig jutott a raglétrán.

Ezek után következzen a TensorFlow, amely kapcsán a Python előkerült. A TensorFlow-t a Google Brain csapata fejlesztette ki eredetileg belső alkalmazásra 2015-ben. A TensorFlow nyílt forráskódú szoftverkönyvtár adatfolyam-programozáshoz. Ez egy szimbolikus matematikai könyvtár, amelyet főként a gépi tanulásos alkalmazásokhoz használnak, például neurális hálózatok esetén. A Google-nél egyaránt kutatások és gyártási folyamatok során is alkalmazzák. A TensorFlow elérhető 64 bites Linux , macOS , Windows és mobil platformokon egyaránt, ide értve az Android-ot és az iOS-t is. Rugalmas architektúrája lehetővé teszi a számítások egyszerű elvégzését különböző platformokon (CPU-k, GPU-k, TPU-k) asztali számítógépről a kiszolgálók és mobil eszközök között. Ezek a számítások állapotfüggő adatfolyam-grafikonok .   
A TensorFlow név olyan műveletekből származik, amelyeket a neurális hálózatok multidimenzionális adatrácsokon hajtanak végre. Ezeket a tömböket nevezik tenzoroknak. 2016 májusában a Google bejelentette, hogy a TensorFlow-ra szabott Tensor feldolgozóegységet (TPU) , egy specifikusan integrált áramkört (egy hardveres chipet) fejlesztettek ki, amely inkább a modellek használatára vagy futására irányul, nem pedig képzésükre . Később bejelentették, hogy több mint egy éve működtettek TPU-kat az adatközpontjaikon belül, és azt találták, hogy egy nagyságrenddel jobb teljesítményt nyújt a gépi tanuláshoz.   
2018. március 1-jén a Google kiadta a Machine Learning Crash Course (MLCC) programot. Eredetileg a Google munkatársainak a gyakorlati mesterséges intelligenciával és a gépi tanulással kapcsolatos fundamentumaihoz való felkészítését tervezték. A Google a világon számos városban kifejlesztette szabad TensorFlow műhelyeit, mielőtt végül kiadta a tanfolyamot a nyilvánosság számára. A nyilvánosságra hozatal által lehetőség nyílt sokak számára a TensorFlow használatára, többek között magánvállalkozásoknak, fejlesztőknek és nekünk diákoknak is.

Az általam választott feladat a TF Objektum Detektálójának felélesztése és kipróbálása volt. Az Android projekt által 4 alkalmazás került feltelepítésre a telefonomra, a TF Classify, a TF Detect, a TF Stylize és a TF Speech.  
A TF Classify az Inception v3 modell segítségével a telefon kamerájának látószögébe kerülő tárgyakat az Imagenet egy osztályába sorolja és ezt az osztályt jeleníti meg a felhasználó számára. Az Imagenet osztályban csak 1000 kategória van, ezáltal sok hétköznapi tárgy kimarad a besorolásból, és sok olyan tárgyat is felismer az alkalmazás, amely nem valószínű, hogy megfordul az ember kezében. Ennek következtében könnyen megeshet, hogy az eredmény valóságtól való eltérése kifejezetten szórakoztató.  
A TF Detect egy multibox modellt használ, amellyel próbálja bekeretezni a a felismert tárgyat a helyének megfelelően. Az eredmény még nem tökéletes, de a tárgyfelismerés jelenleg is egy aktív kutatási terület.  
A TF Stylize egy real-time stílus átalakító. A felhasználó kiválaszthatja melyik stílusban szeretné látni a kamera képét, de akát keverheti is a stílusokat. A stíluson kívül a felbontás is állítható, amely a stílusátalakítás gyorsítását, illetve lassítását okozza.  
Végül, de nem utolsó sorban pedig a TF Speech, amely a megadott szavak közül felismeri azt, amelyet a felhasználó kimond és ezzel egyidőben felvillan az a szó, amely elhangzott.

Tapasztalataim szerint a 4 alkalmazás közül 3 viszonylag jól működött, de a Stylize használhatatlanul lassú volt. Ahogy fentebb is említettem, a Objektum Detektáló érdekes eredményeket adhat vissza, ilyen volt például a bögreként felismert telefontöltő vagy a kötszernek titulált zsebkendő. Pozitívum volt számomra, hogy ténylegesen nagyjából real-time-ban ismeri fel a tárgyakat. Kíváncsi lennék ezen alkalmazások későbbi fejlettebb változatára is, hiszen sok esetben nagyon hasznos lehetne a hétköznapokban.