

SZAKDOLGOZAT

Pertic Szabaszián

2019

Pannon Egyetem

Műszaki Informatikai Kar

Rendszer- és Számítástudományi Tanszék

Programtervező Informatikus BSc

SZAKDOLGOZAT

Programozási feladatok gyakorlását vezérlő szoftver
készítése

Pertic Szebasztián

Témavezető: Orosz Ákos

2019

SZAKDOLGOZAT TÉMAKIÍRÁS

Pertic Szebasztián

Programtervező informatikus BSc szakos hallgató részére

Programozási feladatok gyakorlását vezérlő szoftver készítése

Témavezető: Orosz Ákos

A feladat leírása:

A szakdolgozat célja egy olyan szoftver elkészítése, amely az egyetemen oktatott programozás tárgyak gyakorlását segíti a hallgatók számára. A szoftver egy feladat-adatbázist kezel, aminek a feladatait a hallgatók meg tudják oldani, és a megoldás helyességét a szoftver automatikusan ellenőrzi. A szoftver tárolja hogy mely feladatokat sikerült már megoldani, ezen felül lehetőséget biztosít több különböző tantárgy kezelésére, illetve különböző programozási nyelvek használatára.

A munka elsődleges célja a szoftver elkészítése az első féléves, C alapú programozás tárgyakhoz.

Feladatkiírás:

- Tervezze meg a szoftver felépítését úgy, hogy az támogasson jövőbeni fejlesztéseket is, mint a különböző programozási nyelvek és kurzusok, több felhasználó, változatos feladatok.
- Valósítsa meg a szoftvert az első féléves, C alapú programozás tárgyakhoz.
- A szoftver képes legyen a feladatok megoldását felügyelni, a megoldás helyességét ellenőrizni.
- Készítsen interfészeket, amikkel a későbbiekben a szoftver egyszerűen kiegészíthető új programozási nyelvekkel, és a hozzájuk tartozó szabályokkal.
- Készítsen felületet a feladat-adatbázis feltöltésére.

Orosz Ákos

Tanársegéd

Témavezető

Süle Zoltán

egyetemi docens

szakfelelős

Veszprém, 2019. április 12.

Nyilatkozat

Alulírott Pertic Szebasztián hallgató, kijelentem, hogy a dolgozatot a Pannon Egyetem Rendszer- és Számítástudomány tanszékén készítettem a Programtervező Informatikus BSc végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy a dolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

Veszprém, 2019. május 10.

Aláírás

Alulírott Orosz Ákos témavezető kijelentem, hogy a dolgozatot Pertic Szebasztián a Pannon Egyetem Rendszer- és Számítástudomány tanszékén készítette a Programtervező Informatikus BSc végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozat védelemre bocsátását engedélyezem.

Veszprém, 2019. május 10.

Aláírás

Köszönetnyilvánítás

Köszönettel tartozom a témavezetőmnek, Orosz Ákosnak, aki segítőkészségével, és kiváló ötleteivel hozzájárult a munkám megvalósításához. Továbbá szeretnék köszönetet mondani feleségemnek a végtelen támogatásért, és a szakdolgozatom nyelvi lektorálásáért.

TARTALMI ÖSSZEFOGLALÓ

Mai digitalizált világunkban fontos, hogy a feltörekvő nemzedékek oktatása a legmodernebb technológiákkal történjen, kiváltképp az egyik leggyorsabban fejlődő területen, az informatika és programozás területén.

A szakdolgozat célja egy olyan, a programozási feladatok gyakorlását segítő, Windows és Linux rendszereken egyaránt futtatható szoftver megalkotása volt, amelyben a feladatok kurzusokra bontva érhetők el így igazodva az egyetemi tananyaghoz. A szoftver lehetőséget biztosít a hallgatóknak a feladatok megoldására, majd ezen megoldások helyességét automatikusan ellenőrzi a feladathoz tartozó tesztesetek segítségével. Mindemellett kezeli mind a feladatok, mind a témakörök egymásra épülését, valamint felületet biztosít az oktatóknak a feladatok egyszerű hozzáadására és módosítására.

A szoftver elkészítésénél fontos követelmény a jövőbeni fejlesztések támogatása, valamint hogy egyszerűen kiegészíthető legyen új programozási nyelvekkel és a hozzájuk tartozó szabályokkal. Ezen felül az elkészült munka rendelkezik minden szükséges funkcióval, amelyek segítségével ténylegesen alkalmazható a C programozási nyelvet oktató tantárgyak esetében.

Kulcsszavak: programozás oktatás, automatikus ellenőrzés, feladatbank, Java

ABSTRACT

In our modern digitized world, it is important to educate emerging generations with the latest technologies, especially in one of the fastest growing areas of IT and programming.

The aim of this thesis was to create a program that can be used on both Windows and Linux operating systems, that manages a task bank in which the tasks and exercises are divided into courses, which matches a semester on most of the universities. The software provides the students with the opportunity to solve the tasks and then automatically checks the correctness of these solutions with the help of test cases. Besides this functionality it also makes it possible for instructors to create tasks and define a relation on them easily.

When making this software, it was an important requirement that it's structure supports improvements and development in the future. It is possible, to easily add support for different programming languages, and their rules. In addition, the completed software has all the necessary functions that can be applied to the subjects that teach C programming language.

Keywords: programming education, automatic testing, task bank, Java

TARTALOMJEGYZÉK

1.	BEVEZETÉS	1
2.	A FELADAT ISMERTETÉSE	3
3.	HASONLÓ MEGOLDÁSOK ISMERTETÉSE	4
3.1	CODINGAME	4
3.2	EXERCISM	5
3.3	KONKLÚZIÓ	6
4.	ALKALMAZOTT TECHNOLÓGIÁK.....	7
4.1	JAVA.....	7
4.2	SWING	8
4.3	GRADLE.....	9
4.4	INTELLIJ IDEA.....	10
4.5	GIT	11
5.	TERVEZÉS.....	12
5.1	A SZOFTVER FŐBB LOGIKAI STRUKTÚRÁI	12
5.1.1	<i>Feladat</i>	12
5.1.2	<i>Feladatgyűjtemény</i>	13
5.1.3	<i>Mód</i>	14
5.1.4	<i>Kurzus</i>	14
5.1.5	<i>Felhasználó modul</i>	15
5.2	FELHASZNÁLÓI FELÜLET TERVE	15
6.	MEGVALÓSÍTÁS	17
6.1	FÁJL ALAPÚ ADATBÁZIS STRUKTÚRA	17
6.1.1	<i>Fájl struktúrák és kiterjesztések</i>	17
6.1.2	<i>Kurzusok fájl alapú tárolása</i>	20
6.1.3	<i>Feladatok fájl alapú tárolása</i>	21
6.1.4	<i>Feladatgyűjtemények fájl alapú tárolása</i>	24
6.1.5	<i>Módok fájl alapú tárolása</i>	26
6.1.6	<i>Felhasználók fájl alapú tárolása</i>	27
6.2	PROGRAMOK FORDÍTÁSA ÉS FUTTATÁSA.....	28
6.2.1	<i>Osztályok és interfészek</i>	29
6.2.2	<i>Streamek olvasása</i>	30
6.2.3	<i>Főbb osztályok, és Interfészek létrehozása</i>	31
6.2.4	<i>C programnyelv implementációja</i>	33
6.3	FELADATMEGOLDÓ FELÜLET MEGVALÓSÍTÁSA.....	35
6.4	FELADATKÉSZÍTŐ FELÜLET MEGVALÓSÍTÁSA.....	41
6.5	EGYÉB FUNKCIÓK	46
7.	ÖSSZEFOGLALÓ.....	47

1. Bevezetés

Mai gyorsan fejlődő világunkban, melynek mindennapjait átszövi a digitalizáció és az okos-eszközök használata, fontos, hogy a feltörekvő nemzedékek oktatása is ebben a szellemben történjen és az oktatás során a legmodernebb technológiákat, a modern kor vívmányait hívjuk segítségül, kiváltképp az informatika és programozás oktatás területén. Ez a terület napjainkban a többi tudományágnál jóval gyorsabban fejlődik, jóval fontosabb a naprakész tudás és a fiatalok bevonása, érdeklődésének felkeltése a szakma iránt.

Ez a szakdolgozat is ezt a célt hivatott szolgálni: egy, a programozás oktatása során otthoni gyakorlásra és órai feladatmegoldásra egyaránt alkalmazható szoftver elkészítésével. A szoftver önmagában képes különböző programozási kurzusok feladatait tárolni, megoldásukra és elkészítésükre felületet biztosítani, valamint a kész megoldást automatikusan ellenőrizni és kiértékelni. Mindezt egy felhasználóbarát kezelőfelületen keresztül.

A projekt a tervezés során a **Hades** fantázianevet kapta a görög istenség előtti tisztelgésért, aki, mint sokan talán nem is tudják, nem csak az alvilág istene volt, hanem a föld alatti kincsek, még fel nem tárt bányák istene is [7], így közvetve utalva a programozási készség elsajátítása révén elérhető kimeríthetetlen lehetőségek tárházára, ami egy teljesen új, izgalmas világot tár fel a programozni vágyó tanuló számára.



1.1 ábra: Hades [2]

Dolgozatom első része részletesen ismerteti a feladatot, és azt, hogy milyen követelményeknek és elvárásoknak kellett eleget tenni a megvalósítás során. A harmadik fejezet ismerteti néhány már fellelhető megoldást, szoftvert erre a problémára, amelyből ötleteket és tapasztalatokat gyűjtöttem a feladat megvalósításához. A negyedik fejezetben bemutatja a felhasznált technológiákat, és azok jellemzőit. Az ötödik fejezetben bemutatásra kerülnek a szoftver főbb építő kövei, a felépítésének összefoglaló, és részletes terve. Az ezt követő fejezetben szó esik a szoftver megvalósításáról, a felmerülő nehézségek és problémák megoldásáról, valamint a követelmények teljesítéséről. A dolgozatot egy összefoglaló fejezet zárja, mely még egyszer röviden összefoglalja dolgozatom tartalmát és kiemeli a lényegesebb pontokat.

2. A feladat ismertetése

A cél egy olyan szoftver elkészítése volt, amely elsősorban egyetemen oktatott programozás tárgyak oktatását, gyakorlását segíti a hallgatók számára. A szoftver egy fájl alapú feladatbankot kezel, aminek a feladatait a hallgatók meg tudják oldani, és a megoldás helyességét a szoftver automatikusan ellenőrzi. A szoftver tárolja, hogy mely feladatokat sikerült már megoldani, ezen felül lehetőséget biztosít több különböző tantárgy (kurzus) kezelésére, illetve különböző programozási nyelvek használatára.

A szoftver lehetőséget biztosít arra, hogy a szoftvert felügyelő oktató annak feladatbankját könnyen kiegészíthesse új kurzusokkal, feladatokkal egy felhasználóbarát felületen keresztül.

Elsődleges cél volt az első féléves, C alapú programozás tárgyakhoz a támogatottság elkészítése.

A szoftver elkészítése során figyelembe vett főbb tényezők, jellemzők:

- A szoftver felépítésének megtervezése úgy, hogy az támogat jövőbeni fejlesztéseket is, mint a különböző programozási nyelvek és kurzusok, több felhasználó, változatos feladatok.
- Interfészek készítése, amikkel a későbbiekben a szoftver egyszerűen kiegészíthető új programozási nyelvekkel, és a hozzájuk tartozó szabályokkal.
- A szoftver teljes megvalósítása az első féléves, C alapú programozás tárgyakhoz.
- A szoftver képes a feladatok megoldását felügyelni, a megoldás helyességét automatikusan ellenőrizni.
- Felület létrehozása a feladatbank feltöltésére.
- Platformfüggetlen működés, azaz Windows és Linux rendszereken is módosítás nélkül futtatható a program.
- Szintaktikai színezett kódszerkesztő felületek.
- Egymásra épülő feladatok és feladatcsoportok.
- Offline könnyen használható.

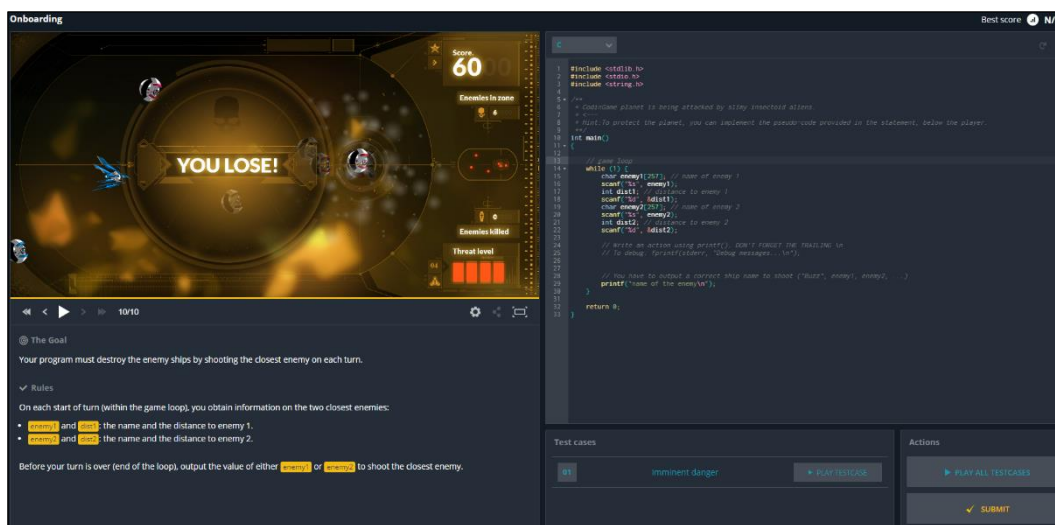
3. Hasonló megoldások ismertetése

Egy szoftver megtervezése illetve elkészítése előtt fontos feladat utánajárni a problémára már létező megoldásoknak, majd ezen megoldásokat tanulmányozni, azok felépítését megismerni. Fel kell mérni, hogy ezen megoldások miben hiányosak, mely követelményeket nem teljesítik a fontosabbak közül, illetve, hogy a követelményeken felül milyen egyéb szolgáltatásokkal bírnak. Ezekből az információkból ötletet merítve kezdhető el a szoftver megtervezése, és megvalósítása.

Olyan létező megoldások keresése volt a cél, amelyek a főbb követelményeket részben fedik. Fontos volt, hogy a szoftver képes legyen több feladat és nehézségi szint kezelésére, illetve az könnyen használható legyen egy kezdő számára is.

3.1 CodinGame

A CodinGame egy 2012-ben alapított online platform, amely egy webes felületen vehető igénybe. Több programozási nyelvet támogat, és egyre nehezedő feladatokon keresztül mutatja be a főbb programozási problémákat, amelyeket egy játékos, animált felülettel egészít ki. A felület továbbá versenyek lebonyolítására is képes, és „leaderboard” is megtalálható benne, amely a legjobb felhasználókat gyűjti össze.



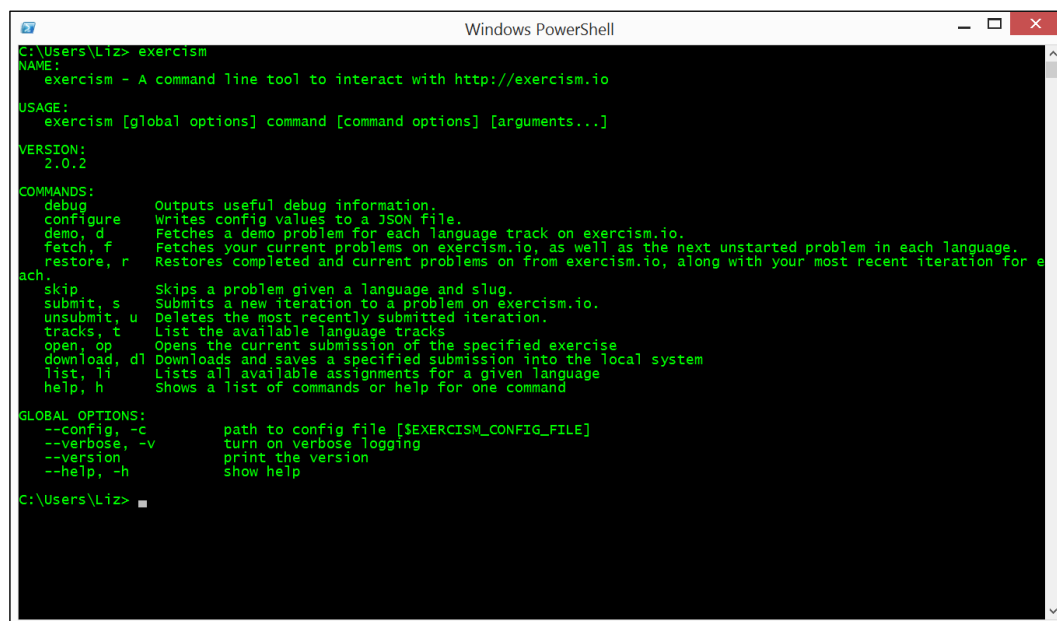
3.1 ábra: CodinGame online felülete (weboldal) [6]

Ez a megoldás állt legközelebb a főbb követelmények teljesítéséhez. További előnye még az, hogy a feladatok ellenőrzése közben animálva láthatóvá válik a program helyessége, ezzel izgalmasabb és látványosabb automatikus ellenőrzést biztosít. Két főbb követelményt viszont nem teljesít.

Nem használható offline módban, csak interneten, böngészőből érhető el. Ez akár előnynek is mondható, viszont használhatóság szempontjából limitált lehet mivel állandó internet kapcsolatot igényel. A második főbb probléma, hogy a feladat adatbázis nem szerkeszthető. Azon felül, hogy nem szerkeszthető a feladat adatbázisa, nem is egészíthető ki új feladatokkal, a feladat leírások csak angol és francia nyelven érhetőek el. A feladatok nincsenek kurzusokba vagy tantárgyakba csoportosítva, valamint oktatás szempontjából nagy hátrány az, hogy nem tudjuk úgy felhasználni, hogy az egy-egy egyetemi kurzus gyakorlására testre szabható legyen. A feladat adatbázis zárt, és ahhoz csak a felületet üzemeltetők férnek hozzá.

3.2 Exercism

Az Exercism [3] egy online felület, amely szintén egy weboldalon, illetve egy kliensen keresztül érhető el. A felület különbözik a legtöbb online kódoló platformtól, ugyanis itt a feladatot a saját számítógépünkön oldhatjuk meg. Számos programnyelv gyakorolható, számuk meghaladja az ötvenet. A főbb működési elve az, hogy egy-egy programozási feladatot letölthetünk a kliensen keresztül, azt megoldhatjuk a saját számítógépünkön, és a megoldást vissza feltölthetjük a kliensen keresztül. Ezt követően több más megoldást is megtekinthetünk. A feladat ellenőrzése és kiértékelése nem automatikus, azokat a felületen tevékenykedő mentorok bonyolítják le. Könnyen látható, hogy ez a megoldás lehetséges, hogy részletesebb visszajelzést küld mint egy automatikus kiértékelés, de lassú és nehézkes a gyakorlás, mivel várni kell a megoldások ellenőrzésére. A feladat adatbázis nem szerkeszthető, habár a felület forráskódja nyílt, de a hivatalos mentorok nélkül nem használható gyakorlásra.



```
C:\Users\Liz> exercism
NAME:
  exercism - A command line tool to interact with http://exercism.io

USAGE:
  exercism [global options] command [command options] [arguments...]

VERSION:
  2.0.2

COMMANDS:
  debug      Outputs useful debug information.
  configure  Writes config values to a JSON file.
  demo, d    Fetches a demo problem for each language track on exercism.io.
  fetch, f   Fetches your current problems on exercism.io, as well as the next unstarted problem in each language.
  restore, r Restores completed and current problems on from exercism.io, along with your most recent iteration for each.
  skip       Skips a problem given a language and slug.
  submit, s  Submits a new iteration to a problem on exercism.io.
  unsubmit, u Deletes the most recently submitted iteration.
  tracks, t  List the available language tracks
  open, op   Opens the current submission of the specified exercise
  download, dl Downloads and saves a specified submission into the local system
  list, li  Lists all available assignments for a given language
  help, h    Shows a list of commands or help for one command

GLOBAL OPTIONS:
  --config, -c      path to config file [$EXERCISM_CONFIG_FILE]
  --verbose, -v     turn on verbose logging
  --version         print the version
  --help, -h        show help

C:\Users\Liz>
```

3.2 ábra: CLI kliens az Exercism felülethez [5]

3.3 Konklúzió

A fent részletezett példák alapján belátható, hogy egy, az összes követelményt teljesítő megoldás nem lelhető fel, csak egy-egy részét oldják meg a problémának. A CodinGame nevű felület nagyon kiforrott, és professzionális megoldást nyújt, de annak feladat adatbázisa nem szerkeszthető, így oktatási célokra nehezebben használható fel. Fontos, hogy a tananyag, és ezzel együtt a gyakorló feladatok is személyre szabottan, az egyetemi tananyagnak megfelelő minőségben és nehézségben legyenek elérhetőek. További felületek is megvizsgálásra kerültek: CheckiO, Codewars, de ezek nem bizonyultak alkalmasnak arra, hogy kezdők is tudják használni egy-egy programnyelv megtanulására.

Már meglévő megoldások tanulmányozása értékes szemléletmódot és ötleteket adott a saját megoldásom megvalósítására.

4. Alkalmazott technológiák

A főbb követelmények ismertetése és a már meglévő megoldások tanulmányozása után kiválasztottam a megvalósításhoz használt technológiákat. Fontos tényező volt a programozási nyelv helyes megválasztása, valamint a projekt későbbi fejlesztését lehetővé tevő eszközök helyes alkalmazása, mivel a követelmények között kiemelt helyen szerepelt, hogy később további programnyelvek támogatását biztosítsa a szoftver.

4.1 Java

A szoftver elkészítése során fontos szempont volt, hogy az Windows és Linux rendszereken is jól működjön módosítás nélkül, azaz lényegében platformfüggetlen alkalmazás legyen. Mindkét operációs rendszer támogatása fontos szempont, hiszen a Windows és a Linux is széles körben elterjedt rendszer. A Linux napjainkban méltán örvend egyre nagyobb népszerűségnek és ment át észrevehetően gyors fejlődésen, valamint az egyetemi programozás gyakorlatok során is Linuxot használnak. Így a cél az volt, hogy a szoftver ne zárja ki egyik felet se a használói köréből, gyarapítsa akármelyik operációs rendszer kedvelőinek táborát.

Ezen feltételeket figyelembe véve, a **Java** programozási nyelv [1] lett a fejlesztés alapja, ami a több platformon való futtatáshoz a lehető legtermészetesebb feltételeket biztosítja.

A Java programnyelv egy általános célú, objektum-orientált programozási nyelv. Első verzióját a Sun Microsystems fejlesztette mely 1996-ban jelent meg. Ez a Java 1.0 verzió volt. Fő jelmondata a „Write once, run everywhere”, vagyis az „Írd meg egyszer, és futtasd mindenhol”, amelyet a Java nyelvet futtató JVM (Java Virtual Machine) tesz lehetővé. A JVM minden nevesebb platformra elérhető ingyenesen.

A JVM fő működési elve, hogy az úgynevezett Java bytekódot értelmezi, és az adott platformnak értelmezhető gépi kóddá alakítja azt át. A Java bytekód lényegében utasítások sorozata a Java virtuális gépnek, amely Java-ban írt programkódból, vagy más JVM-re tervezett programnyelv kódjából is képezhető.

Mivel a bytekód nem direkt az adott platform által van értelmezve, hanem először azt a JVM értelmezi és alakítja át az adott platformnak megfelelő natív gépi utasítások sorozatává, felmerülhet a kérdés, hogy ez nem jár-e teljesítményvesztéssel. Ez igaz is volt a korábban kiadott verziókra, az újabb implementációkban már több fejlesztés is történt ez ügyben, így a legtöbb esetben már nem lehet jelentős különbséget találni a JVM-en futó kód, illetve a natív gépi kód futási sebessége között. Egy nem kritikus rendszernél ez a különbség elhanyagolható.

A platformfüggetlen futási lehetőség a Java nyelv egyik legnagyobb erőssége, de ezen felül számos érv szólt még mellette. A Java ökoszisztémája hatalmas, külső kiegészítés nélkül támogatja a grafikus kezelőfelületek megvalósítását, fájlkezelést, adatbáziskezelést, hálózatkezelést. Ezen felül rendelkezik számos más alapvető, fejlesztést megkönnyítő funkcióval, valamint rengeteg neves, és nagyon jól működő fejlesztői környezet (IDE) létezik hozzá. Ezen felül pedig egy széles körben ismert és tanított programozási nyelvről van szó.

4.2 Swing

A grafikus kezelőfelület, a kinézet, az elrendezés nagyban befolyásolja egy szoftver használhatóságát, tehát nagyon fontos egy alkalmazás szempontjából ennek megfelelő kiválasztása és kialakítása. Napjainkban Java környezetet használva a két fő opció a JavaFX, illetve a Swing.

A JavaFX egy viszonylag új, grafikus kezelőfelületek készítésére szánt technológia, amely a Swing-et hivatott leváltani. Igaz törekednünk kell a lehető legújabb technológiákat alkalmazni, azonban a grafikus kezelőfelület olyan mértékben határozza meg a végső felhasználói élményt, hogy ezen a területen nem engedhető meg, hogy a végeredmény ne legyen egységes, összeszedett, stabilan működő és gyors. Ezért a választás a valamivel régebbi, de éppen ezért kiforrottabb, stabilabb és nagyobb eszköztárral rendelkező Swing-re esett. Egyik ilyen meghatározó tényező, mely a Swing irányába vitte el a döntést a könnyen elérhető szintaxis színezett kódszerkesztő felület.

Emellett nem elhanyagolható az a szempont sem, hogy a Swing erőforrásigénye kevesebb, így a gyengébb gépeken sem okoz zavaró lassulást, ugyanolyan gyors és stabil futást tesz lehetővé. Ez a széles körben felhasználhatóság és a platformfüggetlenség tekintetében is kedvező tulajdonság.

4.3 Gradle

A program tervezése során nem csak az volt fontos szempont, hogy ugyanolyan jól fusson Linux és Windows operációs rendszereken is, hanem hogy a szoftver fejlesztése során is megmutatkozzon ez a rugalmasság. Ebben a szellemben a szoftver alapjaiban úgy lett megtervezve, hogy később könnyen továbbfejleszthető legyen, így annak elemei modulokra vannak bontva. Ezekről a modulokról és a moduláris működésről a későbbiekben lesz majd szó. Ez a fejezet a modularitásnak azt az egy előnyét taglalja, miszerint segítségével a projekt később könnyen bővíthető akár mások által is. A későbbi fejlesztők dolgának megkönnyítésére projektépítő eszköznek (szakszóval élve „build tool” -nak) a Gradle-re esett a választás [4]. A projektépítő eszköz használata lehetővé teszi, hogy a projekt később más fejlesztőkörnyezetben is módosítás nélkül folytatható legyen, így a jövőbeli fejlesztők ötleteinek és kreativitásának semmi sem szab határt, és jól megszokott fejlesztőkörnyezetükben tudnak dolgozni.

A **Gradle** egy projektépítő eszköz, amely az Apache Ant, és Apache Maven koncepcióján alapszik, viszont ezektől eltérő módon Groovy alapú nyelvet használ az XML alapú projektkonfiguráció helyett. Számos előnye van, amelyből egyik legfontosabb a függőségek kezelése (dependency management). Ha a projekthez egy újabb könyvtárat szeretnénk használni, akkor azt nem kell manuálisan letölteni, és azt az éppen adott fejlesztőkörnyezetben beállítani, hanem egyszerűen a deklarációs fájlba (build.gradle) be kell írni a függőség nevét és annak verzióját, és ezt projektépítés után a Gradle automatikusan letölti, valamint hozzácsatolja a projekthez. Példa egy ilyen kifejezésre a build.gradle fájlból:

```
compile group: 'com.fifesoft', name: 'rsyntaxtextarea', version: '2.6.1'
```

Ezzel az egy sorral az *rsyntaxtextarea* nevű könyvtár 2.6.1-es verzióját a Gradle automatikusan letölti, és az már használható is a fejlesztés során.

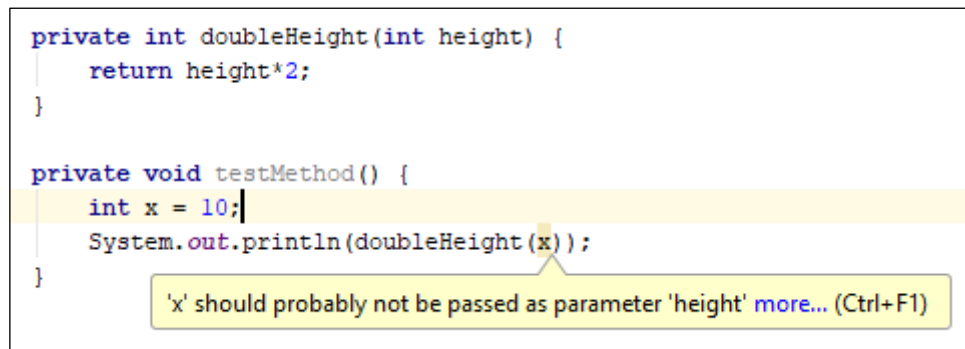
Ezen felül a Gradle könnyen használható eljárásokat biztosít, így például az adott projektből egy ilyen eljárás futtatásával létrehozható egy futtatható jar (**Java ARchive**) fájlt, amelyet már a JVM értelmezni tud.

4.4 IntelliJ IDEA

A Java ökoszisztémához számos fejlesztői környezet áll rendelkezésre, ezek közül a három főbb említésre méltó az **Eclipse**, **Netbeans**, és az **IntelliJ IDEA**. Mindhárom fejlesztői környezetnek létezik nyílt forráskódú verziója, valamint mindegyiküknek van számos előnye és hátránya is. Tanulmányaim során arra a következtetésre jutottam, hogy eme három fejlesztői környezet közül az **IntelliJ IDEA** az egyik legrugalmasabb és legkiforrottabb a választási lehetőségek közül.

A szoftver azért készült IntelliJ-ben, mert a már korábban említett **Gradle** támogatás plugin telepítése nélkül elérhető benne, (habár a másik két környezetnél is elérhető plugin formájában). A másik fő érv, amely inkább IntelliJ mellett szól, az a rugalmas debugolási lehetőségek, intelligens kód-kiegészítés, és hibák automatikus észlelése. IntelliJ-ben ezekre az eszközökre mesterséges intelligencia működik a háttérben, amely nagyon jó minőségben tudja végrehajtani mind a kód-kiegészítést, mind a kód-refaktorálást.

Egy ilyen érdekes példa: Ha egy függvénynek, amelynek egyik paraméterének neve *height*, egy *x* nevű változót adunk át, IntelliJ IDEA már ezt is képes észrevenni és szól, hogy *x* valószínűleg nem feleltethető meg magasságnak (erre inkább egy *y* paraméter lehet jó). Számos ilyen apró hibát képes kiszűrni még fejlesztés közben, melyet más fejlesztőkörnyezetekben nem tapasztaltam ilyen mértékben, ezáltal hihetetlen mértékben gyorsul a fejlesztés menete.



4.1 ábra: intelligens statikus kód elemzés IntelliJ IDEA-ban

4.5 Git

A projekt későbbi fejlesztése, illetve a jelenlegi fejlesztés megkönnyítése érdekében a projekt kezelése verziókövető rendszer segítségével történt a feladat megoldása során. A Git verziókövető rendszer nagy hangsúlyt fektet a sebességre. Verziókövető rendszer használatával nyomon követhető a projekt alakulása, valamint megkönnyíti a változások követését, és a hibajavítást.

A Git napjainkban az egyik legszéleskörűbben használt verziókövető rendszer, mely méltán örvend ilyen nagy népszerűségnek. Hatékonysága és egyszerűsége példanélküli, miközben funkciók széles körét teszi elérhetővé a fejlesztők számára, mint például verziók és verzió-elágazások (branch-ek) követése, összefésülése, nem lineáris fejlesztési történet ábrázolása, és ezáltal könnyű a kollaboratív csapatmunka.

Mivel nagy lélegzetvételű projektről van szó, és fontos volt, hogy későbbiekben a szoftver kiegészíthető legyen, ezért a build-tool (**Gradle**) mellett a verziókövető rendszer, vagyis **Git** használata is ezt a célt szolgálja.

5. Tervezés

Az előző fejezetekben szó esett a fontosabb követelményekről, a már létező megoldások feltárásáról, illetve a szoftver megvalósítása során alkalmazott technológiákról. A szoftver megvalósítása előtt fontos annak körütekintő megtervezése, így a szoftver könnyen bővíthető, érthető maradhat, valamint a fejlesztés közben előforduló meglepetések számát is csökkenti. A program struktúráját úgy kell kialakítani, hogy később ne igényeljen jelentős módosítást egy olyan funkció, amit egy kevésbé átgondolt struktúrába nem illik bele.

A következő alfejezetek kitér a szoftver fő felépítésére, a feladat tárolásához és megoldásához szükséges logikai struktúrákra, valamint a felhasználói felület tervére.

5.1 A szoftver főbb logikai struktúrái

A szoftver alapjaiban véve egy kurzusokra osztott feladat-megoldó és azt kezelő szoftver, amely több felhasználót támogat, így első lépés volt ezen alapfogalmak definiálása, a szoftverben beteljesített feladatuk, valamint a felelősségi köreik meghatározása. A fontosabb és összetettebb részek a mostani áttekintés után részletesen is megtalálhatóak.

5.1.1 Feladat

A szoftver fő építőeleme. A feladatok a legkisebb alkotóelemek, hiszen ezeket a feladatokat lehet megoldani a szoftver használata során. Egy feladathoz a következő adatok tartoznak:

- **Feladat leírása:** Leírja azt, hogy a feladatot milyen módon kell megoldani, mik a főbb követelmények, mire kell odafigyelni, valamint tartalmazhat segítő megjegyzéseket is.
- **Feladat metaadatai:** A feladat nehézsége, hossza, illetve a kulcsszavak (tagek) amelyek a feladatot jellemzik.
- **Forrásfájlok:** Kezdő forrásfájlok, amelyek a feladat megoldásának kezdetén rendelkezésre állnak. Például ilyen lehet egy kezdő forrásfájl egy *main* függvénnyel C programnyelv esetében. Egy-egy forrásfájl lehet írásvédett is.

- **Bemenet/Eredmény párok:** Az ellenőrzés során fontos, hogy annak megoldását a szoftver automatikusan ellenőrizni tudja. A Bemenet/Eredmény párok azt határozzák meg, hogy egy adott bemenetre milyen kimenetet kell adnia a megírt programnak. Egy feladathoz több ilyen pár tartozik. A megoldás helyességének ellenőrzése úgy történik, hogy a bemenetet a futó program bemenetére írja a rendszer, utána minden kimenetét soronként beolvassa, és végül összehasonlítja azt az eredménnyel, és az eltérő sorokat hibaként értelmezi.
- **Történet:** A feladat leíráson túl a feladathoz tartozik egy történet is. Ez általában nem tartalmaz olyan információt, amely a feladat megoldásához szükséges, de lehetővé teszi érdekesebb és változatosabb tananyag létrehozását.

5.1.2 Feladatgyűjtemény

A feladatgyűjtemény egy olyan adatstruktúra, amely feladatok egy halmazából egy irányított gráfot ír le.

Egy feladatgyűjteményhez a következő adatok tartoznak:

- **Feladatok halmaza, és azon értelmezett irányított gráf:** Az irányított gráfban egy feladat egy csúcsnak felel meg, és a feladatok egymásra épüléseit írják le a gráf élei.
- **Feladatgyűjtemény küszöbértéke:** Egy 0 és 1 közötti érték, amely azt írja le, hogy hány feladatot kell egy feladatgyűjteményből megoldani, hogy az teljesítettnek számítsen. Például, ha 10 feladat található benne, és ez az érték 0.5, akkor 5 feladatot kell sikeresen teljesíteni, hogy a feladatgyűjtemény teljesített legyen, és az erre a feladatgyűjteményre épülő feladatgyűjtemények elérhetővé váljanak.

5.1.3 Mód

A mód egy adatstruktúra, amely feladatgyűjtemények egy halmazából egy irányított gráfot ír le. Egy módhoz a következő adatok tartoznak:

- **Feladatgyűjtemények halmaza, és azon értelmezett irányított gráf:** Az irányított gráfban egy feladatgyűjtemény egy csúcsnak felel meg, és a feladatgyűjtemények egymásra épüléseit írják le a gráf élei. Fontos, hogy egy feladatgyűjtemény akkor számít teljesítettnek, ha abból egy bizonyos mennyiségű feladatot megoldottunk már. Ezt az értéket maga a feladatgyűjtemény tárolja, ahogy azt az 5.1.2 alfejezet leírja.
- **Flagek,** amelyek a mód tulajdonságait adják meg. Két ilyen flag a ráépülések és a történetek figyelmen kívül hagyása. A ráépülések figyelmen kívül hagyása azt jelenti, hogy a tartalmazott feladatgyűjteményekben a feladatok közötti éleket figyelmen kívül hagyja. Ez hasznos például egy szabadon játszható mód létrehozására, amely már korábban definiált feladatgyűjteményeket használ fel újra, amiben voltak ráépülések a feladatok között. A történet figyelmen kívül hagyása a megoldás során a mód feladatgyűjteményeiben található feladatokban lévő történetek megjelenítését ignorálja.

5.1.4 Kurzus

A szoftver fő építőelemei a kurzus nevet kapták. Egy-egy kurzus leginkább egy féléves egyetemi programozási kurzusnak feleltethető meg. A kurzusok lehetnek akár programnyelvenként, például C kurzus, C++ kurzus, Java kurzus, de lehetőség van akár programnyelvenként több kategóriát kialakítani, több kurzust létrehozni, így igazodva az oktatási intézmény tantervéhez. Erre példa, hogy a Programozás II és a Korszerű programozási technikák tárgyak mindketten a C++ programozási nyelv oktatásáról szólnak, de a tananyag jelentősen eltér. Tehát egy kurzus mindig egy meghatározott programnyelv, technológia, és tananyag köré épül. Egy kurzuson belül több mód érhető el.

Ezen a ponton fontos kitérni arra, hogy az is lényeges szempont volt, hogy egy kurzuson belül egy-egy feladat, és feladatgyűjtemény újra felhasználható

legyen. Az adatbázis struktúra fájl alapú megvalósítása később így ennek szellemében történt.

5.1.5 Felhasználó modul

A felhasználó modul a szoftver fő használóit írja le. A felhasználó modul fő feladata az, hogy tárolja, hogy mely feladatokat, illetve feladatgyűjteményeket oldotta már meg a felhasználó, valamint a feladatok megoldásait, illetve a folyamatban lévő megoldásokhoz tartozó, szerkesztés alatt lévő forrásfájlokat is. Minden felhasználóhoz külön tárolja ezeket az adatokat, így egy szoftver példányt több felhasználó is használhat egy időben. Egyetemi környezetben a felhasználó lényegében egy hallgatónak feleltethető meg.

5.2 Felhasználói felület terve

A főbb fogalmak definiálása után következett a felhasználói felület megtervezése, a felhasználó által tapasztalt felületek fő felületek definiálása, azok jogköreinek meghatározása. A szoftver alapvetően kettő különálló felhasználói felülettel kell rendelkezzen, mivel a szoftvernek két fő funkciója van. Egyik ilyen funkció a feladat böngésző, és megoldó rész, amelyet a hallgatók használnak. A másik funkció a feladatbank szerkesztése, amely az oktatók számára biztosít felületet arra, hogy a szoftver által tárolt feladatokat, illetve az azokhoz tartozó adatokat szerkeszteni tudják.

A feladat megoldó felület a következő főbb képernyőkből áll: a bejelentkező képernyő, a kurzus böngésző képernyő, és a feladatmegoldó képernyő. A bejelentkező képernyőn léphet be a hallgató a szoftverbe. A kurzus böngésző képernyőn választhatja ki a kurzust, azon belül pedig azt, hogy melyik feladatot szeretné megoldani. Itt a feladatok rövid leírásai, és egymásra épülései is megjelennek. A feladat megoldó felületen jelenik meg a feladat fő szövege és a szerkeszthető forrásfájlok, illetve egy terminál felület, ahol a fordító üzenetei, a futtatott program kimenete, és az automatikus ellenőrzés történik majd.

A feladat szerkesztő felület szintén több képernyőből áll: kurzus választó képernyő, illetve kurzus szerkesztő képernyő, amin belül három főbb felület érhető el: mód szerkesztés, feladatgyűjtemény szerkesztés és feladat szerkesztés.

A kurzus választó felületen választható ki a szerkeszteni kívánt kurzus, a három kurzuson belüli felületen pedig a kurzusban található adatok szerkeszthetők.

A két felület elérhető a fő programból, de alapértelmezetten a feladat megoldó felület kezdőképernyője jelenik majd meg. A feladat szerkesztő felület kezdőképernyője a „-wizard” parancssori argumentum megadásával jelenik meg. Tehát a futtatható JAR fájlt a „-wizard” opcióval kell elindítani.

6. Megvalósítás

A követelmények lefektetése, hasonló szoftverek megismerése, technológiák kiválasztása és kezdeti tervek megalkotása után a fájl alapú adatbázis létrehozása, valamint a felhasználói felületek megvalósítása és az összes funkció implementálása következett. A felhasználói felületek funkcionalitásának implementálása előtt még a szoftver későbbi könnyű bővíthetőségét lehetővé tévő interfészek létrehozása is fontos szerepet töltött be a megvalósítás során.

6.1 Fájl alapú adatbázis struktúra

Az első lépés a fájl alapú adatbázis struktúra megvalósítása volt a feladatok tárolására, amelyeknek létrehozására, megoldására, és szerkesztésére a szoftver lehetőséget nyújt. A fájl struktúrának könnyen lehetővé kell tennie az **5.1** fejezetben definiált logikai struktúra megvalósítását, a szoftverben annak megjelenítését, valamint a könnyű bővíthetőséget és szerkeszthetőséget. További szempont volt az, hogy a kurzuson belül található feladatok és feladatgyűjtemények újra felhasználhatóak legyenek a kurzuson belül.

A szoftverben az adatok tárolása fájl alapon történik. Ez azt jelenti, hogy nem egy távoli elérésű adatbázisban, hanem lokálisan a szoftvert futtató számítógépen, esetleg hálózati meghajtóról érthetőek el a feladatok.

Így a szoftver nem függ folyamatosan szervertől, és rugalmasabban felhasználható, hiszen offline is könnyen használható otthon, vagy akár út közben is. Ez természetesen felveti azt a kérdést, hogy mi van akkor, ha a feladatok adatbázisát később egységesen szeretnénk elérhetővé tenni, és frissíteni a felhasználók számára. Erre jó megoldás lehet, ha a szoftver távoli szerverre csatlakozik, és azt követően a fájlstruktúrába tölti le az adatbázist. Ennek megvalósítása a szakdolgozat témáján már kívül esik, viszont a fájl alapú adatbázis ennek későbbi megvalósítását nem zárja ki.

6.1.1 Fájl struktúrák és kiterjesztések

A fájl alapú adatbázis létrehozásához elengedhetetlen volt a fájlok struktúrájának megfelelő létrehozása, amelyből ez az adatbázis felépül majd. Mivel nem tábla alapú adatbázisról van szó, ezért először meg kell határozni a tárolt fájlok

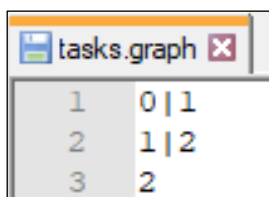
kiterjesztését és felépítését, valamint implementálni az ezekhez tartozó osztályokat.

Az adatbázisban a hagyományos **XML** (Extensible Markup Language) fájlokon kívül több saját formátumú és kiterjesztésű fájl is található.

Első ilyen főbb fájl a *dat* kiterjesztésű fájl. Alapértelmezetten ez egy gyengén strukturált tábla szerkezetet ír le. Az adatok soronként vannak benne tárolva, és egy-egy sorban az adatok a | (pipe) jellel vannak egymástól elválasztva. A *dat* kiterjesztésű fájl bármennyi sort tartalmazhat, és minden egyes sorban bármennyi, eltérő mennyiségű adat lehet. Egy sor akár egyetlen adatot is tartalmazhat.

A második főbb fájl struktúra a *conf* kiterjesztésű fájl. A konfigurációs fájlok egyszerűbb változata, amely a soraiban kulcs-érték párokat tárol = jellel elválasztott formában.

A harmadik főbb fájl struktúra a *graph*, és *graph.view* kiterjesztésű fájl. A *dat* fájlok specifikusabb változatai. Egy gráfot és annak megjelenítését írják le. A *graph* kiterjesztésű fájl önmagában is elég egy gráf értelmezéséhez, a hozzá tartozó *graph.view* kiterjesztésű nézet fájl csak a könnyebb szerkeszthetőséghez biztosítja a szükséges nézet adatokat.



1	0 1
2	1 2
3	2

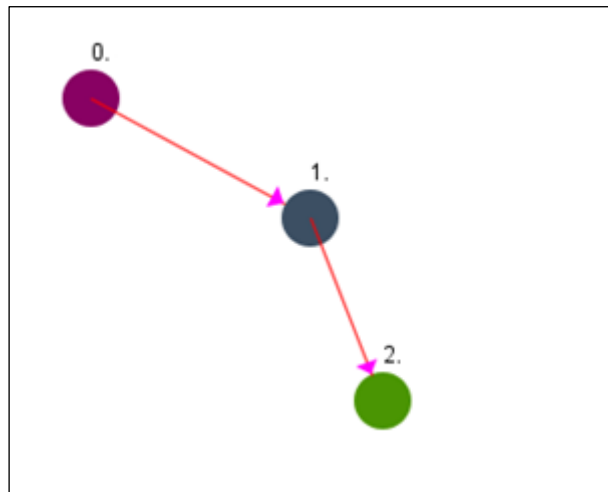
6.1 ábra: Egy *graph* típusú fájl tartalma

A *graph* fájl a | (pipe) szimbólumot használja az adatok elválasztására (mivel ez egy *dat* típusú fájl struktúra). A legelső érték egy sorban mindig a kiindulási csúcs, a rákövetkező adatok pedig azok a csúcsok, amely csúcsoknak ő lesz a szülője. A 6.1 ábrán látható fájlt úgy kell értelmezni, hogy itt a 0. csúcsra épül az 1. csúcs, az 1. csúcsra épül a 2. csúcs. A 2. csúcsra nem épül már semmi. A *graph* fájl kiegészítését a *graph.view* kiterjesztésű fájl tartalmazza. Ennek a fájlnak a struktúrája a következő:

tasks.graph.view					
1	0		136		1 99 80 120
2	1		60		79 99 195 183
3	2		74		149 3 233 279

6.2 ábra: Egy graph.view típusú fájl tartalma

Az első érték a gráf csúcsának indexét jelenti. Az azt követő három adat RGB formátumban értelmezett színekódot ír le, és az utolsó két adat egy x és y koordinátát jelent pixelekből megadva. A 6.4. fejezetben később részletesebben kifejtett feladat adatbázis szerkesztő felületen ez az adat a 6.3. ábrán látható módon jeleníthető meg.



6.3 ábra: graph és graph.view fájl megjelenítve

Egyéb egyszerűbb kiterjesztések közé tartozik az *input*, és *result* kiterjesztés is, amelyek semmilyen extra struktúrát nem írnak le, azok tartalma megegyezik egy szöveges fájlal, csak a megkülönböztetés miatt kaptak egyedi kiterjesztést.

A bemutatott fájlok beolvasásához és értelmezéséhez szükség van a megfelelő osztályokra. A *DataFile* osztály tudja beolvasni a `|` (pipe) jellel elválasztott adatokat. Értelemszerűen ez *dat*, illetve a *graph*, és *graph.view* fájlok beolvasására alkalmas. A *SingleDataFile* osztály olyan fájlokat tud beolvasni amelyben egyetlen adat található. A *DataFile* osztályból származtatott *ConfigFile* osztály az `=` jelet használja az adatok szétválasztására soronként. A feladat adatainak tárolása egy XML fájlban történik, így szükségszerű volt annak értelmezése, amelyet a *DescriptionFile* osztály valósít meg. Ez csak egy bizonyos

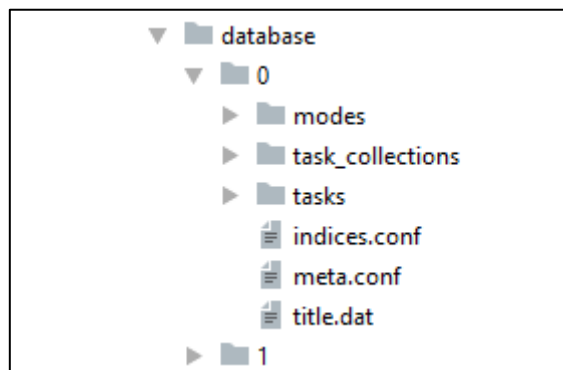
felépítésű XML fájlt képes beolvasni, amelynek struktúráját a **6.1.3** alfejezet ismerteti majd részletesen. A gráf adatok értelmezésére egy *Graph* interfész szolgál, amelyet *AdjacencyList* osztály implementál egy adjacencia lista formájában.

A fájl kiterjesztések, és struktúrák definiálása, illetve azokhoz értelmező osztályok implementálása után az adatbázis struktúra létrehozása következett.

6.1.2 Kurzusok fájl alapú tárolása

Mivel az adatbázis tárolása fájl alapon történik, így az adatbázis kiindulási pontja egy mappa. Ez a mappa alapértelmezetten a *database* nevű mappa a szoftver mellett.

A *database* mappa tartalmazza a kurzusokat. Egy **gyakorlati példán** nézve az alábbi **6.4.** ábrán látható egy olyan adatbázis struktúra, amelyben kettő darab kurzus található.



6.4 ábra: kurzusokat tartalmazó adatbázis mappa

A **6.4.** ábrán látható, hogy a *database* nevű mappában kettő darab mappa található 0 és 1 néven. Ezek a 0. és 1. indexű kurzusnak felelnek meg. Egy-egy kurzus a következő adatokat tárolja a fájlrendszerben:

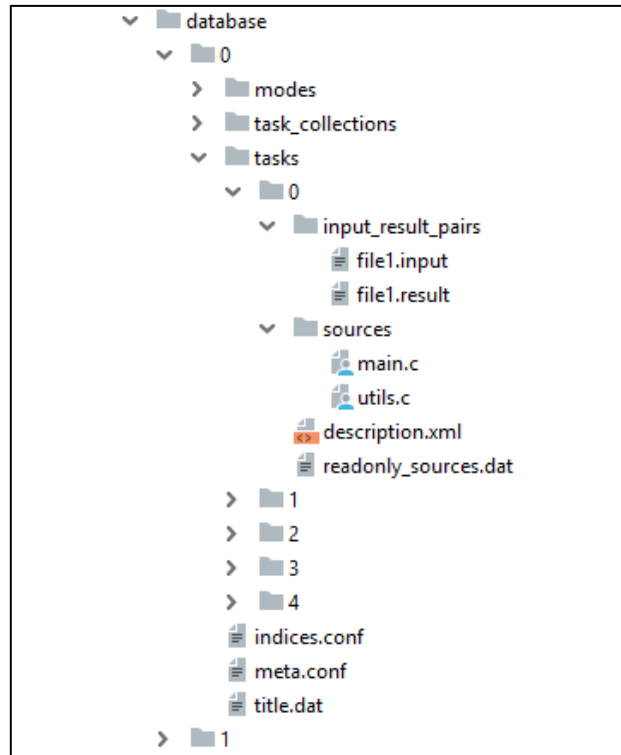
- **modes** mappa: módok mappáit tartalmazza. Ennek a mappának a tartalma a **6.1.5** alfejezetben kerül részletezésre.
- **task_collections** mappa: feladatgyűjtemények mappáit tartalmazza. Ennek a mappának a tartalma a **6.1.4** alfejezetben kerül részletezésre.
- **tasks** mappa: feladatok mappáit tartalmazza. Ennek a mappának a tartalma a **6.1.3** alfejezetben kerül részletezésre.

- **indices.conf:** a módok, feladatgyűjtemények és feladatok indexeléséhez tárol adatokat. Lehetővé teszi, hogy amikor új módot, feladatgyűjteményt, vagy feladatot adunk a kurzushoz, akkor az minden esetben egyedi azonosítóval rendelkezzen. Ez az azonosító megegyezik a mappa nevével, amiben az új feladat, feladatgyűjtemény vagy mód található. Létrehozás után az index értéket inkrementálja 1-gyel, így biztosítva egyediségét az újonnan hozzáadott elemeknek.
- **meta.conf:** a programnyelv megnevezését tartalmazza, amely a kurzushoz tartozik. Minden feladat ezen a programnyelven oldható meg.
- **title.dat:** a kurzus nevét tartalmazza, amit a szoftver megjelenít. Értelmszerűen ez az indextől (a mappa neve) különálló adat.

A feladatok, és feladatgyűjtemények azért vannak a kurzus mappájában tárolva, mert így lehetőség van arra, hogy minden feladat és feladatgyűjtemény újra felhasználható legyen.

6.1.3 Feladatok fájl alapú tárolása

Egy-egy kurzuson belül található egy adott mennyiségű feladat. Ezek a feladatok a kurzuson belül a *tasks* nevű mappában vannak tárolva. A 6.1.2 alfejezetben ismertetett gyakorlati példán keresztül bemutatva a 6.5. ábrán láthatóak szerint alakul a kurzus, és azon belül a feladatokat tartalmazó mappaszerkezet.



6.5 ábra: Egy feladat fájl struktúrája

A 6.5. ábrán látható, hogy a 0. kurzus mappájában található *tasks* mappában öt darab feladat található. Ezeknek az indexei 0 és 4 között mozognak. Egy-egy feladat a következő adatokat tárolja a fájlrendszerben:

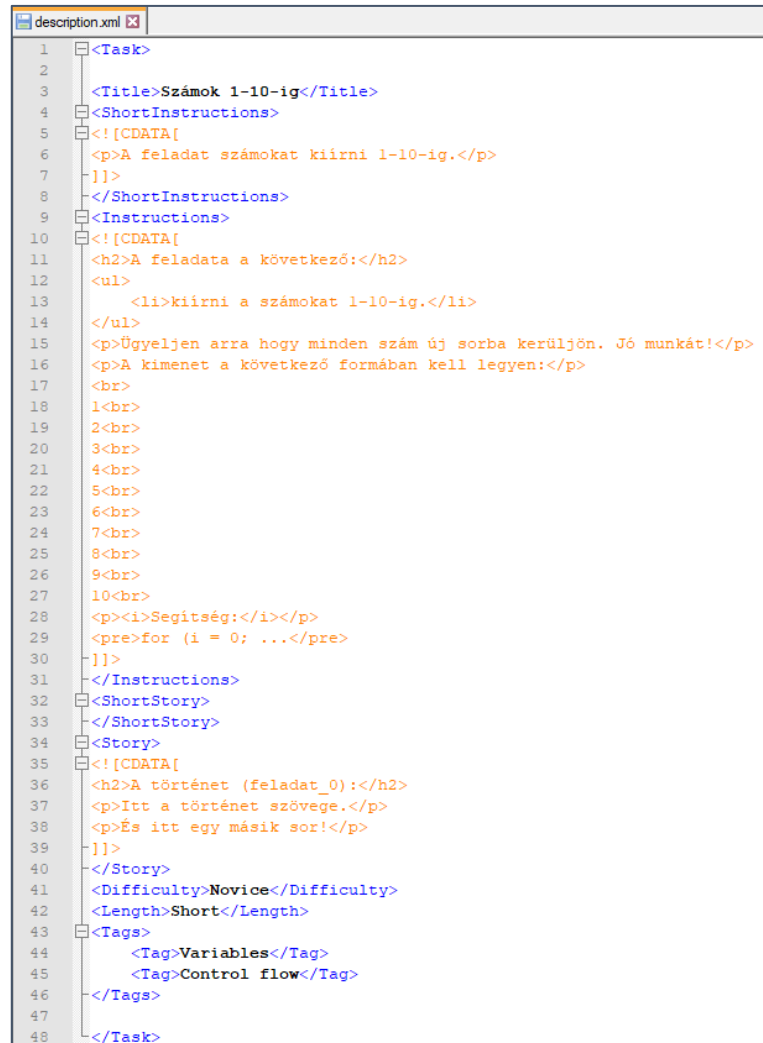
- **input_result_pairs** mappa: ebben a mappában találhatóak azok a fájlok, amelyeket majd a megoldott feladat futtatásához használ a szoftver a megoldott feladat helyességének ellenőrzése céljából. A bemeneti fájlok *input*, az eredmény fájlok *result* kiterjesztéssel vannak ellátva. Minden *input* és *result* fájl párban tartozik egymáshoz, tehát minden *input* kiterjesztésű fájlhoz tartozik egy *result* kiterjesztésű fájl úgy, hogy a fájl neve megegyezik. A fenti példában ez a *file1* nevű fájl. A megoldott feladat futtatása során a szoftver az *input* kiterjesztésű fájl tartalmát a futó program input streamjére írja, utána soronként beolvassa az output streamjét, és összeveti azt a *result* fájl tartalmával soronként. Az eltérő sorokat hibásként értelmezi a program.
- **sources** mappa: a kezdeti forrásfájlok tárolására szolgáló mappa. Ezek a forrásfájlok a feladat megoldásának megkezdésekor rendelkezésre állnak a felhasználó számára. A példában látható egy *main.c* nevű fájl, és egy

utils.c nevű fájl. Ez a két fájl elérhető és szerkeszthető lesz a feladat megoldásakor.

- **description.xml**: egy XML kiterjesztésű és struktúrájú fájl, amely a feladat megjelenített nevét, a feladathoz tartozó feladateleírást, történetet és egyéb jelzés értékű információkat (tagek, nehézségi szint, hossz) tárolja.
- **readonly_sources.dat**: ebben a fájlban a kezdő forrásfájlok nevei vannak bejegyezve, amelyek írásvédettek, így nem szerkeszthetők.

Az XML egy általános leíró nyelv, viszont a *description.xml* fájl struktúrája meg van kötve, az az alábbi nyolc elemet kell tartalmazza:

- **Title**: a feladat megjelenített nevét tartalmazza.
- **ShortInstructions**: a feladathoz tartozó feladateleírás rövidített verzióját tartalmazza HTML formátumban.
- **Instructions**: a feladathoz tartozó feladateleírást tartalmazza HTML formátumban.
- **ShortStory**: a feladathoz tartozó történet rövidített verzióját tartalmazza HTML formátumban.
- **Story**: a feladathoz tartozó történetet tartalmazza HTML formátumban.
- **Difficulty**: a feladat nehézségét tartalmazza.
- **Length**: a feladat hosszát tartalmazza.
- **Tags**: Tag elemeket tartalmaz, amelyek a feladathoz tartozó kulcsszavak. Több Tag is tartozhat egy feladathoz.

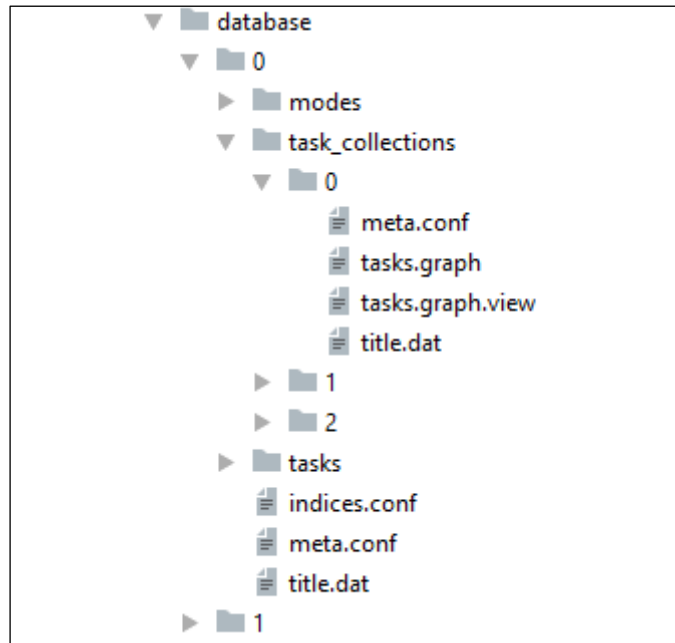


6.6 ábra: A description.xml fájl struktúrája

6.1.4 Feladatgyűjtemények fájl alapú tárolása

Egy-egy kurzuson belül találhatóak a feladatgyűjtemények is, amelyek a kurzuson belül található feladatok közül hivatkoznak egy részhalmaznyi feladatra, illetve azon definiálnak egy irányított gráfot, amely a feladatok egymásra épülését jelzi a feladatgyűjteményen belül. A feladatgyűjtemények a *task_collections* mappában vannak tárolva.

Egy feladatgyűjtemény mappaszerkezete a **6.7.** ábrán látható, az előző alfejezetekben használt gyakorlati példán keresztül bemutatva.



6.7 ábra: Egy Feladatgyűjtemény fájl struktúrája

A 6.7 ábrán látható példában a *task_collections* mappa tartalma három feladatgyűjtemény, a 0, 1 és 2 indexű feladatgyűjtemények. Egy-egy feladatgyűjtemény mappájában az alábbi adatok vannak tárolva:

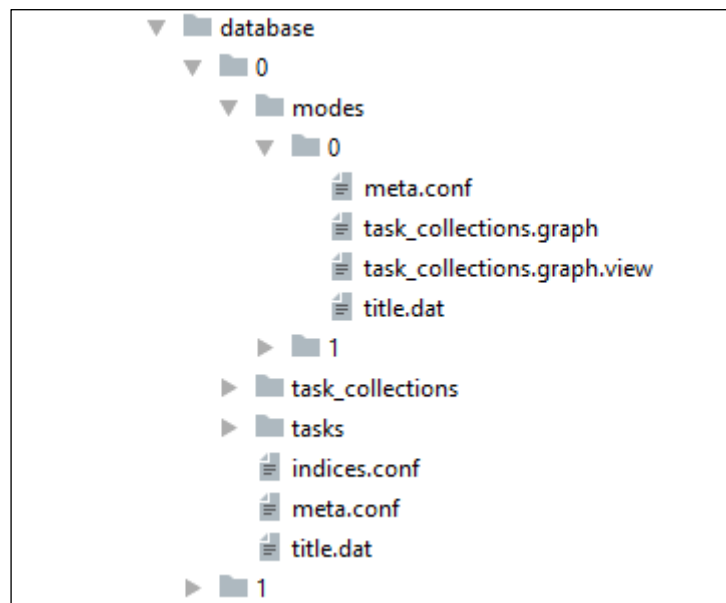
- **meta.conf:** a feladatgyűjtemény meta adatait tárolja. Ez a meta adat jelenleg egy értékből áll, ami azt mutatja, hogy a feladatgyűjtemény összes feladata közül mennyit kell megoldani, hogy a feladatgyűjtemény teljesítve legyen. Például ha 30 feladat van a feladatgyűjteményben és ez az érték 0.66, akkor 20 feladatot kell megoldani a feladatgyűjteményből, hogy az teljesítve legyen. A fájl megléte lehetőséget biztosít hogy a program későbbi kiegészítése során egyéb adatokat is tudjon tárolni.
- **tasks.graph:** egy éllista, amely leírja, hogy ehhez a feladatgyűjteményhez mely feladatok tartoznak, és azok hogyan épülnek egymásra.
- **tasks.graph.view:** ez a fájl extra adatokat tartalmaz a *tasks.graph* fájlhoz. Fontosnak tartottam, hogy a ráépülések szerkesztése a lehető legkönnyebb módon történjen, így a grafikus gráfszerkesztő felületnek biztosítja a szükséges adatokat. Ezt a fájlt csak a feladat adatbázist szerkesztő felület értelmezi, és ezek alapján jeleníti majd meg a gráfot vizuálisan.

- **title.dat:** ebben a fájlban a feladatgyűjtemény megjelenített neve van tárolva. Értelmszerűen ez az indextől (a mappa neve) különálló adat.

6.1.5 Módok fájl alapú tárolása

Egy-egy kurzuson belül találhatóak a módok is, amelyek a kurzuson belül található feladatgyűjtemények közül hivatkoznak a mód számára elérhetőekre, illetve definiálnak egy irányított gráfot, amelyek a feladatgyűjtemények egymásra épülését jelzi a módon belül. Az egymásra épülés így a módok között akár eltérő is lehet. A módok a *modes* mappában vannak tárolva.

Egy mód mappaszerkezete a **6.8** ábrán látható módon néz ki, az előző alfejezetekben használt gyakorlati példán keresztül bemutatva.



6.8 ábra: Egy Mód fájl struktúrája

A **6.8.** ábrán látható példában a *modes* mappában kettő darab mód található 0 és 1 indexszel. Egy mód mappája a következő adatokat tartalmazza:

- **meta.conf:** a mód meta adatait tartalmazza. Ez a meta adat két flagből áll, *ignore_dependency*, és *ignore_story* nevű flagek. Az *ignore_dependency* flag egy igaz/hamis érték, amely azt határozza meg, hogy ebben a módban hivatkozott feladatgyűjtemények közötti ráépüléseket figyelmen kívül hagyja-e a szoftver. Az *ignore_story* flag is igaz/hamis érték, ami azt

határozza meg, hogy a módban tartalmazott feladatgyűjteményekben található feladatokban a történet jelenjen-e meg a megoldás során.

- **task_collections.graph**: egy éllista, amely leírja, hogy ehhez a módhoz mely feladatgyűjtemények tartoznak, és azok hogyan épülnek egymásra.
- **task_collections.graph.view**: ez a fájl extra adatokat tartalmaz a *task_collections.graph* fájlhoz, hogy azt a szoftver meg tudja jeleníteni szerkesztés közben.
- **title.dat**: ebben a fájlban a mód megjelenített neve van tárolva. Értelmszerűen ez az indextől (a mappa neve) különálló adat.

6.1.6 Felhasználók fájl alapú tárolása

A fájl alapú adatbázis struktúrához tartozik egy munkaterület. Ez egy olyan mappa struktúra, amelyben felhasználónkként tárolva vannak a megoldás alatt lévő feladatok, valamint azon feladatok és feladatgyűjtemények azonosítójának listáját is tárolja, amelyeket a felhasználó már megoldott. Ezen adatok tárolásával követhetővé válik a felhasználó előrehaladása, valamint ezen információk segítségével a szoftver eldöntheti, hogy mely feladatokat tudja a felhasználó elkezdni, és melyek azok, amiknek az előfeltételei még nem teljesültek.

A felhasználók előrehaladása egy külön gyökérkönyvtárban van tárolva az adatbázis mappájától függetlenül. Nem szerencsés megoldás ha a felhasználó magát az adatbázist módosítja feladatok megoldása közben. Ezen indokból a munkaterület egy *working_directory* nevű mappában érhető el a szoftver mellett.

Ebben a mappában találhatóak a már elkezdett, illetve megoldott feladatok, az azokhoz tartozó forrásfájlok, valamint egy-egy lista az elkezdett és a teljesített feladatokról, valamint a teljesített feladatgyűjteményekről is. Ezen követelmények alapján a *working_directory* nevű mappában felhasználónként létezik egy-egy almappa, és egy-egy felhasználó mappájában egy *.meta* nevű almappában az előbb említett három lista található:

- **completed_task_collections.dat**: tárolja azon feladatgyűjtemények listáját, amelyeket a felhasználó teljesített, vagyis a megoldott feladatok száma nagyobb vagy egyenlő, mint az összes feladat száma a feladatgyűjteményben, szorozva a teljesítéshez szükséges

küszöbértékkel, amely 0 és 1 között mozog. Egy feladatgyűjteményt a következő bejegyzés segítségével tárol a fájl a listában: *course_id/mode_id/task_collection_id*. Vagyis például, ha a 3. azonosítójú kurzus 0. azonosítójú módjából a 2. azonosítójú feladatgyűjteményét oldotta meg, akkor a következő sor lesz bejegyezve ebbe a fájlba: *3/0/2*.

- **completed_tasks.dat**: tárolja azon feladatok listáját, amelyeket a felhasználó teljesített, vagyis legalább egyszer írt egy olyan kódot, ami az összes bemeneti fájlra pont ugyan azt az eredményt hozta, ami az ahhoz tartozó kimeneti fájlban van. Egy feladatot a következő módon tárol a fájl a listában: *course_id/mode_id/task_collection_id/task_id*. Vagyis, ha például a 5. azonosítójú kurzus 2. azonosítójú módjában a 6. azonosítójú feladatgyűjteményéből a 19. azonosítójú feladatot oldotta meg, akkor a következő sor lesz bejegyezve ebbe a fájlba: *5/2/6/19*.
- **started_tasks.dat**: azokat a feladatokat tárolja, amelyet a felhasználó már elkezdett. Ugyanolyan formátumban tároljuk az azonosítókat, mint a *completed_tasks.dat* fájlban.

A listákon felül szükséges még tárolni a feladatokhoz tartozó folyamatban lévő, illetve megoldott feladatokhoz tartozó forrásfájlokat. A felhasználó fő mappájában az azonosítókkal megegyező mappaszerkezet tárolja ezeket az információkat.

Például ha a USER nevű felhasználó elkezdte a 0. azonosítójú kurzusban a 2. azonosítójú módban a 7. azonosítójú feladatgyűjteményből a 11. azonosítójú feladatot, akkor a következő mappaszerkezetben lesznek annak a forrásfájljai megtalálhatóak: *working_directory/USER/0/2/7/11*. Ha a 11. azonosítójú feladat egy másik módban is elérhető, például a 3. azonosítójú módban, akkor annak megoldása egy másik mappában lesz tárolva.

6.2 Programok fordítása és futtatása

A szoftver alapjainak megtervezése, logikai struktúrájának definiálása, és a fájl alapú adatbázis megvalósítása után a következő lépés a különböző programozási nyelvekkel való együttműködés. Ahhoz, hogy a szoftver több programozási

nyelvet támogatni tudjon, és később ez könnyen kiegészíthető legyen, egy absztrakciós szintre van szükség, amely az összes programozási nyelv esetében hasonlóan működik. Ezt az absztrakciót interfészek segítségével, illetve dinamikus osztály betöltéssel valósítottam meg. Fontos lépés volt még az, hogy ne csak a programok futtatása legyen lehetséges, hanem azok fordítása is, illetve a standard streamek írása és olvasása is lehetséges legyen a szoftverből, hogy a tesztesetek segítségével a megoldást a szoftver automatikusan ellenőrizni tudja.

6.2.1 Osztályok és interfészek

A főbb funkciók implementálásához az alábbi osztályokra és interfészekre van szükség:

- **Program**, amely egy futtatható állományt reprezentál
- **ProgramCompiler**, amely egy külső fordítót reprezentál
- **TaskFactory**, amely a feladat modelljét és a fordítót készíti el

Az interfészek mellett több osztályra is szükség volt, ezek a következők:

- **StreamUtilities** a folyamatok streamjeinek olvasására
- **Result**, amely egy program futása során keletkezett kimenet sorait tárolja
- **User**, amely a jelenleg belépett felhasználót reprezentálja
- **Task** a megoldandó feladat modellje, amely az adatbázisból tölti be az adatait.
- **TaskFactoryDecider**, amely egy *TaskFactory* interfész alapján megírt osztály egy példányát készíti el. Például C esetén ez a *TaskCFactory* nevű osztály egy példánya lesz.
- **CompilerOutput**, amely egy fordító kimenetét tárolja, annak hibaüzeneteit, visszatérési értékét illetve a *Program* interfészt implementáló osztály egy példányát.

A főbb osztályok mellett szükséges a három interfész implementációja is, amelyek a C programnyelv támogatását valósítják meg:

- **ProgramC**, amely egy C nyelvben írt programot reprezentál
- **ProgramCompilerC**, amely a gcc külső fordítót reprezentálja

- **TaskCFactory**, amely képes létrehozni egy C típusú fordítót, és egy *Task* objektumot, amely a megoldandó feladat modellje.

A felsorolt osztályok és interfészek implementálásáról és működéséről a 6.2.2, 6.2.3 és 6.2.4 fejezetek adnak bővebb képet.

6.2.2 Streamek olvasása

Az első lépés néhány segéd metódus implementálása volt. Egy feladat megoldása során két fontos funkcionalitásnak kell működnie. Ez a program fordítása és a program futtatása. Ehhez Javában a *ProcessBuilder* osztály nyújt segítséget, mivel mind a fordító, mind a lefordított program egy külső folyamatként fog futni.

A *ProcessBuilder* osztály képes elindítani folyamatokat. A *ProcessBuilder* konstruktora *String* típusú adatot fogad, és az egy parancssorba beírt parancsnak felel meg. A *ProcessBuilder* objektum *start* metódusának visszatérési értékeként kapható meg az a *Process* példány, amely már a futó folyamatot jelképezi. A *Process* objektum lehetővé teszi az általa kezelt folyamat standard output és error streamjének olvasását, illetve az input streamjére az adatok írását. A segédmetódusok ezt az írást és olvasást hivatottak leegyszerűsíteni.

A fordítóprogram esetében viszonylag egyszerű ilyen stream olvasó metódust írni, mivel a fordítóprogram megbízható kimenetet ad. A segédmetódusok a *StreamUtilities* osztályba vannak implementálva statikus elérési metódusként. A fordítóprogram output streamjének olvasására létrehozott *getStream* metódus úgy működik, hogy azt soronként olvassa be, és egy *ArrayList<String>* listát ad vissza, amiben soronként a fordítóprogram kimenete található. Ez az adat könnyen megjeleníthető.

A lefordított programok output streamjének olvasása már nem ilyen egyszerű művelet. A feladat megoldása során számolni kell azzal, hogy a felhasználó súlyos hibákat követhet el, és a megoldás nem lesz tökéletes. A szoftver az ellenőrzést oly módon oldja meg, hogy a futó folyamatnak az *input_result_pairs* mappából az *input* kiterjesztésű fájlok tartalmát írja a folyamat standard inputjára, utána olvassa annak standard outputját, és összehasonlítja a tartalmát soronként a *result* kiterjesztésű fájl tartalmával, amelynek neve megegyezik az *input* kiterjesztésű fájl nevével.

Probléma akkor merülhet fel, ha például a feladat megoldása során a felhasználó úgy íratja ki a megoldásokat, hogy elfelejt sortörést tenni azok végére. Ekkor nem lehet soronként olvasni a folyamat outputját, mivel egyetlen sornak sincs vége a kiíratás során, így az olvasás sikertelen lesz. Még nagyobb probléma az, ha a megoldás során végtelen ciklus keletkezik, és az olvasás is a végtelenségig fog tartani. Egyszerű soronkénti olvasás helyett az adatot byteonként kell beolvasni, és a buffer méretét limitálni, ami után már az ellenőrző nem olvassa tovább a kimenetet. Ennek segítségével a végtelen olvasási ciklust meg lehet szakítani.

A *getStream* metódus kiterjesztett verziója byteonként olvassa az output streamet. A *getStream* metódustól eltérően ez a verzió egy maximum buffer méretet is kér. Alapértelmezetten 1800 milliszekundumot vár a futó folyamatra. Ha az ez idő alatt semmilyen kimenetet nem ad, akkor megszakad az olvasás. Ha az olvasás sikeres, akkor egy ciklusba lép, amely byteonként olvassa be az adatot, míg nem olvasott be annyi byteot amennyi a megadott maximum (alapértelmezetten 10 kbyte), vagy el nem fogy az olvasni való adat. Olvasás közben az adatokat egy *ArrayList<String>* típusú listában tárolja el minden alkalommal amikor sortörést olvas. Ekkor egy új *String*-et tesz ebbe a listába, így soronkénti listát kapunk, mint az egyszerűbb *getStream* esetében is. Ha az utolsó sor végén nincs lezáró sortörés, akkor is eltárolja azt az utolsó sort, így elkerülve az adatvesztést.

Ezzel az egyszerűbb, soronként olvasó *getStream* metódussal és az összetettebb byteonként olvasó verziójával már az összes nem helyes programot is tudja olvasni a szoftver.

6.2.3 Főbb osztályok, és Interfészek létrehozása

A standard input olvasására létrehozott metódusok implementálása után a következő lépés a főbb interfészek létrehozása volt, annak érdekében, hogy a program fordítására és futtatására egy absztrakciós szintet biztosítson.

Az interfészek fő feladata jelen esetben az, hogy a szoftver később is könnyen bővíthető legyen újabb programnyelvekkel/fordítókkal. Ehhez a szoftverben semmilyen meglévő osztályon nem kell módosítani, csak az

interfészeket implementálni. A feladat megoldás során értelemszerűen két fő funkcionalitás szükséges. Az egyik a megoldandó feladat forráskódjának lefordítása külső fordító meghívásával, a második lépés pedig a lefordított program futtatása, illetve annak standard outputjának olvasása.

Először egy *Result* osztályra van szükség, amely egy-egy program futása során az abban keletkezett kimenet sorait tárolja. Ezt követően, a program futtatásához elkészült a *Program* nevű interfész. Ez az interfész egy absztrakt osztályként van implementálva, mivel mindenképp szükséges azt tárolnia, hogy helyileg hol van a futtatandó program a fájlrendszerben. Egy implementálandó metódusa van, a *run* nevű metódus. Ennek visszatérési értéke egy *Result* objektum. Paraméterként megkapja a bemenetet, a maximum byte mennyiséget, amelyet olvasunk a kimenetéből (alapértelmezetten 10 kbyte, amely egy konfigurációs fájlból érhető el. Ez a fájl a **6.5** fejezetben kerül részletezésre.), és egy atomic flaget, amelyet a *run* metóduson belül igazra állítva megszakítható a *run* függvényt meghívó szál futtatása.

A fordításhoz a *CompilerOutput* osztály készült el. Ez tárolja a fordító kimenetét egy *ArrayList<String>* típusú listában, számként a fordító visszatérési értékét, valamint egy *Program* példányt.

A megírt programok fordításához szükséges funkciókat foglalja össze a *ProgramCompiler* interfész, amelynek két implementálandó metódusa van. Az egyik metódus a *compile*, amely egy tömböt kap a lefordítandó forrásfájlok nevével, és egy *File* típusú paramétert, amely a *working_directory* megfelelő almappjára mutat, amely mappa megfelel a megoldandó feladatnak. Ebbe a mappába készül el a futtatható állomány. A másik metódus a *getCached*, amely fordítás nélkül egy már meglévő futtatható állomány jelenlétével példányosít egy *Program* osztályt. Mindkettő egy *CompilerOutput* példányt hoz létre, amelyben a *Program* benne van, a fordító üzeneteivel együtt.

Miután a két fő interfész és az azokat segítő osztályok elkészültek, a *Task* osztály valósítja meg a megoldandó feladat modelljét. Amikor létrejön, az beolvassa a feladathoz tartozó összes adatot, illetve kap egy *ProgramCompiler*

típusú paramétert is, amellyel a feladat lefordítható lesz. Szükséges volt még létrehozni a *User* osztályt is, amely tárolja a bejelentkezett felhasználó adatait.

A harmadik fő interfész a *TaskFactory*. Ez az interfész foglalja össze az összes eddig definiált interfészt és osztályt. A *TaskFactory* interfésznek egy metódusa van, amely a „factory method” tervezési minta alapján működik. Ez a metódus a *getTask*. Ez a metódus generálja le a feladat modelljét, vagyis a *Task* osztályból készít egy példányt. A *TaskFactory* feladata, hogy létrehozza azt a *ProgramCompiler* típusú osztályt, amelyért ő maga felel. Ez programnyelvenként eltérő implementáció.

6.2.4 C programnyelv implementációja

A három fő interfész létrehozása, illetve az azokhoz tartozó létfontosságú osztályok után C programnyelv támogatása céljából létre kell hozni mind a három interfész implementációját. Ebben az alfejezetben részletezésre kerül, hogy milyen módon kell implementálni a három interfészt, illetve milyen elnevezési konvenciót kell követni, hogy azok működőképeseek legyenek.

Amikor a szoftver elkészíti a feladat modelljét, először el kell döntenie, hogy arra melyik *TaskFactory* implementációt használja fel. Erre készült a *TaskFactoryDecider* osztály. Az osztálynak egy metódusa van, a *decideFactory*, amely egy programnyelv stringből létre tudja hozni a megfelelő *TaskFactory* példányt. A *TaskFactory*-t implementáló osztályoknak ezért követniük kell egy elnevezési konvenciót, mivel a megfelelő osztályt az eldöntő osztály string konkatenációval dönti el. Ezzel a megoldással a *TaskFactoryDecider* osztály módosítás nélkül tud később újabb osztályokat betölteni. Egy gyakorlati példán keresztül ez azt jelenti, hogy ha a *decideFactory* függvény egy „C” String paraméterrel kerül meghívásra, akkor egy olyan *TaskFactory*-t implementáló osztályt fog példányosítani, amely a *TaskCFactory* nevet viseli. Vagyis a „Task” + nyelv + „Factory” típusú osztályt fogja betölteni dinamikusán. Az elnevezési konvenció mellett fontos, hogy ezt Java-ban csak egy adott package-ből tehetjük meg. Az ezen funkcióhoz tartozó package-nek a neve a „*hu.szeba.hades.main.model.task.taskfactory*”, vagyis az összes, az elnevezési

sémát követő *TaskFactory* implementációnak ebbe a package-be kell tartoznia. Ha nincs ilyen osztály, a metódus kivételt dob.

A következő lépés a *ProgramC* osztály implementálása volt, amely a *Program* interfész metódusát implementálja. Az interfész *run* metódusának implementálásában a korábban létrehozott, streamek olvasását segítő *StreamUtilities* osztályt használja, amellyel könnyedén olvasható a futtatott folyamatok kimenete. A *run* metódus paraméterben megkapott inputját a szoftver ráírja a futtatott folyamat input streamjére, illetve egy *Result* osztályba elmenti a folyamat beolvasott output streamjének sorait. Az olvasás után leállítja a folyamatot, majd a függvény visszatér az eredményekkel, vagyis a *Result* osztály egy példányával, amelyben a program futása során keletkezett kimenet sorai vannak tárolva.

A *ProgramC* osztály létrehozása után szükség volt egy fordítóra is. Ez a *ProgramCompilerC* osztály, amely a *ProgramCompiler* interfész implementációja. A *compile* metódusban meghívja a gcc fordítót, és annak argumentumként megadja a forrásfájlokat. A kimeneti fájlból elkészíti a *ProgramC* osztály egy példányát, amelyet egy *CompilerOutput* típusú objektumban tárol el. Ha a fordítás sikertelen volt, vagyis a visszatérési érték nem 0, akkor *null* értéket tárol el a *ProgramC* objektum példánya helyett. A *getCached* függvény hasonló elven készült el, amely egy olyan kimenetet generál, amelyben nincsenek fordító üzenetek, és a visszatérési érték 0. Ez a metódus fordítás nélkül hozza létre a *ProgramC* példányt, amennyiben az már létezik a fájlrendszerben, vagyis korábban már le lett fordítva.

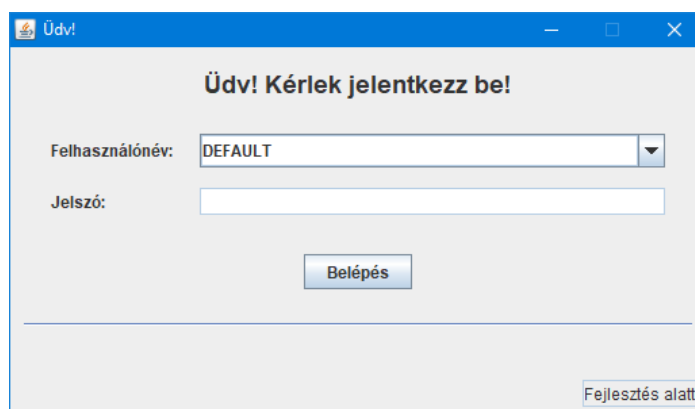
Utolsó lépésként a *TaskCFactory* osztály került implementálásra, amelyet a *TaskFactoryDecider* készít el, amennyiben az elnevezési sémának megfelelően ez lehetséges. A *getTask* metódusa segítségével példányosít egy *Task* osztályt, vagyis a feladat modelljét, illetve a hozzá tartozó fordító osztályt, vagyis egy *ProgramCompilerC*-t.

Az elnevezési konvenciót csak a *TaskFactory*-t implementáló osztályoknak kell követniük, viszont nekik kötelező, máskülönben nem fog működni a dinamikus betöltés, valamint ezeknek az osztályoknak a

„*hu.szeba.hades.main.model.task.taskfactory*” packageben kell lenniük. Ezzel a megoldással új programnyelv támogatásakor elegendő a *TaskFactory*, *ProgramCompiler*, és *Program* interfészeket implementálni, a kódban mást nem kell módosítani.

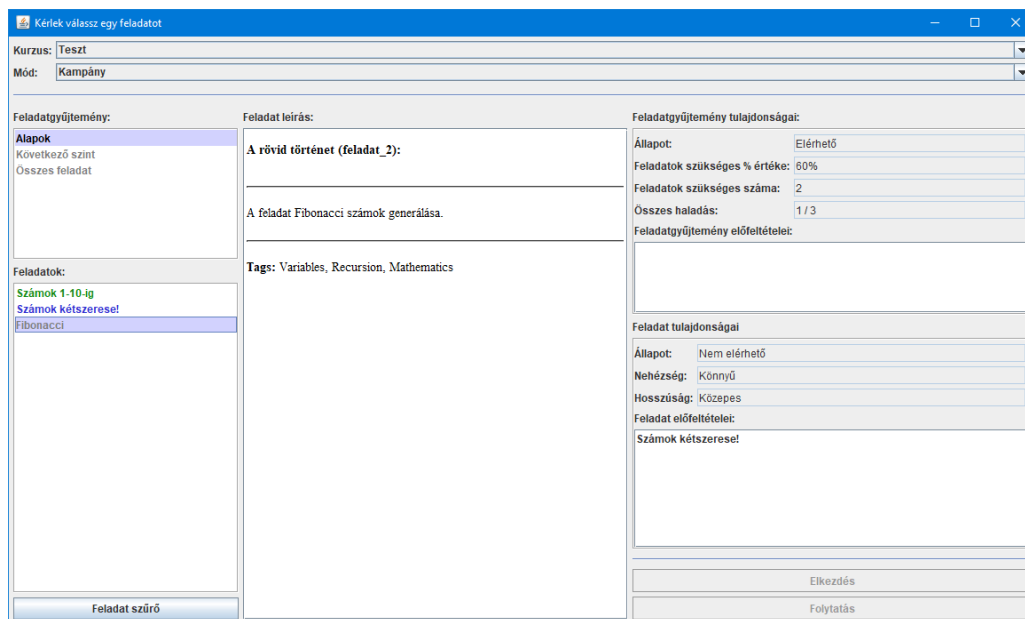
6.3 Feladatmegoldó felület megvalósítása

A fájl alapú adatbázis struktúra megvalósítása, valamint a főbb osztályok, és interfészek létrehozása után a szoftver egyik legfontosabb részének implementációja következett, vagyis a grafikus kezelőfelület, illetve annak összes funkcionalitásának megvalósítása az 5.2 fejezetben ismertetett előzetes koncepciók alapján.



6.9 ábra: a bejelentkező képernyő implementációja

A bejelentkező képernyő a 6.9. ábrán látható. A jelszó mező a szoftver jelenlegi verziójában nem érhető el, de a felületen szerepel annak érdekében, hogy a későbbiekben támogassa ezt a funkciót, amennyiben megvalósításra kerül. A belépés gomb megnyomása után a kiválasztott felhasználóhoz tartozó mappa struktúrát legenerálja, ha az még nem létezik a *working_directory* mappában. Ha létezik, akkor nem generál a mappába új adatot. A *User* osztály egy példánya elkészül, ami a **Felhasználónév** címkéjű mező felhasználónevet kapja meg értelemszerűen. A szoftver ez után átirányít minket a kurzus böngészőbe, ami a 6.10 ábrán látható.



6.10 ábra: a kurzus böngésző implementációja

A kampány, és mód kiválasztása a felső sávban történik, bal oldalon pedig a feladatgyűjtemény kiválasztása (felül), illetve a feladat kiválasztása (alul) található. A nem elérhető elemek a feladat, illetve feladatgyűjtemény listában szürkével vannak jelezve, a teljesített elemek zölddel. A kézzel jelzett feladatok megoldása folyamatban van, már el vannak kezdve, azokat a felhasználó akár előről is elkezdheti az Elkezdés gombbal, de akár folytathatja is a Folytatás gombbal. A középen látható leírás HTML tartalomként jelenik meg, A legfelső sávban a rövid történet, alatta a rövid feladatléírás, és a legalsó szekcióban a feladathoz tartozó címkék találhatóak.

Az információs terület, amely a képernyő jobb szélén látható, több fontos információt is tartalmaz a kijelölt feladatgyűjteményről, illetve feladról. A **6.10** ábra kiemelt részlete, az információs terület a **6.11** ábrán látható.

Feladatgyűjtemény tulajdonságai:	
Állapot:	Elérhető
Feladatok szükséges % értéke:	60%
Feladatok szükséges száma:	2
Összes haladás:	1 / 3
Feladatgyűjtemény előfeltételei:	
Feladat tulajdonságai	
Állapot:	Nem elérhető
Nehézség:	Könnyű
Hosszúság:	Közepes
Feladat előfeltételei:	
Számok kétszerese!	

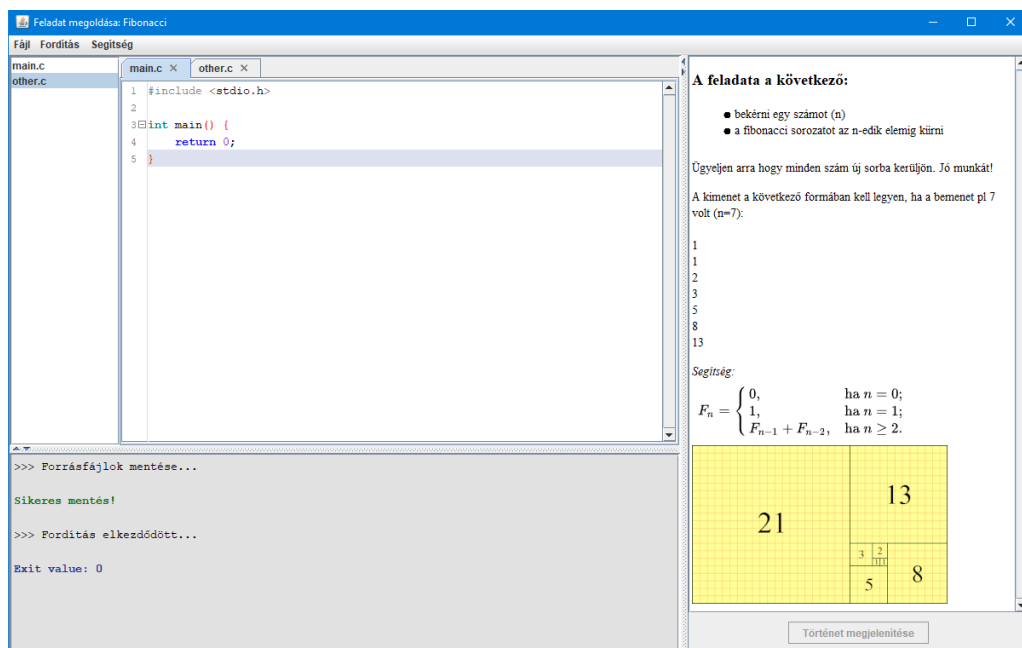
6.11 ábra: a kurzus böngésző információs területe

A kiemelt példán látható, hogy a **6.10.** ábrán kijelölt feladatgyűjtemény elérhető, és a feladatok 60%-át kell abból megoldani, hogy az teljesített legyen. Ehhez kettő feladat szükséges, illetve az is látható, hogy a felhasználó már egy feladatot teljesített korábban. A kijelölt „Fibonacci” nevű feladat még nem érhető el, ugyanis még nem oldotta meg a felhasználó az előkövetelményét, vagyis a „Számok kétszerese” nevű feladatot. Az is látható a **6.10** és **6.11** ábrán, hogy a feladat nehézsége könnyű, a hosszúsága pedig közepes. A bal oldalon található listák alatt egy „Feladat szűrő” címkéjű gomb is található. Ez lehetővé teszi a felhasználónak, hogy az adott feladatgyűjteményben található feladatokat különböző feltételekkel leszűrje, ezzel keresési lehetőséget biztosít. A gombra rákattintva a **6.12.** ábrán látható szűrő jelenik meg.

Szűrések beállítása a feladat listához	
Feladat neve:	
Nehézség:	Összes
Hosszúság:	Összes
Állapot:	Összes
<input checked="" type="checkbox"/> Variables <input checked="" type="checkbox"/> Control flow <input checked="" type="checkbox"/> Recursion <input checked="" type="checkbox"/> Mathematics	<input type="button" value="Összes"/> <input type="button" value="Egyik sem"/>
<input type="button" value="Oké"/>	

6.12 ábra: a kurzus böngésző feladat szűrő képernyője

Ahogy az a **6.12** ábrán is látható, a feladatokat név, nehézség, hosszúság, állapot (teljesítve, elkezdve, nem elérhető) lehet szűrni, illetve tageket, vagyis kulcsszavakat is lehetőség van beállítani. Amikor a felhasználó megtalálta a számára megfelelő feladatot, amennyiben elérhető, azt elkezdheti vagy folytathatja. A gomb megnyomására a *TaskFactoryDecider* a feladat által tárolt programnyelv string alapján eldönti, hogy melyik *TaskFactory*-t fogja használni a feladatmodell legenerálásához. Ha a feladatot az Elkezdés gombbal indítja, és már korábban is dolgozott rajta, akkor megkérdezi a szoftver, hogy biztosan újratekdi-e a feladatot. Már korábban elkezdett feladat esetén a Folytatás gomb is elérhető, amely a már elkezdett fájlokat használja a munkaterületről. A kapott *TaskFactory*-t implementáló objektummal létrejön a feladatmodell, vagyis a *Task* objektum, és a hozzá tartozó *ProgramCompiler* interfészt implementáló, a fordítót reprezentáló objektum. Miután ez megtörtént, a **6.13** ábrán látható képernyő jelenik meg.



6.13 ábra: a feladat megoldó képernyő

A program legfontosabb képernyője, ahol az idő nagy részét fogja tölteni a felhasználó. Felül található egy menü sáv három menüponttal. A Fájl menüben hozhatunk létre új forrásfájlt, menthetjük a megoldásunkat, vagy törölhetjük az alul látható terminál tartalmát. A Fordítás menüpontban tudjuk fordítani a programot, illetve futtatni és ellenőrizni. Ebben a menüpontban van még lehetőség arra, hogy a felhasználó leállítsa a megírt programot, ha például több teszt is van,

de már az elsőnél látszik, hogy nagyon rossz a megoldás. A Segítség menüben található egy menüpont, amely elérhetőségeket jelenít meg, például e-mail címet, ahol az oktatóval fel lehet venni a kapcsolatot. A bal oldali listában található az összes elérhető forrásfájl, valamint középen egy szintaxis-színezett kódszerkesztő felület bezárható fülekkel. Ha egy fület bezárunk, a fájl újra megnyitható a forrásfájl listában a nevére kétszer kattintva. A forrásfájl listában jobb egérgombbal kattintva egy helyi menüből lehetőség van a fájlokat törölni, illetve azokat átnevezni. Jobb oldalon található a feladat leírása HTML tartalomként.

A jobb alsó sarokban található a „Történet megtekintése” gomb. Ha a feladathoz tartozik történet, akkor ennek a gombnak a megnyomásával egy felugró HTML tartalmú ablakban azt bármikor meg lehet tekinteni. A feladathoz tartozó történet a feladat megnyitásánál is megjelenik egy felugró ablakban, a gomb csupán arra szolgál, hogy azt később is meg lehessen tekinteni.

A fordítás, vagy futtatás menüpontra kattintva egy háttér szál (Thread) végzi el a megfelelő feladatot, hogy a grafikus kezelőfelület továbbra is reszponzív maradjon. A futó szál folyamatosan küldi az üzeneteket a terminál felületre, így azokat valós időben el lehet olvasni.

Fordítás esetén a *ProgramCompiler* interfész *compile* metódusát végzi el a háttérszál. Futtatás esetén a *Program* interfész *run* metódusát végzi el. Ezek bármikor megszakíthatóak futtatás közben a „Leállítás” gomb megnyomásával a „Fordítás” menüpontból. Fordítás során a terminálban az adott fordító standard outputra írt üzenetei láthatóak, valamint annak visszatérési értéke is megjelenik. Például helyes fordítás esetén nincs fordító üzenet, a visszatérési érték pedig 0. Ha a fordítás nem helyes, arról is tájékoztat a terminál felület.


```
>>> Forrásfájlok mentése...

Sikeres mentés!

>>> Fordítás elkezdődött...

sources/main.c: In function 'main':
sources/main.c:5:1: error: expected ';' before '}' token
    }
    ^
Exit value: 1
```

6.14 ábra: a terminálon olvasható üzenet, ha a fordítás sikertelen gcc esetében

Ha a fordítás sikeres, akkor a program futtatható lesz. A megírt programot a „Futtatás” gomb megnyomása után a szoftver többször is lefuttatja, az összes tesztesetre külön-külön. Az hibás sorokat a terminál felületen jelzi piros színnel, és azt is, hogy a kimenet melyik sorban hibás.

A 6.15 ábrán szemléltetett példában látható feladat egy szám bekérése, majd a Fibonacci sorozat n-edik eleméig kiírni azt, minden számot új sorba írva. A feladathoz kettő darab teszteset tartozik. Az első teszteset során $n=3$, a második esetben $n=8$, vagyis az kettő input fájlban 3, illetve 8 szerepelnek. Jól látható a 6.15 ábrán, hogy ha a Fibonacci sorozat kiszámítását nem sikerült implementálni, hanem csak az egyik teszteset megoldását programozta le a felhasználó egyszerű kiírásokkal, akkor csak arra a teszt esetre lesz hibátlan a megoldás, a többire nem.

```
>>> Program futtatása...

>>> Bemenet: file1.input

> Kimenet:
1. 1
2. 1
3. 2

>>> Bemenet: file2.input

> Kimenet:
1. 1
2. 1
3. 2

* különbség ebben a sorban: 4. "" -> ennek kéne lennie "3"
* különbség ebben a sorban: 5. "" -> ennek kéne lennie "5"
* különbség ebben a sorban: 6. "" -> ennek kéne lennie "8"
* különbség ebben a sorban: 7. "" -> ennek kéne lennie "13"
* különbség ebben a sorban: 8. "" -> ennek kéne lennie "21"

> Összesen 5 hibás sor, és 0 alkalommal nem válaszolt

...Futtatás vége!
```

6.15 ábra: több teszteset futtatása

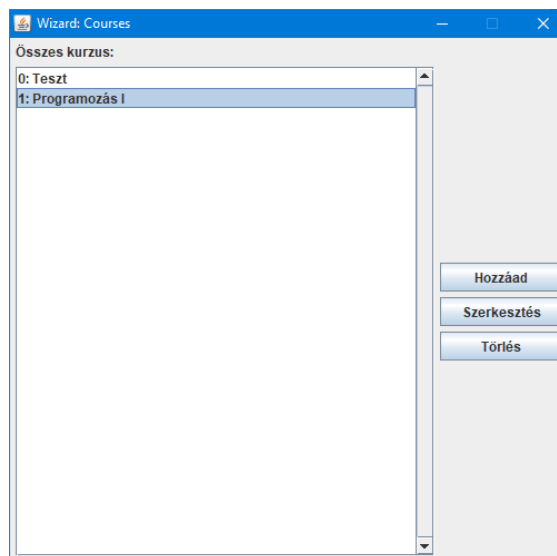
Itt a megírt program fixen az 1, 1, 2 számokat írja ki a kimenetre. Ez az első teszt esetre tökéletesen jó, viszont a második teszt esetre nem, mert ott $n=8$ -ig kell a Fibonacci sorozatot kiíratni. A szoftver jelzi, hogy az 1, 1, 2 sorozat után az összes üres sorban milyen értéknek kellene szerepelnie. Az összes tesztet futtatása után összegzi az eredményt, hogy a megoldás hány helyen adott hibás kimenetet, illetve, hogy az hányszor nem válaszolt, vagyis hányszor történt végtelen ciklus, vagy memóriacímzési hiba például C esetében.

Amennyiben a megoldás hibátlan, azt a terminálon szintén jelezni fogja a szoftver. Ez után kilépve a feladatmegoldóból, vissza a kurzus böngésző ablakba, az összes feladat, vagy feladatgyűjtemény, ami a most megoldott feladatra épül, elérhetővé válik.

6.4 Feladatkészítő felület megvalósítása

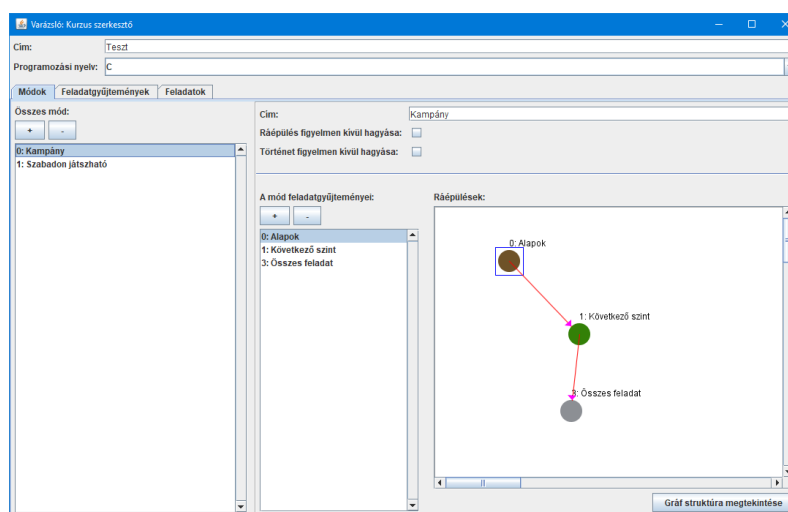
Fontos követelmény volt, hogy a feladatbank is könnyen szerkeszthető legyen, azt könnyen fel lehessen tölteni feladatokkal. Mivel az adatbázis fájl alapon van tárolva, és strukturált szöveges fájlokban van tárolva az adat, ezért azok manuálisan is szerkeszthetők, viszont az adatok konzisztenciáját megtartani nehéz, illetve indexek kezelése is meglehetősen problémás ilyen formában. A sok hibalehetőség, és kényelmetlenség elkerülése érdekében szükséges volt egy felhasználói felület biztosítása, amelyben segítségével könnyedén bővíthető, illetve szerkeszthető a feladatbank. A feladat megoldó felület megvalósítása után így ez lett az utolsó mérföldkő a szoftver megvalósításában.

A feladatkészítő felület alapértelmezetten nem érhető el a szoftver hagyományos úton való indítása után, azt egy külön argumentummal kell kiegészíteni, vagyis a „-wizard” argumentummal kell elindítani a JAR fájlt. Ha ezzel az argumentummal indítjuk a szoftver, a **6.16** ábrán látható képernyő látható a hagyományos belépő képernyő helyett.



6.16 ábra: feladatkészítő felület kezdőképernyője

A felületen megjelenik az összes kurzus, amely az adatbázisban megtalálható. A jobb oldalon található gombokkal hozzáadható új kurzus, módosítható egy meglévő, vagy akár törölhető is. Hozzáadás esetén létrejön egy új azonosítójú kurzus a soron következő számmal, így annak értéke mindig egyedi lesz. Egy új kurzushoz generálódik egy alap mappa struktúra, és egyetlen feladatot, feladatgyűjteményt, vagy módot sem tartalmaz alapértelmezetten. A kurzus szerkesztésére egy új képernyő jelenik meg, amelyben szerkeszthetők a kurzus főbb adatai, mint a címe, illetve a programnyelve. Egy fülekkel tagolt felületen szerkeszthetők a kurzushoz tartozó módok, feladatgyűjtemények, illetve feladatok.

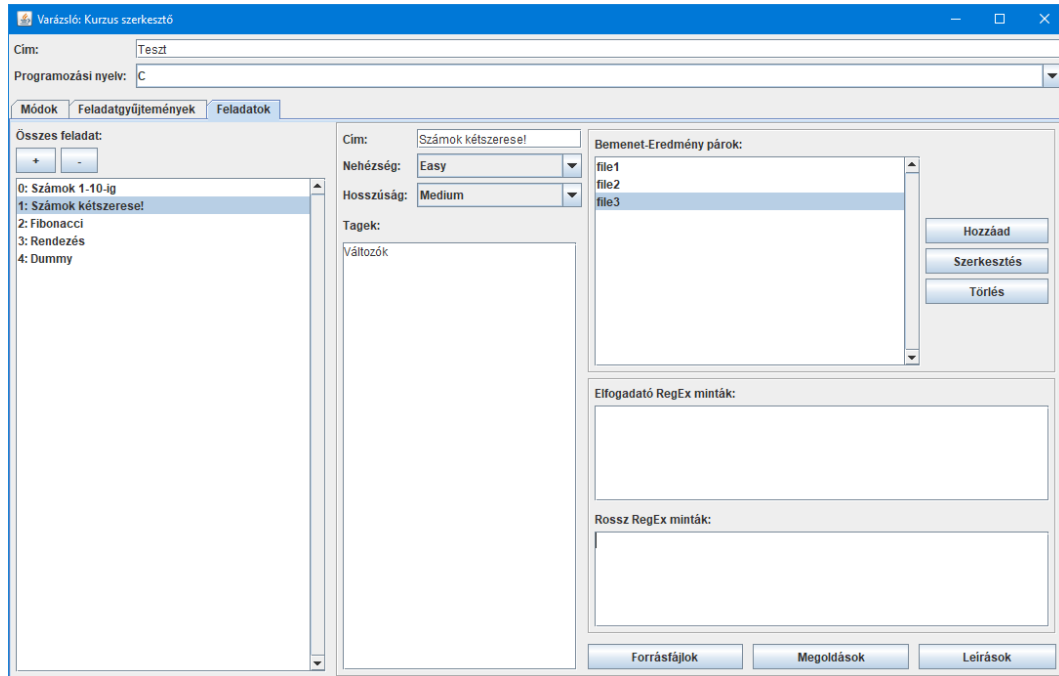


6.17 ábra: kurzus, illetve alul a módok szerkesztése felület

A 6.17. ábrán látható egy kurzus, ami a szerkesztés gomb segítségével nyitható meg. A fenti sávban a kurzus nevét és programozási nyelvét lehet megadni. Középen egy fülekkel tagolt felületen a módok szerkesztése látható. A bal oldali listában láthatóak a módok. Egyet kiválasztva annak adatait jelennek meg jobb oldalon, szerkeszthető formában. A feladatgyűjtemények között a ráépülések beállításához egy grafikus gráfszerkesztő áll rendelkezésre. Az adatbázisban a *graph* kiterjesztésű fájl mellett a felület a grafikus adatokat egy *graph.view* kiterjesztésű fájlban tárolja el, amelyben a csúcsok színe és pozíciója van tárolva. A gráfszerkesztő felület mellett található lista segítségével lehet feladatgyűjteményeket hozzáadni a kurzushoz. Ezt követően a listában rá kell kattintanunk egy feladatgyűjteményre, majd bal egérgombbal lehet azt a csúcsot letenni a gráfszerkesztőbe, ha az a csúcs még nincs benne. Ha benne van, akkor bal egérgombbal át lehet helyezni azt máshová. Másik csúcsra kattintva arra az új csúcsra változik a kijelölés. Egy aktuálisan kijelölt csúcsból a jobb egérgombbal lehet éleket adni abból más csúcsokba. Fontos megjegyezni, hogy egy csúcs önmagára nem tartalmazhat élt, illetve ha egy csúcsból megy egy másik csúcsba él, akkor abból a csúcsból a szülő csúcsba már nem tudunk élt megadni. Ráépülések esetén önmagára hivatkozásnak, és egymásra hivatkozásnak nincs értelme. A körkörös ráépülést nem figyeli a szoftver, de az könnyen észrevehető a grafikus megjelenésnek köszönhetően. A rajzterület görgethető vertikálisan és horizontálisan is, így nagy ráépülési gráfok is megrajzolhatóak benne. A gráf struktúrája bármikor megtekinthető adjacencia lista formájában a jobb alsó sarokban található gombbal. Ez a korábban említett *graph* kiterjesztésű fájl struktúrájával megegyező módon mutatja meg a gráf szöveges felépítését.

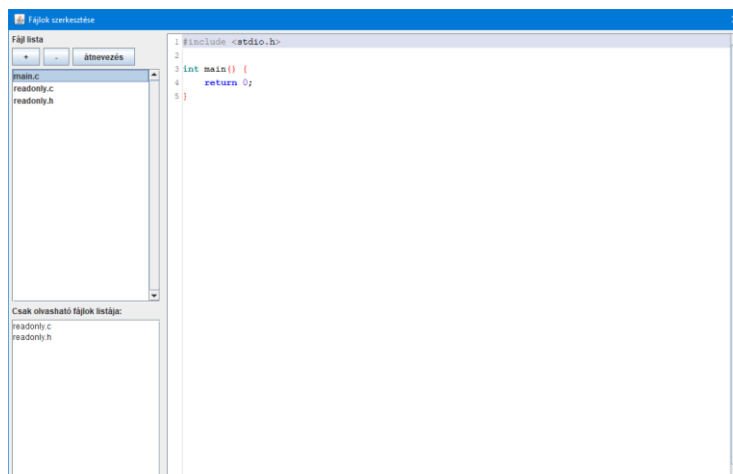
A feladatgyűjtemény szerkesztő szintén hasonló felépítésű, mint a módok szerkesztése, és egy ugyan ilyen gráfszerkesztő felületen lehet megadni a ráépüléseket a feladatok között egy-egy feladatgyűjteményben. A feladatgyűjtemény két fontosabb adatát, vagyis a címét és azt a százalékos értéket lehet szerkeszteni, amely azt jelenti, hogy mennyi feladatot kell megoldani belőle, hogy teljesített legyen a feladatgyűjtemény, és az arra épülő feladatgyűjtemények elérhetőek legyenek a felhasználó számára.

Az utolsó fül a feladat szerkesztő felület. Itt feladatokat lehet hozzáadni, szerkeszteni, illetve törölni egy kurzuson belül. Az alábbi **6.18.** ábrán látható a feladat szerkesztő felület felépítése.



6.18 ábra: feladat szerkesztő felület

A feladat szerkesztő felületen szerkeszthető egy-egy feladat címe, nehézsége, hossza, valamint a címkék egy szöveges szerkesztőfelületen sortörésekkel elválasztva. Bemenet-Eredmény párok is hozzáadhatóak egy-egy feladathoz, amelynek a tartalma egy felugró ablakban szerkeszthető. Ugyanígy egy felugró ablakban szerkeszthetőek a kezdő forrásfájlok a Forrásfájlok gomb megnyomása után, amelynek a felülete a **6.19** ábrán látható. A leírások szerkesztése is egy felugró ablakként van implementálva, amelyben egy színezett HTML szerkesztőben adható meg a feladatok leírása. Fontos megjegyezni, hogy a HTML tartalmakban lehetőség van képek és animációk megjelenítésére is. Ezek a képek és animációk a szoftver mappájában a *resources* mappában találhatóak. HTML-ben a következő módon hivatkozhatunk rá, például egy kép esetén: ``.



6.19 ábra: forrásfájl szerkesztő

Fontos megjegyezni, hogy a kurzus szerkesztése során az adatbázisban fájl szinten nem történik módosítás, csak mentés során, amely a CTRL+S billentyűkombinációk megnyomása után történik meg, vagy a kurzus szerkesztő képernyő bezárására, ugyanis ekkor egy felugró ablakban lehetőség van mentés nélkül kilépni (így a legutolsó mentés óta történt változásokat eldobni), illetve menteni, és kilépni a kurzusból. Ilyen módon, ha valami véletlen, nem kívánt módosítást történt a kurzusban, abból könnyen ki lehet lépni, azok nem lesznek elmentve.

A feladat szerkesztő felület a tervekhez képest még kettő kisebb változtatást kapott, ez pedig az Elfogadható RegEx minták, és a Rossz RegEx minták panel a jobb alsó sarokban, valamint a Megoldások gomb. A RegEx minták felületen Java nyelven értelmezhető reguláris kifejezéseket lehet megadni, amely a feladat megoldás ellenőrzése során a tesztesetek előtt a forrásfájlokon lenne értelmezve. Ez a funkcionalitás a feladatmegoldó felület implementálásába nem került bele idő hiánya miatt, vagyis a reguláris kifejezéseket a szoftver jelenlegi verziója nem veszi figyelembe, viszont a feladatkészítőtől felületen szerepel az opció, mivel az könnyen felhasználható esetleges későbbi fejlesztések során. Ugyanilyen extra nem használt funkció a Megoldások gomb, ahol a feladathoz egy mintamegoldást adható, amit a felhasználó megtekinthetne. Jelenleg nem lehet a mintamegoldásokat megtekinteni, de a jövőben ki lehet bővíteni a programot ezzel a funkcióval.

6.5 Egyéb funkciók

A szoftver implementálása során arra is figyelni kellett, hogy egyéb főbb elérési útvonalak, és kisebb beállítások könnyen módosíthatóak legyenek. A szoftver mappájában található egy *config* nevű mappa, amelyben két *conf* kiterjesztésű fájl található. Ez a *main.conf*, és a *paths.conf*.

A *main.conf* fájlban lehet megadni azt, hogy a *max_stream_byte_count* értéke mennyi legyen. Ez az érték azt jelenti, hogy amikor a feladatokat tesztesetek során futtatja a szoftver, akkor mennyi byte adatot olvasson be abból. Alapértelmezetten ez 10 kbyte.

A *paths.conf* fájlban egyéb elérési útvonalak módosíthatóak. A *database* változó azt jelzi, hogy az adatbázis melyik mappában van. Ez alapértelmezetten egy relatív útvonal, és a „database” értéket veszi fel. A munkaterület megadása a *working_directory* érték módosításával lehetséges, amelynek alapértelmezett értéke a „working_directory” relatív elérésű mappa. Az utolsó érték, amely itt megtalálható az a *compiler_c* amelyet a *ProgramCompiler* interfész C alapú megvalósítása használ fel. Ez a fájl kiegészíthető további felhasználó által definiált értékkel, amelyet szabadon fel lehet használni egy *Options* segédosztály felhasználásával. Az *Options* osztály *getPathTo* statikus elérésű metódusa egy string paramétert vár, és a paraméterrel megegyező változó értékét adja vissza a *paths.conf* fájlból. Ez hasznos további programnyelvek támogatásánál a fordítóprogramok eléréséhez, vagy más útvonalak eléréséhez.

7. Összefoglaló

A szakdolgozat fő célja a követelményeknek megfelelő szoftver megtervezése, valamint az használhatóságához szükséges összes funkció implementálása Java programozási nyelven, objektum orientált módon.

Az elkészült szoftver képes a feladatokat fájl alapon tárolni, azokat kurzusokba, azon belül módokba, majd feladatgyűjteményekbe szervezve. Képes több felhasználót támogatni, és teljes értékű kezelőfelületet biztosít a feladatok böngészésére, keresésére, illetve megoldására. Képes több programnyelvet és technológiát támogatni egyszerű interfészek implementálását követően, vagyis azok külső fordítójával futtatható programot készíteni, tesztesetekkel futtatni, és automatikusan ellenőrizni a feladat megoldásokat. Képes a feladatgyűjtemények, és feladatok közötti egymásra épüléseket kezelni és jelezni. Végző soron pedig felületet biztosít a fájl alapú adatbázis teljes mértékű szerkeszthetőségére, kurzusok hozzáadására és azon belül a módok, feladatgyűjtemények, és feladatok módosítására.

A szoftver olyan elvek alapján készült, hogy a későbbi kiegészítéseket minél nagyobb mértékben támogassa. Ilyen kiegészítések lehetnek a különböző programozási nyelvek, a megoldások bemenet-kimenet összehasonlításnál bonyolultabb ellenőrzése, valamint a példamegoldások megtekintésének lehetősége.

A szoftver jelenlegi állapotában üzemképes, minden szükséges funkcióval rendelkezik ahhoz, hogy az oktatók feltöltsék a feladatbankot, ami után használható lesz a C programozási nyelv elsajátításának elősegítésére. Így a szoftver a 2019/20/1-es szemeszter során már rendelkezésre is fog állni az érintett kurzusok hallgatói számára.

IRODALOMJEGYZÉK

- [1] <https://docs.oracle.com/javase/8/docs/api/> (letöltés dátuma 2019. április 19.)
Java dokumentáció
- [2] <https://en.wikipedia.org/wiki/File:Hades-et-Cerberus-III.jpg> (letöltés dátuma 2019. április 19.) *Hades kép*
- [3] <https://exercism.io/> (letöltés dátuma 2019. február 16.) *Exercism online felület*
- [4] <https://gradle.org/features/> (letöltés dátuma 2019. április 21.) *Gradle projektépítő eszköz leírása*
- [5] <https://learningnerd.com/2015/05/06/exercism/> (letöltés dátuma 2019. február 16.) *Exercism kliens*
- [6] <https://www.codingame.com/ide/puzzle/onboarding> (letöltés dátuma 2019. február 16.) *CodinGame online platform*
- [7] <https://www.crystalinks.com/plutorome.html> (letöltés dátuma 2019. február 15.) *Hades érdekességek*

ÁBRAJEGYZÉK

1.1 ábra: Hades	1
3.1 ábra: CodinGame online felülete (weboldal).....	4
3.2 ábra: CLI kliens az Exercism felülethez	6
4.1 ábra: intelligens statikus kód elemzés IntelliJ IDEA-ban	10
6.1 ábra: Egy graph típusú fájl tartalma	18
6.2 ábra: Egy graph.view típusú fájl tartalma	19
6.3 ábra:graph és graph.view fájl megjelenítve.....	19
6.4 ábra: kurzusokat tartalmazó adatbázis mappa	20
6.5 ábra: Egy feladat fájl struktúrája.....	22
6.6 ábra: A description.xml fájl struktúrája.....	24
6.7 ábra: Egy Feladatgyűjtemény fájl struktúrája.....	25
6.8 ábra: Egy Mód fájl struktúrája	26
6.9 ábra: a bejelentkező képernyő implementációja	35
6.10 ábra: a kurzus böngésző implementációja	36
6.11 ábra: a kurzus böngésző információs területe.....	37
6.12 ábra: a kurzus böngésző feladat szűrő képernyője	37
6.13 ábra: a feladat megoldó képernyő	38
6.14 ábra: a terminálon olvasható üzenet, ha a fordítás sikertelen gcc esetében...	40
6.15 ábra: több tesztet futtatása	40
6.16 ábra: feladatkészítő felület kezdőképernyője.....	42
6.17 ábra: kurzus, illetve alul a módok szerkesztése felület	42
6.18 ábra: feladat szerkesztő felület.....	44
6.19 ábra: forrásfájl szerkesztő.....	45

CD Melléklet

A CD melléklet tartalma:

- A dolgozat docx és pdf formátumban (2 fájl)
- Felhasznált képek mappa (23 fájl)
- Az elkészített program projekt mappája (hades)
 - .gradle mappa (7 mappa, 12 fájl)
 - .idea mappa (5 fájl)
 - .build mappa (38 mappa, 112 fájl)
 - config mappa (1 mappa, 5 fájl)
 - database mappa (34 mappa, 70 fájl)
 - gradle mappa (1 mappa, 2 fájl)
 - libs mappa (1 fájl)
 - out mappa (34 mappa, 112 fájl)
 - resources mappa (5 fájl)
 - src mappa (33 mappa, 93 fájl)
 - themes mappa (1 fájl)
 - working_directory mappa (0 fájl)
 - .gitignore fájl
 - build.gradle fájl
 - gradle_deploy.backup fájl
 - gradlew fájl
 - gradlew.bat fájl
 - LICENSE fájl
 - README.md fájl
 - settings.gradle fájl
- Az elkészített futtatható program mappája (hades release)
 - config mappa (1 mappa, 5 fájl)
 - database mappa (34 mappa, 70 fájl)
 - resources mappa (5 fájl)
 - system mappa (176 mappa, 2788 fájl)
 - working_directory mappa (0 fájl)
 - hades.jar futtatható JAR fájl
 - launcher.bat fájl
 - wizard.bat fájl