Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai tanszék

Nyikes Dániel

TDK Dolgozat– 2016

HALLGATÓI FELADATOKAT AUTOMATIKUSAN KIÉRTÉKELŐ ÉS PONTOZÓ KERETRENDSZER TERVEZÉSE, FEJLESZTÉSE ÉS BEVEZETÉSE

KONZULENS

Marton József Ernő

BUDAPEST, 2016

Tartalomjegyzék

Összefoglaló **4**
1. Bevezetés **6**
2. Automatizálási eszközök az értékeléshez, vizsgálatuk és használatuk **7**
2.1. Korszerű javító eszközök [1] 7
2.1.1. Értékelő eszközök használatának előnye általánosságban 8
2.1.2. Értékelő eszközök használatának hátránya általánosságban 8
2.1.3. Javító eszközök csoportosítása osztályozási sémák alapján 8
2.1.4. Javítóeszközök jellemzői 12
2.2. A kiértékelő eszközök használatának pedagógiai hatása 14
2.2.1. Kísérleti tanulmány 14
2.2.2. Szükséges faktorok az automatikus kiértékelésben 16
2.2.3. Automatikus értékelés bevezetésekor előjövő problémák 17
2.3. Az AKÉP összehasonlítása egy konkrét eszközzel 18
2.3.1. AJupyterNotebook elemzése röviden [4] 19
2.3.2. Az Nbgrader elemzése [5] 20
2.3.3. Összehasonlítás 22
3. Az AKÉP részletes ismertetése: tervezés és a tényleges alkalmazás bemutatása 24
3.1. AKÉP architektúra és működés 24
3.1.1. Keretrendszer felszíne 24
3.1.2. AKÉP modulok egy-egy mondatban 26
3.1.3. Referenciakezelő 27
3.1.4. Fordító 28
3.1.5. Csatorna feldolgozás 29
3.1.6. Kiértékelő modul 32
3.1.7. Hibakezelő 34
3.1.8. Triggerelő 35
3.1.9. Kommunikáció 40
3.1.10. Adatgyűjtés és elemzés 41
3.1.11. Értékelések megjelenítése 43
3.2. Értékelési mechanizmusok és bővíthetőségük 44

3.2.1. Értékelő függvények működése ... 45
3.2.2. Új kiértékelő függvény bevezetése ... 46
3.3. Modulárisan lecserélhető, bővíthető architektúra ... 47
3.3.1. Modulok kicserélhetősége ... 48
3.3.2. Modulok bővítése ... 49
3.4. Integráció LMS eszközbe ... 50
3.4.1. Preprocesszorok a laborokhoz ... 50
3.4.2. Feladatleírók a feladatokhoz ... 52
3.4.3. Hallgatói feladatok eljuttatása a keretrendszernek ... 53
3.4.4. AKÉP Megjelenítő integrálása a hallgatói munkák értékeléséhez ... 54
3.5. A kialakított szakterületi nyelv ... 54
3.5.1. Előnyök és hátrányok[12] ... 55
3.5.2. A séma főbb elemei és azok szerkezete, funkciója ... 55
4. Eredmények, jövőbeli tervek ... 59
4.1. Tanulságok, tapasztalatok ... 59
4.2. AKÉP Eredmények mérési szempontok alapján ... 60
4.3. Jövőbeli tervek ... 63
5. Jegyzékek ... 64
6. Irodalomjegyzék ... 65
Függelék ... 67

Összefoglaló

Tömegoktatásunk egyik komoly kihívása a nagy számban keletkező feladatmegoldások minősítése. A programozási feladatok értékelése különösen nem triviális feladat, mert számos összetevője van: a kódminőségtől kezdve a specifikációnak megfelelő kimeneten át a szemantika alapos vizsgálatáig. Ezek egy része lefedhető I/O-tesztekkel, de a mérnöki munka összetettsége ennél többet kíván, legalább annyira fontos a megoldás belső szerkezete is. Az így összeadódó értékelési részfeladatok mennyisége hagyományosan nagyon sok emberi erőforrást köt le, amely gépi támogatással jelentősen csökkenthető, a felszabaduló idő pedig olyan oktatási tevékenységre fordítható, ahol valóban szükséges az ember. Ilyen például a gyakori hibák beépítése az oktatásba (megelőzés) vagy a tananyag tematikai fejlesztése. Ezek pedig végül a hatékonyabb tanulást segítik.

A probléma nem újkeletű, ezért mára számos eszköz létezik, azonban a potenciális felhasználóik sokszor nem is hallottak róluk, vagy nincsenek tisztában a felhasználási lehetőségeikkel vagy éppen a licence nem engedi az esetleges átalakítást. Emiatt minden egyes technológia és programozási nyelv esetén sajátot fejlesztenek[1]. Ezek az eszközök az oktatónak rengeteg időt tudnak spórolni azzal, hogy egy értékelés olyan mechanikus lépéseit megtegye, mint a formai ellenőrzés, programok fordítása és futtatása, ill. az előre betáplált tesztesetek végrehajtása. Sok esetben képesek a hallgatóknak előzetes visszajelzést is adni a hibáikról. Hallgatói oldalról az azonnal visszakapott eredmény, egységes javítás, javítási javaslatok és a leadási határidő előtti javítási lehetőség ad ösztönzést a feladatokra adott megoldásokcsiszolására, kódminőség, hatékonyság, robusztusság javítására.

Ezekről az eszközökről egy kiváló és friss áttekintés található az [1]irodalomban. A fő probléma az, hogy az eszközök száma folyamatosan és gyorsan gyarapszik, így egyre reménytelenebb válogatni a megannyi script közül, hogy megtaláljuk azt, ami leginkább lefedi az igényeinket. Ennél előremutatóbb lenne létrehozni egy modulárisan bővíthető keretrendszert, mely alapjaiban tartalmazza a legtöbb fontos funkciót, és az ellenőrzendő programozási nyelvtől függetlenül valósítja meg a képességek mögötti logikát.

Éppen ezért a cél egy olyan keretrendszer tervezése, megvalósítása és bevezetése volt, mely képes ellátni a hallgatói feladatok automatikus értékelése során a feladatok többségét, és lehetővé teszi a funkcióinak bővítését.

A rendszer motivációját és első felhasználását a Szoftver laboratórium 5/Adatbázisok labor tárgyban előkerülő különböző technológiák (SQL, Java, SOA) segítségével elkészített, csaknem 700 hallgató 2800 házi feladatának értékelése adja. Ennek során az elkészült AKÉP (Automatikus Kiértékelő és Pontozó) rendszer már az első évben is jelentős segítséget nyújtott az oktatói oldalnak, a javítási feladatok jelentős részének elvégzése mellettdokumentálva a részletes eredményeket, valamint az eltéréseket az emberi és automatikus értékelések között, ahol a manuális értékelés felülbírálta a gépit.

1. Bevezetés

TDK dolgozatom fő célja egy olyan,programozási feladatok javítását szolgáló keretrendszer elkészítése és bevezetése, melynek legfontosabb tulajdonságai:

- Az oktatói gárda számára egy meghatározott szakterületi nyelv (DSL) segítségével teszi lehetővé feladatlapok előállítását, amely a feladatok szerkezete, szövegezése mellett a teszteteket is tartalmazza (későbbiekben: feladatleíró) [3.5]
- Beérkezett hallgatói munkák elemzése és pontozása platformfüggetlen modellben [3.1.6]
- Eredmények megjelenítése webes felületen [3.1.15]
- Értékelési mechanizmusok analízise (napi riport) [3.1.14]
- Értékelési mechanizmusok ésbővíthetőségük [0]
- Modulárisan lecserélhető, bővíthető architektúra [3.3]
- Integráció LMS (Learning Management System) eszközbe [3.4]

A dolgozat harmadik fejezete részletezi ezeket a pontokat, miközben bemutatjaa keretrendszer felépítését és működését, illetve valódi példával illusztrálja a használatát. Előtte azonban bemutatom az automatizálási témakört, a rendelkezésre állókorszerűjavítási eszközöket.Egy konkrét rendszert részletekbe menően is összehasonlítok az AKÉP-el. Látható lesz milyen előnyei és hátrányai vannak általánosságban és konkrétan egy-egy automatizálási eszköznek.Kísérleti tanulmány alapján ismertetem, hogy ezen eszközök használata hogyan befolyásolja az oktatási mechanizmust és miképp segíti a diákok előrehaladását, milyen faktorok szükségesek az automatizálásból adódóan, és milyen nehézségekkel kell szembenézni egy ilyen rendszer bevezetésekor.Az utolsó fejezetben mérésekkel és elemzésekkel mutatom be a rendszert és a tapasztalatait.Milyen sebességgel lettek kiértékelve a hallgatók különböző nyelvben és technológiábanbeadott munkái. Milyen hibák és milyen mennyiségben fordultak elő az egyes feladatokban, azon belül az egyes tesztesetekben. Végül a jövőbeli tervek leírásával zárom a dolgozatom, mely meghatározza ennek a projektnek a további lépéseit.

2. Automatizálási eszközök az értékeléshez, vizsgálatuk és használatuk

Már 40 évvel ezelőtt is használtak oktatók olyan eszközöket, amelyek megkönnyítették a programozói feladatok értékelését és ma is van még hova fejlődnie a területnek.Az ilyen eszközök többnyire az olyan ismétlődő és monoton feladatokat képesek automatizálni, mint pl. afordítás, futtatás, statikus kódanalízis.Egyes esetekben ezek az eszközök képesek arra is, hogy a benyújtott megoldásokról azonnal visszajelzést adjanak, ezzel is motiválva a hallgatókat a feladataik még beadás határidő előtti javítására (pl. a statikus kódelemzés úgy találta, hogy egy feltételrendszerük feleslegesen bonyolult, egyszerűsíthető ismerve a bool algebrát).[1]
A 2.1 fejezet bemutatja milyen lehetőségek állnak rendelkezésre, különböző szemszögekből vizsgálva az eszközök működését, előnyeit, hátrányait. A 2.2 fejezet egy tanulmányi kísérlettel mutatja meg, hogy a tanulók szempontjából milyen hatást fejt ki az automatikus kiértékelés és visszacsatolás, továbbá milyen szükséges faktorok kellenek egy ilyen rendszer bevezetéséhez és milyen nehézségek/problémák adódnak ekkor. Végül 2.3 fejezet egy összehasonlítást mutat az AKÉP és a jupyternbgrader között.

2.1. Korszerű javító eszközök [1]

Nem triviális feladat a programozói feladatok kiértékelése, mert számos jellemző vizsgálata szükséges, miközben már a hallgatók munkájának fordítása és futtatása is sok erőforrást igényel. Amennyiben a diákok száma nagy, a helyzet még rosszabb és a kiértékelés is menedzselhetetlenné válik. Ennek veszélye, hogy az oktatók nem vizsgálják kellő részletességgel (több szempontból) a beadott munkát vagy valamilyen felületes értékelést végeznek (pl. mintavételes javítás). Ez pedig a hallgatói munkákra adott részletes visszacsatolás rovására megyegyre inkább megfosztva a tanulókat attól a lehetőségtől, hogy tanuljanak a hibáikból. Ezen limitációk eltüntetésének/csökkentésének érdekében számos eszköz készült. A következő alfejezetekben leírt ismeretek, csoportosítási sémák, 30 programozási feladatokat javító eszköz ([1] IX táblázat)vizsgálata után alakult ki.

2.1.1. Értékelő eszközök használatának előnye általánosságban

Az eszközök hasznosak mind oktatói, mind diák oldalról. Az oktatók szemszögéből az eszközök elvégzik az olyan ismétlődő feladatokat, mint a fordítás, a futtatás, tesztesetek futtatása. Más szavakkal az adott eszköz, valamilyen bemenettel indukálja a vizsgált alkalmazást (vagy annak részeit) és vizsgálja, hogy a kapott kimenet megfelel-e a definiált elvárásoknak vagy sem. Sok esetben (megfelelő konfigurációval) értékel olyan jellemzőket is, mint modularitás foka, kódstílus, dokumentáltság, hatékonyság. Tanuló szemszögéből a jó visszacsatolások segítik a hibák megértését, azok későbbi elkerülését. Vannak eszközök, melyek engedik is a tanulóknak, hogy javítsák beadott munkájukat még a határidő előtt és azonnali visszajelzést kapnak, sikerült-e a problémát orvosolni. Ebből adódóan a tanulók motiváltak feladataik folyamatos javításában.
Az eszközök segítségével kialakított kiértékelés gyorsasága, egységessége, pontossága, könnyű érthetősége, továbbá hogy minden esetben konzisztens, mindkét fél számára jelentős előrelépést jelenthet.

2.1.2. Értékelő eszközök használatának hátránya általánosságban

Amegfelelő segítő eszközöket általában nehezen lehet megtalálni (melyik eszköz mire képes, hogyan alkalmas javítási mechanizmusok megoldására). Az oktatóknak az eszközöket olyan szinten meg kell ismerniük, hogy integrálni tudják saját környezetükbe, mely különösen nehéz, ha nincs pontos egyeztethetőség az eszköz lehetőségei és a kurzusuk módszertana között. További problémát jelenthet az eszköz használatának megtanítása a diákoknak és az eszközre épülő feladatokat előállítani (Erre saját tapasztalatból származó példákat adok a 4.1 fejezetben).

Azonban ezek a hátrányok kezelhetők, egyrészt a következő fejezetekben leírt eszközök csoportosítása segítségével, másrészt a dolgozatban 3.5, 3.1.6, 3.3.2 fejezetekben kifejtett szakterületi nyelv kialakításával, technológia független tesztelési lehetőséggel, és bővíthetőséggel. Ezen kívül ugyancsak hátrány, hogy a legtöbb kiértékelő eszközt nem LMS-el történő integrációra tervezték, ahogy az látható az [1] forrásX. táblázatának I5-ös oszlopában (ld. még a 3.4 fejezetben).

2.1.3. Javító eszközök csoportosítása osztályozási sémák alapján

Három fő dimenzió alapján lehet osztályozni a kiértékelő eszközöket. A kiértékelés **típusa**, milyen központú **megközelítés**t alkalmaz és **specialitás**ok.

A **kiértékelés típusa** alapján három kategória különböztethető meg:

- **Kézi értékelés:** Azon eszközök, melyek segítséget nyújtanak az oktatónak a feladatok javításában, de az értékelés manuális marad. Erre példa a [1] forrásban bemutatott T25-ös eszköz, mely a hallgató gépén fut le, majd a fordítási és futási információkat beküldi az oktató szerverének, ahol azt az oktató manuálisan ellenőrzi. A dolgozatomban a 3.1.5.2 fejezetben ismertetett preprocesszorok is hasonló szerepet töltenek be, oly módon hogy kapcsolatot teremtenek a hallgató munkájával és a kigyűjtött információkat továbbítják.

- **Automatikus értékelés:** Azon eszközök, melyek automatikusan képesek az értékelést elvégezni. Ehhez azonban előtte bizonyos konfiguráció szükséges.Több eszköz esetén is az oktatónak meg kell adnia az értékelés módját, pontozását

- **Fél automatikus értékelés:** Azon eszközök melyek a felső két kategóriát ötvözik. Jó példa erre az olyan eszköz, amely megvizsgálja, hogy a beadott munka a definiált inputokra az elvárt kimenetet szolgáltatja-e. Amennyiben nem, akkor kéri az oktatót, hogy manuálisan értékelje a munka érintett részét.

A 1. Táblázat ezen kategóriák erősségeit és gyengeségeit mutatja be, különböző szempontok alapján.

Kategória	Be- és kimenet formátuma	Oktató által adott részletes értékelés	Szisztematikus értékelés	Idő és erőfeszítés csökkentése
Kézi	A tanuló munkájának nem kell követnie megszabott bemeneti és kimeneti formát. ▲	Teljes mértékben támogatott. ▲	A kiértékelés nem szisztematikus. ▼	Az oktatónak nem sokat segít. ▼
Automatikus	A tanuló munkájának muszáj követnie bizonyos megkötéseket. ▼	Semmilyen esetben sem támogatott. ▼	A kiértékelés szisztematikus. ▲	Az oktatónak erősen segít. ▲
Fél automatikus	A tanuló munkájának muszáj követnie bizonyos megkötéseket. ▼	Részben támogatott. ◆	A kiértékelés részben szisztematikus. ◆	Részben segít. ◆

Jelzések: ▼ Gyengeség ▲ Erősség ◆ Részben erősség/gyengeség

1. Táblázat A kiértékelés típusa szerinti osztályozások összehasonlítása. Forrás: [1]irodalom VI. táblázatának fordítása.

A következő dimenzió, a**megközelítés osztályozása**azt határozza meg, hogy a kiértékelési folyamatot mi kezdeményezi (triggereli). Ennek megfelelően három kategória van:

- **Oktató központú**: Ezek az eszközök az oktató indítási parancsára kezdenek neki az értékelésnek. Amikor az oktató jelez, az eszköz kiértékeli a beadottmunkákat

és megmutatja az oktatónak az eredményt. Ezt követően az oktató felülvizsgálhatja az eredményeket, majd továbbíthatja a tanulóknak.

- **Diák központú**: Ebben az esetben a hallgató beadása kezdeményezia kiértékelést. Rendszerint az oktató definiálja a kiértékelés specifikációját és egyéb értintett paramétereket (teszt adat, program referencia, stb.). A hallgatók így megnézhetik a kiértékelési specifikációt és ezek alapján tervezhetnek és adhatnak be megoldásokat. Minden beadásnál az eszköz összehasonlítja a hallgató megoldását az oktató által adott paraméterekkel. Amikor a kiértékelés végez, mind az oktató mind a hallgató megkapja az eredményeket.
- **Hibrid megközelítésű**: Ezeket az eszközöket olyan stratégiák alapján implementálták, hogy az előző két kategória erősségeit átvegyék. Az itt azonosított fő stratégiák:
    - **Előzetes validáció**: Ebben az esetben az oktató indítja a kiértékelési folyamatot, ugyanakkor ha a tanuló kíváncsi, hogy munkája jó irányba halad-e, elküldheti kiértékelésre és ilyenkor az eszköz pár tesztet (nem az összest) lefuttat rá és visszaküldi ezek eredményét.
    - **Részleges visszajelzés**: A munka verifikálásának csak egy része automatikus. Amikor a hallgató beküldi a munkáját ezekre azonnal kap visszajelzést. Ugyanakkor a nem automatizált feladatok manuális értékelését az oktató csak a határidő elérte után javítja és küldi el a hallgatóknak
    - **Oktatói felülvizsgálat**: Hasonló a diák központú megközelítéshez, azonban itt az azonnal visszakapott eredmények nem véglegesek, az oktatói felülvizsgálat következtében változhatnak.

A 2. Táblázatezeket a megközelítéseket hasonlítja össze különböző szempontokból. Hasonlóan ahhoz, ahogy a típus alapú osztályozás tükrözte a szisztematikus/objektív ill. a mélyebb/szubjektív kiértékelések közötti átmenetet, addig a megközelítés szerinti osztályozás ugyanezt tükrözi az "azonnali visszajelzés a hallgatónak" és az "oktató kezében van a kontrol" kategória között.

Megközelítés kategóriája	Visszacsatolás	Diák részbeadások	A kiértékelés kontrolálása	Hallgatói munkák összehasonlítása
Oktató központú	A hallgató nem kap azonnali visszajelzést. ▼	Az előzetes részbeadások nem mindig támogatottak. ▼	Oktatók tudják teljesen kontrolálni a kiértékelést. ▲	Támogatott. ▲

Diák központú	A hallgató azonnali visszajelzést kap. ▲	Mindig támogatott. ▲	Oktatók nem teljesen kontrolálhatják a kiértékelést. ▼	Nem támogatott. ▼
Hibrid	A tanuló munkájának csak egy részére kap azonnali visszajelzést. ◆	Mindig támogatott. ▲	Oktatók részben kontrolálják a kiértékelést. ◆	Támogatottt. ▲

Jelzések: ▼ Gyengeség ▲ Erősség ◆ Részben erősség/gyengeség

2. Táblázat Megközelítés dimenziója szerinti összehasonlítások. Forrás: az [1] irodalom VII. táblázatának fordítása.

Számos eszköz rendelkezik egyedi specialitással, hogy minden részletre kiterjedő tanulási környezetet teremthessen. A megvizsgált 30 értékelő eszköz alapján három kategória alakult ki. Ezek a következők:

- **Versenystílusú specialitás**: Általánosságban azon eszközök, melyek lefordítják a beadott munkát, majd teszteseteket futtatnak. A végrehajtást követően egy rövid eredményt szolgáltatnak, amiben jelzik, hogy a diák megoldása el lett fogadva vagy sem. Ilyen az[1] forrásban bemutatott T15-s eszköz, mely a következő eredményeket adhatja ki egy megoldásra (és ebben hasonlít is az AKÉP válaszára):
    - Elfogadva: a program a megfelelő kimenetet nyújtott
    - Idő limit túllépése: amennyiben a program nem végzett a megadott idő alatt
    - Formátum, prezentációs hiba: amennyiben a program által adott kimenet közelített az elvárt kimenethez.
    - Rossz válasz: amikor a program nem megfelelő kimenetet szolgáltatott
- **Kvíz stílusú specialitás**: Olyan eszközök, melyeknél a feladat kérdések sorozatára bontott, ahol a diáknak mindig csak az aktuális kérdésre kell az adott kódrészletet előállítania.
- **Szoftvertesztelés specialitás**: Ezen eszközök tesztelik a hallgatók által készített munkát és az erre szintén a hallgatók által tervezett teszteseteket is. Az ötlet lényege, hogy a tanulók ne csak a programozási alapokat sajátítsák el, hanem tesztelési gyakorlatot is szerezzenek. Az[1] forrásban bemutatott T19 eszköz ezt úgy valósítja meg (és erre az AKÉP csatorna rendszere is lehetőséget teremt), hogy:
    - Összehasonlítja a hallgató programjának kimenetét a referenciaprogram kimenetével.

- Futtatja a hallgató tesztjeit a hallgató munkáján, vizsgálva, hogy azok mennyire fedték le a hallgató programját.
- Futtatja a hallgató tesztjeit a referenciaprogramon, hogy megvizsgálja valóban helyesek-e a tesztek (jól vizsgálják-e az elvárt követelményeket).

Ezek a csoportosítások (ahogy az folytatódik a következő alfejezetben is) segítik az oktatókat az eszközök tulajdonságainak áttekintésében és a céljaiknak leginkább megfelelő eszköz kiválasztásában. Ugyanakkor a különböző eszközök kombinálására, így az érdemeik összeadásáranem igazán van lehetőség. Ezért is szorgalmazott egy keretrendszer kialakítása, melyben tetszőlegesen lehet bővíteni ezen funkcionalitásokat.

2.1.4. Javítóeszközök jellemzői

Az osztályozási sémákat követően az eszközök három jellemző alapján kerültek csoportosításra,amelyek a következők: főbb funkcionalitások, verifikáció típusai, támogatott interfészek, és programozási nyelvek. A **főbb funkcionalitások**:

- **F1**: Elektronikus beadás: Lehetővé teszi ahallgatói munkák beszedését és tárolását. Továbbá bátorítja a tanulókat a munkájuk időben történő beadására.
- **F2**: Automatikus ellenőrzés: Az automatikus és félautomatikus kiértékelő eszközök képesek bizonyos verifikációs tesztek elvégzésére biztosítva az oktatókat a hallgató megoldásának minőségének megfelelő szintjéről, anélkül, hogy az oktatóknak sok időt és erőfeszítést kéne tennie ennek ellenőrzésére.
- **F3**: Automatikus pontozás: A kiértékelési eredmények alapján, az eszközök egy része képes a megoldott feladatok osztályozására, pontozására. Ezzel biztosítják egy tanuló munkájának objektív mérését.
- **F4**: Azonnali visszajelzés: A hallgató központú megközelítésű, illetve a hibrid eszközök képesek a hallgatók beadott munkájáról azonnali visszacsatolást adni, ezzel is bátorítva őket megoldásaik javítására a beadási határidőn belül.
- **F5**: Kérdések gyűjteménye: Lehetővé teszi az oktatónak, hogy a feladatsorok összeállítása egy tárból történő kiválasztással történjen ahelyett, hogy nulláról kellene összeállítani az újat.
- **F6**: Beadások visszamenőleges tárolása: Azok az eszközök, melyek lehetővé teszik az elektronikus beadást, azt is lehetővé teszik, hogy minden egyes beadás

eltárolásra kerüljön. Ez a funkció lehetővé teszi a diákok gondolkodási mechanizmusának elemzését egy adott feladat megoldása során. Például egy hibás beadást követő helyes megoldás alapján vizsgálható, hogy milyen stratégiával javította ki az adott tanuló a munkáját.

- **F7**: Statisztikai riport készítése: Lehetővé teszi az oktatónak, hogy statisztikai adatok alapján vizsgálhassa az osztály/kurzus teljesítményét. Így az oktató képes azonosítani a fő problémákat, hiányosságokat és a későbbiekben ezekre megoldást tud nyújtani, beépítve az oktatásba.

A második a **verifikációs típusok** alapú csoportosítás. Két fő kategóriájaa dinamikus verifikáció, mely esetén szükséges futtatni a tanuló megoldását, illetve a statikus verifikáció, amikor nem kötelező ezt megtenni. A dinamikus alapú csoportosítás:

- **V1**: Program validálása: A diák programja azt a kimenetet nyújtja-e, amit elvárunk a beadott input alapján?
- **V2**: Program teljesítménye: A diák programjának futása nem haladja meg az időlimitet, nem haladja meg a memórialimitet.
- **V3**: Tesztek megfelelősége: A diák által definiált tesztek megfelelnek a tesztelési kritériumoknak (pl. lefedik a program funkcionalitásait)?

A statikus verifikáció alapú csoportosítás:

- **V4**: Komplexitás: A diák által írt algoritmus megfelel-e bizonyos komplexitási metrikáknak?
- **V5**: Fordítható programkód: Kódban nincsenek a fordítást megakadályozó hibák?
- **V6**: Dokumentáció: Megfelelően kommentezett a forráskód? Mérve a comment sorok és a kód sorok közti hányadost.
- **V7**: Megfelel a diák munkája a kért kódolási gyakorlatnak, mint formázás, modularitás, beszédes változónevek?
- **V8**: Eredetiség: A hallgató munkájáttartalmilagvetiössze más munkákkal, detektálja a plagizálást.

Az **interfész típusa** alapján három fő kategóriába sorolták az eszközöket. A vizsgált eszközök közül egy-kettő kiterjesztésként integrálható más környezetbe. A fő kategóriák:

- **I1**: Parancssoros interfész (CLI): Számos eszköz támogatja, azt jelenti, hogy az adott eszköz parancssorosban futtatható és az eredményeket is konzolon jeleníti meg.
- **I2**: Grafikus felület (GUI): A funkciók és eredmények ablakokon keresztül érhetőek el, ikonokkal, menükkel, mutatókkal.
- **I3**: Webes felület (WUI): Az eredmények weboldalakon keresztül érhetőek el és a funkciók elérése valamilyen webszerverhez kapcsolódik.

Kiterjesztés más környezetbe:

- **I4**: Integrált fejlesztői környezet (IDE): A diákok a munkájuk beadását és a kiértékelés eredményét a fejlesztői környezetbe beépítve tudják megtekinteni.
- **I5**: Tanulói menedzsment rendszer (LMS): Mind a tanulók, mind az oktatók az LMS interfészént képesek elérni az értékelő eszköz funkcióit.

Az utolsó, de ugyancsak fontos jellemzője az értékelő eszközöknek, hogy milyen programnyelveket támogatnak. A fenti csoportosításokkal egybevonva az[1] forrás X. táblázata teljes összehasonlítást mutat az összes felsorolt eszközre, beleértve a programnyelveket is

2.2. A kiértékelő eszközök használatának pedagógiai hatása

A 2.1.1 és a 2.1.2 fejezetekben olvasható az automatikus értékelő eszközökhasználatának előnye és hátránya általánosságban véve. Ebben a fejezetben először egy kísérleti tanulmányt írok le, melya hallgatók oldaláról vizsgálja, hogy ők miképp tudják hasznosítani az automatikus értékelésből származó visszacsatolást, pontozást. Fontos tényező, hogy ittdiák központú megközelítésről van szó [2.1.3]. A fejezet másik felében pedig az automatikus kiértékelésbevezetéséhez szükséges faktorokat,nehézségeket ismertetem.

2.2.1. Kísérleti tanulmány

A [2] forrásban található kísérletben 46 hallgató vett részt, közülük egy sem használt még korábban automatikus kiértékelő rendszert. A résztvevők két részre lettek osztva: egy kontrol csapatra és egy kísérleti csapatra. A kísérleti csapat 10 héten keresztül a Mooshak nevű automatikus kiértékelővel dolgozott (és beadásonként folyamatos visszacsatolást kapott), míg a kontrol csapatnak házi feladatokat kellett készíteni, amit az

oktatók manuálisan osztályoztak. Minden hallgatónak C nyelven kellett megoldani különböző problémákat (testing|tesztelés, debugging|hibakeresés, deployment|kitelepítés, versioning|verziókezelés), melyekhez a 3. Táblázat mutatja a felhasználható eszközöket.

Eszközök	Lefedett terület
gcc	Fordítás, statikus analízis, statikus és dinamikus linkelés, statikus és dinamikus könyvtárak készítése
Gcov, gprof	Tesztelés: kód lefedettség, lefutási idők
Gdb	Kivételek kezelése, visszakövetés
Doxygen	Dokumentációk készítése
Assert, nana	Állítások összehasonlítása, logolás
Make	Szoftver fordítása és telepítése
Efence	Dinamikus memóriafoglalási hibák érzékelése
CVS	Verziókezelő rendszer
Autoconf	Konfigurációs fájl generálása
Automake	Makefile generálása
Libtool	Statikus és dinamikus programkönyvtárak készítése és kitelepítése

3. Táblázat[2] forrásban használt eszközök

Mooshak-ban ezekre az eszközökre készítettek tevékenységeket (activities), amit a kísérleti csapat diákjainak el kellett végezni. Az év végén pedig egy vizsgát írattak minden diákkal. A1. Ábra azt mutatja, hogy a vizsgált időszakban a kísérleti csapatban részvevők milyen mértékben, és milyen eredménnyel teljesítették a tevékenységeket.Az, hogy a futásidejű hibák száma nagyobb, mint a rossz válasz szám, abból ered, hogy az esetek többségében egy futásidejű hiba bug javítása után rossz válasz lett az eredmény.A legtöbb újból beadás 15 volt egy adott hallgató és egy adott probléma esetén. Minden probléma súlyozva volt és a hallgatók tetszőlegesen választhattak, nem volt minimális szám meghatározva arra, hogy egy adott (pl. gcc) tevékenységből mennyi problémát kell megoldani.

A vizsgán pedig a 2. Ábra-n jól láthatóan a kísérleti csapat szerzett magasabb pontszámot, bizonyítvaaz automatikus kiértékelőtől származó folyamatos visszacsatolás hatékonyságát, valamint az így jobban megértett problémák eredményességét. A forrásban leírt hallgatói visszajelzések is teljesen pozitívak. A 0-4 skálán az eszközt használók 87%-os visszajelzés mellett 3.6-os átlaggal és 1.2-es szórással értékelték az automatikus értékelést és visszajelzést.

Tool	#p	#s	A	WA	RE	SP
gcc	7	140	55(39.3%)	39(27.8%)	46(32.9%)	20.0
gcov	5	76	31(40.7%)	20(25.3%)	25(32.8%)	15.2
gprof	5	74	32(43.2%)	19(22.3%)	23(31.1%)	14.8
gdb	7	117	56(47.8%)	27(24.7%)	34(31.5%)	16.7
doxygen	4	76	35(46.0%)	17(22.3%)	24(28.2%)	19.0
assert	6	92	45(48.9%)	21(23.5%)	26(29.2%)	15.3
make	8	126	59(46.8%)	32(25.3%)	35(27.7%)	15.7
efence	7	96	44(45.8%)	25(25.2%)	27(28.1%)	13.7
libraries	5	92	40(43.4%)	24(25.0%)	28(30.4%)	18.4
cvs	7	146	68(46.5%)	35(32.4%)	43(29.4%)	20.8
All	61	1035	465(44.9%)	259(25.0%)	311(30.1%)	16.9

#p = problémák száma, #s = beadások száma, A = elfogadott, WA = rossz válasz, RE = futásidejű hiba, SP = #s / #p beadások/probléma átlag

1. Ábra[2] forrásban Mooshak-ban készített tevékenységekre használt eszközök mérési eredményei

Statistics	MN		SD		MD		MDN	
Group	Exp.	Ctr.	Exp.	Ctr.	Exp.	Ctr.	Exp.	Ctr.
TEST	4.78	4.47	2.23	2.31	5	6	5	4
DEBUG	6.04	4.39	2.22	1.94	7	4	6	4
DEPLOY	5.78	4.65	2.23	2.01	5	4	6	4
VERSION	5.43	4.26	2.25	2.00	5	4	5	4

MN = átlag, SD = szórás, MD = mód, MDN = medián

2. Ábrakísérleti csoport és kontrol csoport vizsgaeredménye az egyes teszteknél a [2] forrásban

2.2.2. Szükséges faktorok az automatikus kiértékelésben

A [3] forrás MOOC (Massive Open Online Courses) szempontból emeli ki számos megvizsgált irodalmi kutatás publikált tapasztalatait automatizált kiértékelés terén (melyeknél nagy számú hallgatósággal indítottak kurzust). Ezek a **faktorok** a következők:

- Az első és legfontosabb faktor a **feladatok minősége**. Manuális értékelés során rendszerint a gyengén tervezett feladatsorok kompenzálhatók azzal, hogy a megoldás kreativitását pontozzák. Azonban ez legtöbb esetben nem lehetséges akkor, amikor automatikus kiértékelést vezetnek be. Az automatikus kiértékelés használata jóval gondosabb pedagógiai tervezést igényel mind az értékelési eljárásoknál, mind a feladatok készítésekor.
- A **megfogalmazott követelményeknek jóval precízebbnek kell lennie**, mint manuális értékelés esetén, különös tekintettel az egyértelműségre. Amennyiben a specifikáció nem egyértelmű, megtörténhet, hogy egy helyes megoldást elutasítson az automatizmus, mert arra az értelmezésre nem lett bekonfigurálva.

- **Jól megválasztott teszt adatok**tól függ a beadott munkán elért fedettség az ellenőrzés során, továbbá a kiértékelésből származó osztályozás pontossága. A teszteseteknek jól átgondoltnak kell lennie, hogy megelőzhető legyen a rossz működésű program átengedése. A pontos tesztek elkészítése legalább akkora kihívás, mint a feladat megoldása, ha nem nagyobb. A tesztadatoktekinthetők egy automatikus kiértékelő Achilles sarkának.
- Az értkélési **eredmények visszacsatolásának** fontosságát nem lehet eléggé hangsúlyozni. A folyamatos visszajelzés a diákoknak kulcsfontosságú, így tudnak csak igazán tanulni a hibáikból, javítani a gyengeségeiket. Teljesen automatikus visszajelzést készíteni rendkívül nagy kihívást jelent. Minőségi visszajelzés (nem csak az adott teszt eredménye, esetlegesen érintett függvény hibás kimenete) leginkább csak olyan fél automatikus megközelítésben létezik, ahol az oktató a feladathoz előre rögzített instrukciókat tesz, így azok is megjelennek az egyes tesztek nem teljesülésekor.
- Amennyiben az automatikus kiértékelés bevezetésének a célja a képzés, elengedhetetlen, hogy a diákok **többször beadhassák munkájukat**, így a visszacsatolás révén javíthatják azt. A gyors kiértékelés és visszajelzés növeli a diákok motivációját. Ugyanakkor a korlátlan újra beadási lehetőség nemkívánatos viselkedést eredményez a diákoknál. Ebben az esetben nem koncentrálnak kellőképp a megoldásra és a kiértékelőt tesztelőként használják.
- **A diákok tesztelési kultúrájának fejlesztése** ugyanolyan fontos, mint programozási képességeik javítása. Így lényeges, hogy az automatizált kiértékelés készítésekor ezt is figyelembe vegyük.

2.2.3. Automatikus értékelés bevezetésekor előjövő problémák

Az előző fejezethez hasonlóan itt is a más tanulmányokat áttekintő, összegezett [3] forrásból dolgoztam. A nehézségek/problémák egy automatikus értékelő rendszer bevezetésekor a következők:

- **Speciális kihívás a feladatsorok készítése**, mivel minden kis hiba a definíciók osztályozásában problémához vezethet. Az oktatóknak kezdetben így nem csökken a terheltsége, csak áthelyeződika manuális értékelésről a feladatok és teszte-

setektervezésére és implementálására. Különösen nagy nehézséget jelent, ha a diákoknak megengedett valamekkora tervezői szabadság. Ugyanakkor az erőfeszítés kifizetődő a feladatok és tesztek újra felhasználásával.

- Sajnos az automatizmus érdekében írt precíz feladatok **csökkentik a diákok kreativitását** egy adott probléma megoldásánál. A feladatok helyességét és hatékonyságát legkönnyebben alfeladatokra bontással lehet elérni, ugyanakkor ennek hatása, hogy a diákoknak már nem maguknak kell dekomponálni a problémát.
- **Plágium megelőzése és kezelése** sajnos szintén szükséges, mivel automatikus kiértékelés esetén gyakran megnövekszik a megoldások lemásolása más diákról, akinek már a rendszer jó értékelést adott.
- A hatékony felhasználás érdekében el kell készíteni **a rendszer használati útmutatóját**.
- **Rosszindulatú programokra fel kell készülni**, melyek ha nem is a szó szoros értelmében akarnak kárt okozni a rendszerben, de hibás működésükkel mégis azt teszik. Erre a legjobb módszer a futási privilégiumok megfelelő beállítása vagy izoláltkörnyezet megteremtése (pl.„homokozóban” történő futtatással). Fel kell készülni arra is, hogy például egy adott munka a tényleges funkcionalitás helyett csak az előre kiszámítottkimentet nyújtja (csalás).
- **Korlátozott képesség**: A dinamikus típusú vizsgálatoknál leginkább a funkcionális követelményeknek való megfelelést vizsgáljuk különböző teszteken keresztül. Sajnos nem mindig oldható ez meg input-output unit tesztekkel. Eseményvezérelt grafikus felületű alkalmazásoknál például különösen nehéz hozzáférni tesztekkel az alkalmazáshoz. Ugyanakkor statikus kódelemzéssel (pl. feltételek komplexitása, kommentezettség) sokféle követelmény vizsgálható. Általánosságban elmondható, hogy minél többféle technikát alkalmazunka beadott munka minőségének mérésére, annál jobb visszajelzés küldhető a hiányosságok javítása érdekében.

2.3. Az AKÉP összehasonlítása egy konkrét eszközzel

Az AKÉP -ahogy az a dolgozat későbbi fejezeteiben azonosítható-, a 2.1 fejezetben leírt csoportosítások kategóriáira, funkciók mindegyikére próbál egy egységes keretrendszert teremteni, amiben az oktatónak már "csak" a mechanizmus definiálása (milyen

típusú, megközelítésű legyen a keretrendszer által nyújtott kiértékelés) és a tesztek elkészítése marad (egy definiált szakterületi nyelv segítségével, függetlenül a vizsgált munka programozási nyelvétől).

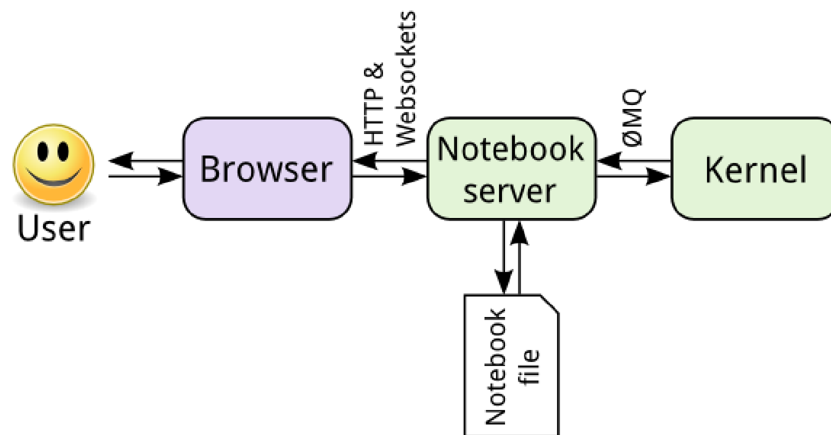
A probléma az, hogy ilyen keretrendszerre nem találtam példát az eddigi kutatásom során. Ugyanakkor lényegesnek érzem, hogy valamilyen értékelő rendszert párhuzamba állítsak vele. A választásom napjaink egyik igen csak feltörekvő,pythonban készültnyílt forráskódúprojektjére, aJupyterre esett.Az érdekesség, hogy ez nem kiértékelésre vagyLMS-neklett kitalálva, de könnyen azzá tehető az nbgradernevezetű csomag telepítésével.

2.3.1. AJupyterNotebook elemzése röviden [4]

A JupyterNotebook egy interaktív számítási környezetet ad, lehetővé téve a felhasználóknak, hogy olyan notebook dokumentumokat készítsenek, melyek tartalmazhatnak: futó kódot, interaktív modulokat (widgets), diagrammokat (plots), formázott szöveget, matematikai egyenleteket, képeket, videókat. Ezek a dokumentumok számos formátumba konvertálhatók, úgy, hogy a futtatott kód és eredménye is része lesz a dokumentumnak. Három komponensből áll. Az első a notebook webes applikáció, mely interaktív lehetőséget biztosít futó kódot tartalmazó dokumentumok készítésére, szerkesztésére, menedzselésére. A második a kernel, mely elválasztja a dokumentumban megjelenített tartalmat és forráskódot a végrehajtástól. A kernel hajtja végre a dokumentum adott kódrészletét és az eredményt beilleszti a dokumentumba. A forráskód típusa dönti el, melyik kernel végzi el ezt a műveletet. A harmadik a notebook dokumentum. Minden dokumentum saját kernellel rendelkezik. Ezek a dokumentumok a fentilistából tartalmazhatnak elemeket tetszőleges mennyiségben és sorrendben (ezek az úgynevezett cellák).
A három komponens elhelyezkedését és kommunikációját illusztrálja a 3. Ábra.

A dokumentumok tárolnak minden inputot és outputot a végrehajtás után, ugyanúgy ahogy a szimpla szövegeket is. Valójában ezek a tartalmak csak a webes applikációban vagy már exportálás utánöltenek HTML, LaTeX, PNG, SVG, PDF stb. alakot, addig egy .ipynb fájlként léteznek, mely JSON formátumban tartalmazza az összes adatot szimpla szöveges, jól olvasható formában (cellánként meghatározva a meta adatokat). A stílust a meta adatok illetve szimpla szöveges reprezentáció esetén a Markdown leíró nyelv határozza meg.

A jupyter kernel modulja teszi lehetővé a notebookokban lévő kód futtathatóságát. Mindenkernel egy adott programozási nyelvben írt forráskód futtatására képes. Számos kernel érhető el, alapértelmezetten pedig egy kernel van telepítéskor a környezetben, ez pedig a python. A felhasználók egyszerű módon határozhatják meg, hogy dokumentumukhoz melyik kernel fusson. A kernelek websocketen keresztül kommunikálnak a klienssel JSON formátumban.



3. ÁbraJupyter Notebook architektúra felszínesen, forrás: [4]

2.3.2. Az Nbgrader elemzése [5]

Az nbgrader egy olyan eszköz, mely megkönnyíti a feladatok készítését és pontozását a Jupyter notebookra épülve. Az oktatók számára lehetővé teszi, hogy egyszerűen készítsenek notebook alapú feladatsorokat, melyek vegyesen tartalmazhatnak kódolási feladatokat és szabad szöveges válaszokat egyaránt. Az nbgrader ezen felül egy az oktatási forgatókönyvnek megfelelő feladat elkészítése, kiadás(release), beszedés(collect), osztályozás(autograde|fromgrade),visszajelzés(feedback) funkcionalitást is nyújt.
Az elemzés során csak az eszköz lehetőségeit és működési elvét ismertetem, a részletek (mint pl. pontos kialakított mappastruktúra) megtalálhatók az[5] forrásban.

2.3.2.1 A feladat készítésétől a kiértékelés utáni visszajelzésig

Az oktatói oldalról legelőször egy kurzust kell indítani, melyhez az nbgrader kialakít egy meghatározott könyvtárstruktúrát. A mappa gyökerében elhelyez egy konfigurációs fájlt, mely leírja melyik feladatsor milyen határidővel rendelkezik, illetve mely felhasználók tartoznak a kurzushoz. A kialakított könyvtárstruktúrában hozhat létre az oktató feladattípusokat és azon belül feladatsorokat. Amikor egy feladatsort elfogad az oktató, akkor az abban lévő megoldásokat kiveszi az nbgrader és a helyére tesz egy hiba

dobást (mind a megoldást jelölő komment, mind a behelyezésre kerülő tartalom konfigurálható), továbbá opcionálisan a notebook tetejére oda tud fűzni egy másik notebookot(ha például több feladatsor eleje közös). A kiadás parancs hatására az nbgrader az elfogadott feladatsorokat a konfiguráció alapján átmásolja egy megadott könyvtárba (megadott könyvtár/feladat típus/feladatsor.ipynb). Ez a könyvtár kerül a kurzus tagjaival megosztásra. Az adott tanuló a jupyter notebook felületen kéri le a kiadott feladatsorokat. Az adott feladatsort kiválasztva letöltheti a saját tárjába és elkezdheti a megoldását. Amikor tesztelni szeretné a munkáját, ezt megteheti a felületen. Amikor úgy érzi végzett, akkor a munkáját megjelölheti elvégzettként, így az nbgrader átmásolja azt a beadott munkák közé, ami szintén egy megosztott mappa. Ezt a műveletet a beadási határidőig többször is megteheti. Amikor az oktató begyűjti a beadott munkákat, akkor azok átmásolódnak a saját kontextusába.Ezt követően automatikusan kiértékelheti ezeket, illetve a manuális kiértékelést megkönnyítő form-rendszerben írhat megjegyzéseket, pontozást egyszerű módon mindegyikhez. A kiértékelés végeztével a lepontozott és megjegyzésekkel ellátottmunkát az oktató visszaküldheti a tanulóknak.

2.3.2.2 A feladatsor készítő eszköz tulajdonságai és lehetőségei

Az oktató a feladatsor elkészítésekoraz nbgraderjupyter felületébe integrált funkcióit használhatja. A nem kód típusú cellákra a következők közül választhat: "semmi", "manuális pontozású" vagy "csak olvasható". Minden kód típusú cellaaz előző három opción kívüllehet"automatikus osztályozású", „automatikus osztályozáshoz teszt”típusú is. A"semmi" típusú cellákon kívül a többinek rendelkeznie kell egy ID-val. A "manuális pontozású" cellák az oktató kézi tesztelését várják el a beadott munkánál. Az "automatikus osztályozású" cellákra az „automatikus osztályozáshoz teszt” típusú cellákban leírt unit tesztek futhatnak le. Az utóbbinál lehet a pontszámot is meghatározni. Az „automatikus osztályozáshoz teszt” cellák mindig csak olvashatóak, ami itt azt jelenti, hogy ha a diák át is írja, akkor is az eredeti tartalom fog visszakerülni bele.

Jelenleg sajnos a jupyter notebook szisztematikájából adódóan nincs lehetőség a tanulók elől elrejteni a teszteseteket. Ez számos esetben nagy hátrány, hiszen a diákok többsége próbál ügyeskedni, „olcsóbban” megoldani a feladatokat. Ha látják a teszteseteket, akkor nem arra fognak fókuszálni, hogy hogyan oldják meg a feladatot, hanem hogy hogyan oldják meg a tesztet és a kettő közel sem azonos. Az érvek között felmerül, hogy amenynyiben tudják, hogy automatikus tesztek futnak, akkor is ügyeskedni próbálnak, azonban

közel sem biztos, hogy ilyen döntést hoznak, ha nem látják a teszteseteket (hiszen a tesztesetek elrejtése megnehezítené az ügyeskedésüket).

2.3.3. Összehasonlítás

Az AKÉP-et és az nbgrader-t a 2.1 fejezetben leírt csoportosítások és jellemzők alapján hasonlítom össze. Mielőtt ennek leírása következne kiemelek néhány fontos információt: az AKÉP nem LMS és nem is akar az lenni, ugyanakkor LMS-be integrálható és erre látható is példa a dolgozat3.4 fejezetében. Mivel nem LMS, így bizonyos funkciók nem tartoznak a hatáskörébe és ez az összehasonlításnál islátszódni fog.
A4. Táblázata két eszköz összehasonlítását mutatja funkcionalitás, verifikációs lehetőségek, csatlakozási interfész alapján.

Eszköz	F1	F2	F3	F4	F5	F6	F7	V1	V2	V3	V4	V5	V6	V7	V8	I1	I2	I3	I4	I5
AKÉP		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x		x
Nbgrader	x	x	x	x		x		x	x		x	x	x	x		x		x		x

4. Táblázat: AKÉP és nbgrader összehasonlítása jellemzők alpján

A programozási nyelvek támogatása szempontjából mindkét rendszer nyelv- illetve technológia-független. Az nbgrader a jupyter kernel segítségével, az AKÉP pedig a preprocesszorokkal[3.1.5.2] oldja meg ezt.

Az AKÉP szemszögéből összehasonlítás eredménye

F1-nél (elektronikus beadás) azért nincs x, mert a feladatokat nem az AKÉP-nek adják be a tanulók, hanem az LMS-nek. Az LMS-ből pedig a trigger továbbítja azt az AKÉP-nek. Az F5-höz (kérdések gyűjteménye) a referenciarendszerét [3.1.3] használja fel. F6 (beadások visszamenőleges tárolása) esetében - az LMS-től függetlenül - az AKÉP rendszere eltárolja a korábbi beadásokat, az azok során történt kiértékelési eljárások minden információjával együtt. F7-nél (statisztikai riport készítése) az analizáló modul kap szerepet [3.1.14]. V3-nál (tesztek megfelelősége) fontos megjegyeznem, hogy mivel egy keretrendszerről van szó, a lehetőségeket vizsgálom, így ebben az esetben is, amennyiben a csatornákat úgy konfigurálják [3.1.5.1] minden további nélkül előállítható egy ilyen verifikációs lépés, ugyanez igaz a V8-ra (eredetiség vizsgálata) is.

Nbgrader szemszögéből összehasonlítás eredménye

F5 (kérdések gyűjteménye) nem támogatott, egyedül a feladatsort képező notebook elejét lehet egy másik notebookból betenni, ami a feladatsor közzétételekor kerül bele. F7-re

(statisztikai riport készítése) nincs konkrét támogatás, ugyanakkor az eredmények adatbázisba kerülnek be, így onnan manuálisan előállíthatók riportok. V3-ra (tesztek megfelelősége) nincs lehetőség ebben az eszközben, mert a korábban írt „automatikus osztályozáshoz teszt" típusú cella az, ami tesztelhet, ugyanakkor ezmindig megőrzi az oktató által beírt tartalmat, tehát ha a diák ide írna valamit, az nem futna le. V8-ra (eredetiség vizsgálata) szintén nincs direkt támogatás, ugyanakkor mivel megőrzi a beadott munkákat ezt lehet egy külső eszközzel vizsgálni.

Összehasonlítás osztályozási sémák alapján

Eszközök	Értékelés típusa	Megközelítés	Specialitás
AKÉP	*	*	Versenystílusú, *
Nbgrader	Fél automatikus	Diák központú	Kvíz stílusú

5. Táblázat: AKÉP és nbgrader összehasonlítása osztályozási sémák alapján

Az AKÉP-nél azért szerepel minden oszlopban csillag, mert a keretrendszer nem köti meg, hogy miként használjuk fel. Mindegyik dimenzió esetében támogatja a 2.1.3 fejezetben leírt kategóriákat. Az értékelés típusa csak a feladatleírótól függ, minden olyat, amire készített tesztet az oktató a keretrendszer kiértékel. A megközelítés tulajdonképp két tényező alapján dől el a keretrendszer szemszögéből: mit filterezünk ki a kiértékeléssel kibővített feladatleíróból és mi triggerelheti a kiértékelést. Specialitás szempontjából alapvetően versenystílusúnak lehet mondani, ugyanakkor a csatornák rendszeréből adódóan annyi specialitás tehető hozzá amennyit az oktató definiál a feladatleíróban.

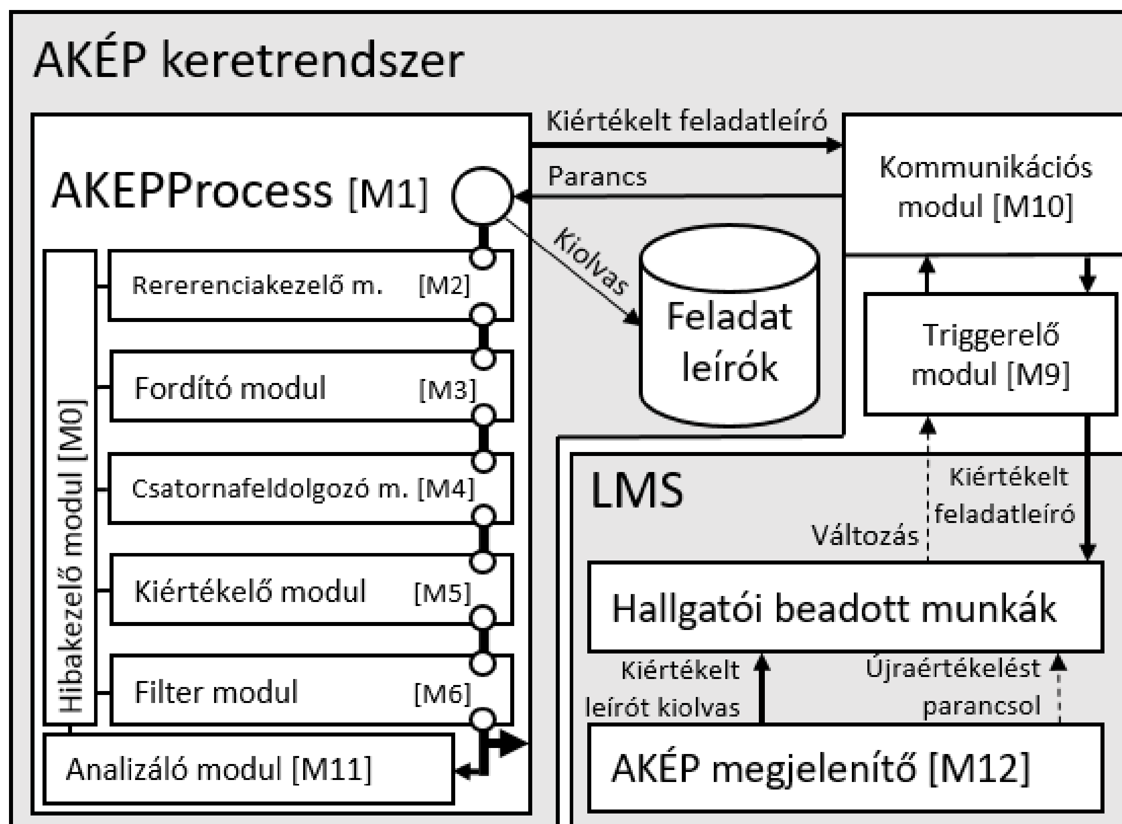
Általánosságban elmondható, hogy az nbgrader előnye, hogy egy olyan projekten ül, ami várhatóan nagyon közkedveltté fog válni a jövőben és nagyon sok újítás is a részévé válik, majd ezen a téren. Fő hátránya hogy keretek közé van szorítva, mint pl. a tesztek elrejthetősége, amit a jupyter működési elve nem enged.

Elmondható az is, hogy több közös tényező is van az AKÉP és a jupyter között, miközben teljesen más célra lettek tervezve. Ilyen például a feladatleíró és a Notebook file szerepe, vagy a preprocesszorok és a kernel funkciójának azonossága.

3. Az AKÉP részletes ismertetése: tervezés és a tényleges alkalmazás bemutatása

3.1. AKÉP architektúra és működés

Előbb a python nyelven íródott keretrendszer magasszintű architektúráját (4. Ábra), majd a fő elemeit mutatom be. Ezen elemek előbb röviden szerepükkel ismertetve, egy-egy mondatban leírva, majd részletesen kifejtve mutatom be a következőkben.



4. Ábra AKÉP architektúra

3.1.1. Keretrendszer felszíne

A keretrendszer AKEPProcess osztálya [M1] definiálja a folyamatmodellt, meghatározva a modulok végrehajtási sorrendjét. Architektúra szempontjából egy Pipes and Filters (csővezetékek és szűrők) minta illeszthető rá [6], ahol a szűrők az egyes modulok, az adata csővezetékben pedig a hallgató munkája és a hozzá kötődő feladatleíró (exercise.X.xml, ahol az X az egyedi azonosítója a DSL segítségével definiált feladatsornak [3.5]).

A minta előnye, hogy a filterek lecserélhetők és újrahasználhatók, valamint (ami nem a minta része) konfigurációjuk a feladatleíróból származik, mely szintén hordozható, könnyen átírható. További előny a párhuzamosítási lehetőség is, melyet lehetne kezelni itt is, a csatornák szintjén is, azonban 3.3 fejezetben látni fogjuk, miért nem ezen a szinten lett kialakítva.

A minta hátránya a hibakezelés, ugyanis felmerül a kérdés, hogy hiba esetén mi történjen, honnan folytassuk, melyik filtertől vagy azon beül pontosan melyik végrehajtó funkciótól? Ezt a problémát megoldja az egyik legfontosabb modul, a hibakezelő [M0], mely átszövi az egész keretrendszert, meghatározza mi számít kritikus hibának és mi olyannak amely mellett folytatható a végrehajtás.

Egy kis eltérés a mintától, hogy a filterek sorrendje itt nem tetszőlegesen kombinálható, azaz például előbb kell bekövetkezni a tesztek végrehajtásának, mint a pontozásnak. A következő sorrend egy adott lefutásra érvényes (érkezett egy parancs kiértékelésre). A keretrendszer választ egy végrehajtási szálat és:

1. Referenciák feloldásával kiegészíti a feladatleírót [M2][3.1.3]
2. Dinamikusan változó tartalmakat lecseréli a megjelölt helyeken a feladatleíróban (pl. felhasználónév, jelszó, dátum) [M3][3.1.4]
3. Feldolgozza a csatornákat [M4][3.1.5]
   a. csatornák inicializálása [3.1.5.1],
   b. végrehajtása és feltöltése adatokkal [3.1.5.2]
4. Teljes kiértékelést végez [M5][3.1.6Kiértékelő modul]
   a. tesztek futtatása [3.1.6.1]
   b. pontozás [3.1.6.2]
5. Törli azon elemeket a feladatleíróból, melyeknek nem kell kikerülnie az értékelésbe [M6]
6. Kezeli a hibákat [M0] [3.1.7Hibakezelő] (ez kicsit kilóg a sorból, mert folyton jelen van mindegyik modulban)

3.1.2. AKÉP modulok egy-egy mondatban

A két leglényegeseb modul az M4és M5, amelyekre számos más modul építkezik, vagy éppen forrásként szolgál ezenközponti szerepet ellátó elemek részére. Most mindegyik4. Ábra-n lévő modulról következik egy-egy mondat:

- Referenciakezelő [M2]: Segítségével feladatleírók között és azokon belül lehet hivatkozni tetszőleges tartalmat (tesztcsoportot, csatornát, feladat leírását stb.) csökkentve így a redundanciát és növelve az adott tartalom/funkció átláthatóságát, újrahasználhatóságát.
- Fordító [M3]: Kulcs-érték pár alapján működő funkció, mely egy hierarchikus kulcs-érték tároló láncon addig halad, amíg meg nem találta a kulcshoz tartozó értéket. Az érték lehet statikus vagy dinamikus (ezen belül minden pillanatban kiértékelt, vagy egyszer kiértékelt és onnantól kezdve statikus)
- Csatorna feldolgozás [M4]: Olyan felkonfigurált scriptek tetszőleges láncolata, melyek kapcsolatot teremtenek a beadott munkával. Ezek a csatornák összeköthetők, lefutási sorrendjük determinisztikusan beállítható. A futtatásukat követően a csatornák feltöltődnek meghatározott formátumú tartalommal, melyeket a kiértékelési fázisban feladatra bontva lehet vizsgálni tesztekkel.
- Teljes kiértékelés [M5]: A feltöltött csatornákat felhasználva minden, a feladatleíróban található tesztekkel ellátott feladatot végignéz és az adott kiértékelési kifejezésnek (melyek összetett logikai kifejezések is lehetnek) megfelelően eldönti az adott megoldás helyességét és teljes egészében, vagy részeiben több stratégia alapján pontozza azt.
- Eredmény filterezése [M6]: Elemek törlése attól függően, hogy az adatnyelő mit szeretne, például hallgatónak szánt kimenet esetén nem biztos hogy kívánatos a tesztesetek felfedése, továbbá a csatornadefiníciókra nincs szükség a kimenetben. Tetszőlegesen definiálható mi az, amit nem szeretnénk a kimenetben viszontlátni. Ehhez annyit kell tenni, hogy a 3.1.4 fejezetben leírt módon egy beépített kulcs értékét kell meghatározni tömb formában. Pl:
  `"notCopyFromDescrtiption":["script","info","exerciseKeys"]`
- Hibakezelés [M0]: Ez a modul gondoskodik arról, hogy hiba esetén a hibáról megfelelő információ kerüljön a megfelelő helyre. Végzetes vagy nem kezelt hiba ese-

tén a visszaadott kiértékelés üres root elemébe, egy error mezőbe kerül a hibaszöveg (itt csak az adott értékelés lefutása szakítódott meg, nincs kihatással a rendszer működésére), folytatható hiba esetén pedig az adott értékelés mellé kerül be egy error attribútum, melyre akár tesztet is lehet írni.

- Triggerelő [M9][3.1.8]: A keretrendszeren kívül álló híd egy LMS és az AKÉP között, feladata, hogy amint egy munka beadásra került az LMS-ben, továbbítsa azt értékelő parancs formájában az AKÉP interfészére és a választ eljuttassa az LMS-be.
- Kommunikáció [M10][3.1.13]: Meghatározza az interfészt, amin keresztül a fogadás és továbbítás zajlik. Jelenleg TCP socketekkel lett megvalósítva. Amikor egy új parancs érkezik a szabad szálak közül egy kiszolgálja azt, a válasz előállításáig, pedig aktív marad a socket és azon keresztül küldi vissza a keretrendszer a választ.
- Analizálás [M11][3.1.14]: A kiértékelés visszaküldését követően az AKÉP strukturáltan elmenti CSV fájlokba a mérési adatokat a végrehajtásról (csatornák lefutásáról, értékelési állapotokról, pontozási eredményekről, hibákról, a teljes lefutás időbeli tényezőiről, kritikus hiba esetén az utolsó állapot azonosításáról).
- Eredmények megjelenítése [M12][3.1.15]: Szintén a keretrendszer hatáskörén kívül eső, független elem. Feladata, hogy az adatforrás által elmentett, az értékelési eredményekkel bővített feladatleírót megjelenítse. Könnyen integrálható legyen egy LMS-be.

3.1.3. Referenciakezelő

Sok minden kiderül, amikor gyakorlatban egy eddig kézi tesztelésű tárgy átalakul félig automatikusan teszteltté. Az egyik ilyen, hogy amennyiben a feladatsorokból kreált feladatleírók tartalmaznak ismétlődéseket, például csatorna definíciójában, azt jó lenne nem külön leírni és tárolni minden egyes feladatleíróban. Szó volt már arról, hogy a feladatleírók XML nyelvre lettek ültetve, ez a tulajdonság egy olyan lehetőséget biztosít, melyet minden bizonnyal más nyelv esetén csak bonyolultabban lehetne felhasználni.Természetesen az XPATH-ról (vagy általánosan egy adott tartalom könnyű hivatkozásáról) van szó. Az AKÉP-ben kialakított referenciarendszer az XPATH alapú hivatkozás mellett `reference`attribútumban meghatározható feladatleíró azonosítóval, feladatleírók közötti

tartalmak átvételét is biztosítja. Kérdésként merülhet fel, hogy mi van akkor, ha az átvett tartalom is tartalmaz hivatkozást és esetleg az visszamutat a kiindulópontba (kört kialakítva ezzel). Ezennél a modulnál nem jelent problémát, amint ilyet észlel kritikus hibaként jelzi, hogy a feladatleíró elkészítése nem sikerülkörkörös hivatkozásból adódóan. A tartalon átmásolásának két módja lehet vagy valamilyen helyőrzőt vált fel az adott tartalom, vagy a referenciával ellátott elem belsejébe kerül. Az 1. Kódpélda az előbbit mutatja be. A helyőrző funkciót a *refPlaceholder* jelöli, míg a *reference* és *refChildrenFind* meghatározza mely feladatleíróban milyen XPATH-nek megfelelő elemek kerüljenek át.

```
<scriptname="" refPlaceholder=""
reference="common" refChildrenFind="./script[@common]"
/>
```

1. Kódpélda Referencia definiálása feladatleíróban

3.1.4. Fordító

A feladatleírókban számos helyen szükséges az adott kulccsal megjelölt tartalom valamilyen értékre történő cseréje. Leggyakrabban egy halmazból dinamikusan választott belépési adatok megadására használjuk csatornadefiníciókban. A kialakított DSL-ben lényegében a változók szerepét töltik be a dinamikus vagy statikus tartalmú kulcs-érték párok. A kulcshoz visszaadott érték az alábbi hierarchia alapján kerül ki. Ha az adott szinten nem található a kulcs, akkor halad tovább lefele a keresés:

1. Kapott parancsban → Futási szint, ThreadSpecific Storage [7]
2. Feladatleíróban lett definiálva → Feladatleíró szint (lokális szint) (exercise.X.xml)
3. Adott környezetben elindított AKÉP konfigurációjában definiált → (fél)globális szint (akep.local.cfg)
4. Minden AKÉP futáshoz egyaránt érvényes konfiguráció → globális szint (akep.cfg)
5. Végzetes hiba küldése, mert nem tudja előállítani a megfelelő feladatleírót hiányzó érték esetén.

Az érték akkor számít statikusnak, ha nem dinamikus, azaz nem beépített funkcionalitás eredménye (pl. felhasználó-jelszó páros választása listából úgy, hogy a választást a keretrendszer végzi megadott szerepkör alapján, figyelembe véve a párhuzamos szálak közötti egyedi választást), illetve az eleje nincs @ jellel ellátva. A @ jellel ellátott érték esetén a keretrendszer megpróbálja @ jel utáni részt kiértékelni a pythoneval függvényével, majd

az eredményt visszaadni értékként. Ez hasznos lehet például dátumokra épülő teszteknél ahol megadott formában az aktuális napot kell a tesztben valamilyen input részeként előállítani. A 2. Kódpélda egy JSON objektum kulcs-érték sora, mely definiálja a timestamphez tartozó dinamikus típusú értéket. A 3. Kódpélda pedig egy statikus típusú értéket definiál.

```
"timeStamp":"@str(time.time()).replace('.','')"
```

2. KódpéldaTimeStamp fordítása dinamikus értékre JSON objektumban

```
<exerciseKeys><Keykey="SQL_SCRIPT">KONYV</Key></exerciseKeys>
```

3. Kódpélda SQL_SCRIPT értéke meghatározva feladatleíróban

3.1.5. Csatorna feldolgozás

A csatornafeldolgozás modulnak az architektúrája szintén Pipes and Filters és itt a mintasorrendezési feltétele sincs megsértve. A csatornák lefutásának sorrendjét elsődlegesen adefiníciójukban szereplő belépési pont (*entry* attribútum) dönti el, másodsorban pedig a feladatleíróban lévő sorrendjük. Az *entry* értéke és azok jelentése a 6. Táblázatban látható.

entry	Az érték jelentése
pre	A *main* jelölésű csatornák végrehajtása előtt fut le. Ide érdemes olyan csatornákat/scripteket definiálni, melyek a szükséges környezetet állítják elő.
main	Azon csatornák, melyek a középső fázisban futnak le, a *pre* csatornák után. Ilyen jelzéssel érdemes azon csatornákat létrehozni, melyek kimenete az értékelés során használva lesz minden egyes feladatnál.
con	A main és post jelzésű csatornával egy időben futó folyamat. Ide olyan csatornát érdemes definiálni, mely valamilyen szolgáltatást nyújt a *main*-ben futó tesztekhez.
post	A main leállása után indul el az itt definiált csatorna. Itt érdemes olyan tevékenységeket elhelyezni, amik tisztítják a munkakörnyezetet vagy plagizálást vizsgálnak.

6. Táblázat A csatornák entry attribútumának értékei, azok jelentése

3.1.5.1 Csatornák inicializálása

A minta annyiban megsérül, hogy nem csak feltétlenül a következő csatorna bemenete az előző csatorna kimenete. A modul inicializálásakor a csatorna bemenete feltöltődik. A feltöltődés fajtája többféle lehet, valamint kettes kombináció is lehetséges az

ebben a fejezetben lejjebb leírt okokból. A 7. Táblázat megnevezi a lehetséges csatorna bemeneti fajtákat és azok jelentését.

Közvetlenül a csatorna definícióján belül, inputstream elemekben megadott bemenet	Amennyiben nincs megnevezve az `inputType` attribútum alapján más csatornabemenet, ez lesz az egyedüli bemenete a csatornának. Ellenkező esetben párosul az attribútumban megadott bemenettel a fejezetben lejjebb leírt módon és okokból.
`inputType == 'inline'`	Ebben az esetben a keretrendszer összegyűjti a feladatleíróban az egyes feladatokban elhelyezett inputstream elemben lévő tartalmat és elmenti azokat strukturált tartalomban.
`inputType == 'external'`	Hasonló, mint *`inputType == 'inline'`* esetén, de ilyenkor a külső fájlnak xml formátumúnak kell lenni, ami megfelel a channel.xsd sémának.
`inputType=='channelOutput'`	Ebben az esetben a *`fromChannel`* attribútumban meghatározott csatorna azonosítója alapján a megfelelő csatorna kimenete kerül alapértelmezetten ennek a csatornának a bemenetére. Ez felülírható az *`inputTo`* argumentummal, mely a bemenetet a meghatározott érték felülírásával a csatorna argumentumában helyezi el.

7. Táblázat Lehetséges csatorna bemeneti fajták

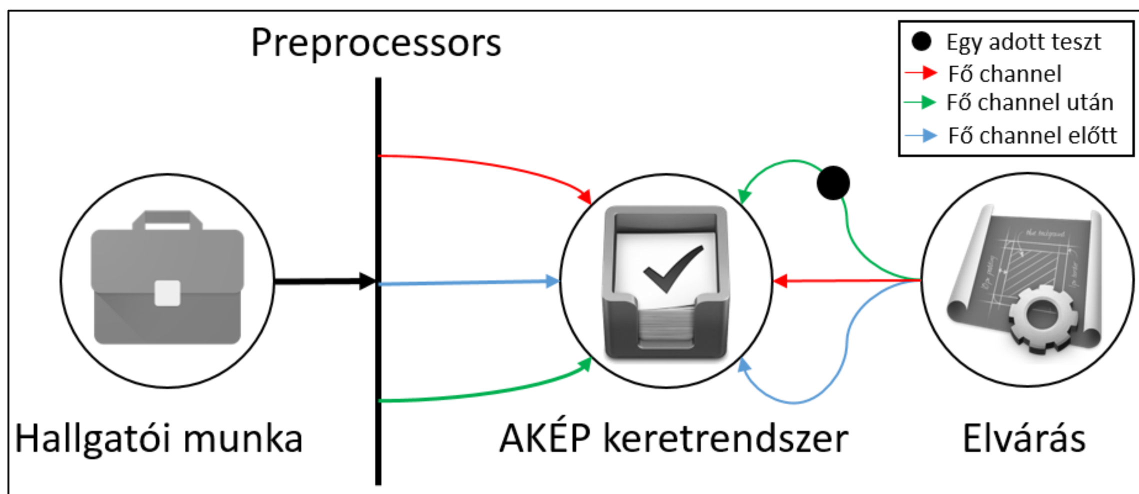
A kettes kombinációjú bemenet, azért hasznos, mert el lehet választani a feladatspecifikus bemenetet az esetlegesen ugyanabban a folyamatban szükséges bemenő adatoktól. Amennyiben egy hallgatói munka váratlanul bezárul és azt a csatornában felkonfigurált script nem kezeli le, a keretrendszer adott körülmények között (*`inputTypeinline`*, vagy *`external`* értékű) újra indítja, és ott folytatja a munkát ahol megszakadt. Ilyenkor felmerülhet a kérdés, hogy mi van azokkal a dolgokkal a folyamat működésében, melyeknek mindig le kell futniuk, például parancs egy adatbázis csatlakozáshoz, melyhez a hallgatói munka várja a felhasználót és jelszót. Erre az esetre tökéletes a kettes kombinációjú bemenet, hiszen hiba esetén a csatornánál közvetlenül definiált input beadásra kerül a hallgató újraindított munkájába, ugyanakkor a feladatfüggő inputok azután folytatódnak, ahol a hiba előtt zárult a végrehajtás.

3.1.5.2 Csatornák végrehajtása és feltöltése adatokkal

A csatornák rendszere az, ami egy technológia független lenyomatot teremt a keretrendszer számára. Pontosabban csak azok a csatornák, melyek közvetlenül megteremtik a kommunikációt a keretrendszer és a hallgató munkája között (ezek általában abban

a nyelvben készülnek, mint maga a munka, amivel történő kommunikációt kezdeményezi az AKÉP), ezek az úgynevezett preprocesszorok, melyek viselkedésüket tekintve proxy mintát valósítanak meg. A preprocesszoroktól a keretrendszer egyetlen egy szabály betartását várja el és emellett további ajánlást tesz a működésükre.
A szabály: készüljenek fel arra, hogy egy strukturált bemenetet kapnak a keretrendszertől (szeparálva a feladatokat) és a keretrendszer ezt viszont elvárja a visszakapott kimenetekre.
Ajánlások: amennyiben a preprocesszor kívánja kezelni a hallgatói munka esetleges váratlan leállását, vagy/és hibáit, úgy a kapott hibát közölje ugyanúgy, szeparált formában, a standard hiba kimeneten és ne nyelje el azokat. A hibák saját kezelése egyébként csak speciális esetben javasolt, nyugodtan rábízható a keretrendszerre.

Az 5. Ábra a fő szereplőket mutatja be a csatornák világában. Minden egyes csatorna egy adott preprocesszorhoz kapcsolódhat, melyek eltakarják a keretrendszer felé a munka technológiáját. A kapcsolódás módját és paramétereit a feladatleírókban kell meghatározni.



5. Ábra Az AKÉP keretrendszere és a preprocesszorok, feladatleírók, hallgatói munka kapcsolata

A csatornák meghatározott lefutási sorrendje alapján a keretrendszer elindítja a csatornákban *path* attribútummal megadott helyen elérhető parancsot/scriptet az *arguments*-ben megadott argumentumokkal és az inicializálásnál meghatározott *inputstream*-mel. A nem *'con'* típusú csatornáknál egy meghatározott időtúllépési értékig vár az elindított csatorna lefutásáig. Az eredményt rögzíti, legyen az hiba vagy helyes kimenet. Amennyiben az *inputType inline*, vagy *external* értékű a definiált csatornánál, úgy különle-

ges bánásmódot élvez a csatorna. Egyrészt amennyiben meghal a folyamat, a keretrendszer próbát tesz annak visszaállítására és onnan folytatja a munkát ahonnan megszakadt, a hibát pedig elmenti és hibásnak jelöli az adott feladatra kapott kimenet fajtáját. Továbbá elvárja a kimenet gondos szeparáltságát, ahogy az a preproesszoroknál elhangzott követelményként. Minden olyan csatorna esetén, amely nem rendelkezik ezzel a beállítással és a folyamathívás visszatérési értéke nem nulla, a keretrendszer kritikusnak veszi a hibát és leállítja az adott munka értékelését.

A csatorna láncon végig érve a keretrendszer leállít minden futó állapotban maradt folyamatot. Jobb esetben ez csak a *'con'* típusú csatornát érinti amennyiben szerepelt ilyen a láncban. Ugyanakkor hiba esetén előfordulhat, hogy nem áll le a csatorna elindításával elindult folyamat, ilyenkor a keretrendszer először megpróbálja a futását felfüggeszteni (*terminate*), majd ha erre sem reagál, akkor kényszerített bezárást parancsol (*kill*).

3.1.6. Kiértékelő modul

A kiértékelő modul a csatornák tartalmára tud teszteket futtatni, és a feladatleírókból valamint az értékelő modul könyvtár közös együttműködéséből születik a technológia független logika a teszteléshez. A csatornák tartalma már strukturáltan ugyan, de csak szöveges adatot tartalmaz (legyen az valamilyen kimenet, forráskód részlet, hiba, …). Az érékelő algoritmusok pedig, melyek köre bővíthető, ezen szöveges adatok tesztelésére adnak mindenféle lehetőséget. A következő két alfejezet részletezi a tesztek futtatásának mechanizmusát és az ehhez szorosan kapcsolódó pontozást.

3.1.6.1 Tesztek leírása és futtatása

A teszteket *'solution'* elemekben lehet leírni. Ezek lehetnek elemi és azokból felépített összetett tesztek. A kiértékelés során egy rekurzív logika bejárja a *'solution'* elemek által felállított kiértékelési kifejezést az adott feladatleíróban. Ezek egy fát alkotnak, melynek a levelei az elemi tesztek, tehát az olyan *'solution'* elemek, melyek rendelkeznek *'evaluateMode'* attribútummal. Az elemi teszteknél az *'evaluateMode'* attribútumban egy értékelő algoritmust implementáló függvény neve található. A levél a kiértékelés függvényének eredménye alapján igaz vagy hamis lesz. A csomópontok pedig bool-algebrai műveletek (negáció, and, or, xor, …), melyek segítségével összetett, de mégis átlátható tesztelés oldható meg. Ez két szempontból is előnyös. Az egyik, hogy az elemi tesztek újrahasználhatók, valamint ha összetettebb eseteket vizsgálunk, akkor az részeiben is értékelhető. A másik nagyon lényeges része, hogy számos alternatívát fel

lehet állítani egy ilyen értékelési fában úgy, hogy ha az egyik ága el is hasal a hallgató munkájára, egy másik ág még bizonyíthatja annak az adott feladatra készített megoldás helyességét.

A helyesség definiálható úgy is egyes esetekben, hogy az elvárt kimenet egy hiba. Ilyenkor jön jól, hogy a csatorna minden esetben tartalmazza a struktúra adott elemére, hogy az hibás kimenetnek lett-e minősítve. Az elemi tesztnél egy *'errorCheck'* attribútummal érhető el, hogy az legyen helyes, hogy hibát dobott a munka adott része, természetesen a teszt igaz kimenetéhez ekkor az is szükséges, hogy a hiba szövege megfeleljen az elvárt tesztnek.

Minden elemi teszt futtatásához kötelezően meg kell határozni, hogy az mely csatornára értelmezett. Ezt a *'channelName'* attribútummal állíthatjuk be. A tesztek bemenete minden esetben a megjelölt csatorna kimenete az adott feladatra (aminek a kontextusában a teszteset leírása van). Amennyiben a csatorna nem szeparált kimenettel rendelkezik, akkor az egész tartalma kerül átadásra.

Az értékelő algoritmusokat tartalmazó könyvtár kiegészítésekor egy interfésznek kell megfelelni, mely a következőt várja el. A függvény, amiben leírásra kerül a logika minden esetben legalább két paramétert kap. Az első a hallgató megoldása az adott csatorna adott feladatára, a második pedig a feladatleíró készítője által elvárt formális követelmény, mely a kiértékelő függvény által megkívánt szintaxist követi. A 4. Kódpélda *and* kapcsolatban álló két elemi tesztet mutat, szerepeltetve a fent leírt attribútumoknak egy részhalmazát.

```
<solutionscore="1" operator="and">
        <solutionevaluateMode="rowNumEq" channelName="Main">
             43
        </solution>
        <solutionevaluateMode="cellData" channelName="Main">
             0,1:0547928211|||0,2:IGEN|||42,1:0486278077|||42,2:NEM
        </solution>
</solution>
```

4. KódpéldaPélda egy teszteset leírására

3.1.6.2 Pontozó rendszer

A pontozó rendszer teljesen illeszkedik a kiértékeléshez, ugyanabban a modulban is szerepel. Ahogy a tesztleírást a modul fa struktúrában dolgozza fel, úgy a pontozás is ezt a mintát követi. Minden *'solution'* elemen meghatározható pontszám, azonban nem

kötelező mindenhol. A rendszer a következő módon működik. A levelek irányából terjednek felfele a pontok. Amennyiben egy csomóponton meg van határozva a pontszám és a csomópontban igaz értékű a boolean algebra, a csomópont pontszáma terjed felfele tovább, egyébként pedig a csomópont gyerekelemeiben elhelyezkedő pontszámok (származzanak csomópontból vagy levélből) eredménye egy meghatározott függvény alapján. Ennek a függvénynek a meghatározása a csomópontban definiált operátortól függ. Amennyiben ez *'and'* úgy összeadódnak a pontszámok, minden más esetben pedig a maximumot választja ki a modul. Miért jó ez? Azért mert így az *'and'* operátorral ellátott csomópontok gyermekeinek pontszámbeli eredménye tovább mehet még akkor is, ha a csomópont értéke hamis (mert nem mindegyik gyerek értéke lett igaz), azaz részpontszámot kap a megoldás.

Korábban pontozási stratégiáról írtam, ez nem csak a fent leírt részpontszám és automatikus pontszámterjesztésre vonatkozik, hanem arra is, hogy különböző pontozási fák építhetők fel. A normál pontozáson kívül lehetőség van bónusz és mínusz pontozást is beiktatni. Ezek ugyanúgy vegyesen szerepelhetnek az előző fejezetben leírt értékelési fában, azonban a *'solution'* elemeknél a pontozási stratégiát meg kell jelölni a *'scoreType'* attribútumban. Természetesen a feladatokra adott összesített pontszám: normál pont – mínusz pont + bónusz pont.

3.1.7. Hibakezelő

A lehetséges hibákat két fő csoportba sorolom, továbbá megjegyezném, hogy a most következő állítások csak a keretrendszer tartományában érvényesek. Az egyik ilyen csoport a felkészült esetek gyűjteménye, ezek egységesen egy saját hibaosztályon keresztül kerülnek rögzítésre. A másik pedig a nem várt hibák esete. A hibák kezelése ezen kívül megkülönböztethető aszerint, hogy az adott kiértékelés folytatható-e mellette vagy sem. A kettő között van annyi összefüggés, hogy minden nem várt hiba egyben az adott értékelés megszakítását is jelenti. Az ilyen kimenetelű kritikus hibák részleteit egy üres root elementet tartalmazó XML error attribútumában rögzíti a keretrendszer, majd elküldi az adatnyelőnek.

Hova kerül a hiba? Erre több irányú megközelítést támogat a keretrendszer. Alapértelmezetten logolásra kerül, továbbá standard kimenetre, valamint az adatnyelőhöz bizonyos esetekben. A hiba mértékének függvényében a python loggerén keresztül számos handler

beállítható, így például definiálható az is, hogy kritikus hibák esetén emailt kapjanak bizonyos személyek.

A 8. Táblázatban felsorok néhány olyan hibát, melyre fel van készítve a keretrendszer és mindegyiknél jelzem, hogy kritikus hibának minősül-e.

Modul	Hibajelenség	Kritikus? Következmény?
Main	Akep.cfg vagy a feladatleírók sémáját meghatározó fájl nem található.	Igen, és ebben az esetben az AKÉP nem is indul el.
Csatorna – inicializáció	Adott csatornánál definiált script elérési útvonala nem elérhető.	Igen, kiértékelés megszakítása.
Csatorna – futtatás	Az `inputType == 'inline'` és a preprocesszor működése leállt.	Nem, error elmentése adott feladathoz, és a preprocesszor újraindítása, valamint folytatás a következő feladattól.
Fordító	A fordítás során nem sikerül egy @ jellel kezdődő érték feloldása `eval` függvénnyel	Nem, a kulcshoz tartozó érték ebben az esetben maga a @-al kezdődő tartalom
Kiértékelő – tesztek futtatása	Nem sikerül az adott értékelő algoritmust elérni valamilyen okból (pl. nem létezik olyan azonosítóval függvény) egy adott teszt során.	Nem, a kiértékelés eredménye hamis és a pont rá 0, valamint `error` attribútumba bekerül az adott értékeléshez, hogy `evaluateError`.
Kiértékelő – pontozás	Nem létező operátor definíció a kiértékelési fa csomópontjában (nem elemi `'solution'` elem).	Igen, a kiértékelés megszakítása. (ennek oka, hogy itt már lefutott tesztekről van szó, így nem nullázhatom a rá vonatkozó pontot, mert azzal félrevezetném a javítót, miközben lehet hogy a teszt igazzal tért vissza).
Referencia-kezelő	Nem létező feladatleíróra/elemre történő hivatkozás, vagy kör kialakítása.	Igen, kiértékelés megszakítása (ebben a fázisban nem dönthető el, hogy a másolandó tartalom hiánya miképp befolyásolja az eredményt, egyszerűbb ilyenkor a feladatleírót javítani)
Kommunikáció	Megszakadt a kapcsolat az adatnyelővel.	Igen, a kiértékelés megszakítása.

8. Táblázat Kezelt hibák egy részhalmazának felsorolása az egyes modulokra

3.1.8. Triggerelő

3.1.9. A AKÉP modulok egy-egy mondatban

A két leglényegeseb modul az M4és M5, amelyekre számos más modul építkezik, vagy éppen forrásként szolgál ezenközponti szerepet ellátó elemek részére. Most mindegyik4. Ábra-n lévő modulról következik egy-egy mondat:

- Referenciakezelő [M2]: Segítségével feladatleírók között és azokon belül lehet hivatkozni tetszőleges tartalmat (tesztcsoportot, csatornát, feladat leírását stb.) csökkentve így a redundanciát és növelve az adott tartalom/funkció átláthatóságát, újrahasználhatóságát.
- Fordító [M3]: Kulcs-érték pár alapján működő funkció, mely egy hierarchikus kulcs-érték tároló láncon addig halad, amíg meg nem találta a kulcshoz tartozó értéket. Az érték lehet statikus vagy dinamikus (ezen belül minden pillanatban kiértékelt, vagy egyszer kiértékelt és onnantól kezdve statikus)
- Csatorna feldolgozás [M4]: Olyan felkonfigurált scriptek tetszőleges láncolata, melyek kapcsolatot teremtenek a beadott munkával. Ezek a csatornák összeköthetők, lefutási sorrendjük determinisztikusan beállítható. A futtatásukat követően a csatornák feltöltődnek meghatározott formátumú tartalommal, melyeket a kiértékelési fázisban feladatra bontva lehet vizsgálni tesztekkel.
- Teljes kiértékelés [M5]: A feltöltött csatornákat felhasználva minden, a feladatleíróban található tesztekkel ellátott feladatot végignéz és az adott kiértékelési kifejezésnek (melyek összetett logikai kifejezések is lehetnek) megfelelően eldönti az adott megoldás helyességét és teljes egészében, vagy részeiben több stratégia alapján pontozza azt.
- Eredmény filterezése [M6]: Elemek törlése attól függően, hogy az adatnyelő mit szeretne, például hallgatónak szánt kimenet esetén nem biztos hogy kívánatos a tesztesetek felfedése, továbbá a csatornadefiníciókra nincs szükség a kimenetben. Tetszőlegesen definiálható mi az, amit nem szeretnénk a kimenetben viszontlátni. Ehhez annyit kell tenni, hogy a 3.1.4 fejezetben leírt módon egy beépített kulcs értékét kell meghatározni tömb formában. Pl:
  ```
  "notCopyFromDescrtiption":["script","info","exerciseKeys"]
  ```
- Hibakezelés [M0]: Ez a modul gondoskodik arról, hogy hiba esetén a hibáról megfelelő információ kerüljön a megfelelő helyre. Végzetes vagy nem kezelt hiba esetén a visszaadott kiértékelés üres root elemébe, egy error mezőbe kerül a hibaszöveg (itt csak az adott értékelés lefutása szakítódott meg, nincs kihatással a rendszer működésére), folytatható hiba esetén pedig az adott értékelés mellé kerül be egy error attribútum, melyre akár tesztet is lehet írni.

- Triggerelő [M9][3.1.8]: A keretrendszeren kívül álló híd egy LMS és az AKÉP között, feladata, hogy amint egy munka beadásra került az LMS-ben, továbbítsa azt értékelő parancs formájában az AKÉP interfészére és a választ eljuttassa az LMS-be.
- Kommunikáció [M10][3.1.13]: Meghatározza az interfészt, amin keresztül a fogadás és továbbítás zajlik. Jelenleg TCP socketekkel lett megvalósítva. Amikor egy új parancs érkezik a szabad szálak közül egy kiszolgálja azt, a válasz előállításáig, pedig aktív marad a socket és azon keresztül küldi vissza a keretrendszer a választ.
- Analizálás [M11][3.1.14]: A kiértékelés visszaküldését követően az AKÉP strukturáltan elmenti CSV fájlokba a mérési adatokat a végrehajtásról (csatornák lefutásáról, értékelési állapotokról, pontozási eredményekről, hibákról, a teljes lefutás időbeli tényezőiről, kritikus hiba esetén az utolsó állapot azonosításáról).
- Eredmények megjelenítése [M12][3.1.15]: Szintén a keretrendszer hatáskörén kívül eső, független elem. Feladata, hogy az adatforrás által elmentett, az értékelési eredményekkel bővített feladatleírót megjelenítse. Könnyen integrálható legyen egy LMS-be.

3.1.10. Referenciakezelő

Sok minden kiderül, amikor gyakorlatban egy eddig kézi tesztelésű tárgy átalakul félig automatikusan teszteltté. Az egyik ilyen, hogy amennyiben a feladatsorokból kreált feladatleírók tartalmaznak ismétlődéseket, például csatorna definíciójában, azt jó lenne nem külön leírni és tárolni minden egyes feladatleíróban. Szó volt már arról, hogy a feladatleírók XML nyelvre lettek ültetve, ez a tulajdonság egy olyan lehetőséget biztosít, melyet minden bizonnyal más nyelv esetén csak bonyolultabban lehetne felhasználni.Természetesen az XPATH-ról (vagy általánosan egy adott tartalom könnyű hivatkozásáról) van szó. Az AKÉP-ben kialakított referenciarendszer az XPATH alapú hivatkozás mellett *reference*attribútumban meghatározható feladatleíró azonosítóval, feladatleírók közötti tartalmak átvételét is biztosítja. Kérdésként merülhet fel, hogy mi van akkor, ha az átvett tartalom is tartalmaz hivatkozást és esetleg az visszamutat a kiindulópontba (kört kialakítva ezzel). Ezennél a modulnál nem jelent problémát, amint ilyet észlel kritikus hibaként jelzi, hogy a feladatleíró elkészítése nem sikerülkörkörös hivatkozásból adódóan. A tartalon átmásolásának két módja lehet vagy valamilyen helyőrzőt vált fel az adott tartalom,

vagy a referenciával ellátott elem belsejébe kerül. Az 1. Kódpélda az előbbit mutatja be. A helyőrző funkciót a *refPlaceholder* jelöli, míg a *reference* és *refChildrenFind* meghatározza mely feladatleíróban milyen XPATH-nek megfelelő elemek kerüljenek át.

```
<scriptname="" refPlaceholder=""
reference="common" refChildrenFind="./script[@common]"
/>
```

1. Kódpélda Referencia definiálása feladatleíróban

3.1.11. Fordító

A feladatleírókban számos helyen szükséges az adott kulccsal megjelölt tartalom valamilyen értékre történő cseréje. Leggyakrabban egy halmazból dinamikusan választott belépési adatok megadására használjuk csatornadefiníciókban. A kialakított DSL-ben lényegében a változók szerepét töltik be a dinamikus vagy statikus tartalmú kulcs-érték párok. A kulcshoz visszaadott érték az alábbi hierarchia alapján kerül ki. Ha az adott szinten nem található a kulcs, akkor halad tovább lefele a keresés:

6. Kapott parancsban → Futási szint, ThreadSpecific Storage
7. Feladatleíróban lett definiálva → Feladatleíró szint (lokális szint) (exercise.X.xml)
8. Adott környezetben elindított AKÉP konfigurációjában definiált → (fél)globális szint (akep.local.cfg)
9. Minden AKÉP futáshoz egyaránt érvényes konfiguráció → globális szint (akep.cfg)
10. Végzetes hiba küldése, mert nem tudja előállítani a megfelelő feladatleírót hiányzó érték esetén.

Az érték akkor számít statikusnak, ha nem dinamikus, azaz nem beépített funkcionalitás eredménye (pl. felhasználó-jelszó páros választása listából úgy, hogy a választást a keretrendszer végzi megadott szerepkör alapján, figyelembe véve a párhuzamos szálak közötti egyedi választást), illetve az eleje nincs @ jellel ellátva. A @ jellel ellátott érték esetén a keretrendszer megpróbálja @ jel utáni részt kiértékelni a pythoneval függvényével, majd az eredményt visszaadni értékként. Ez hasznos lehet például dátumokra épülő teszteknél ahol megadott formában az aktuális napot kell a tesztben valamilyen input részeként előállítani. A 2. Kódpélda egy JSON objektum kulcs-érték sora, mely definiálja a timestamphez tartozó dinamikus típusú értéket. A 3. Kódpélda pedig egy statikus típusú értéket definiál.

```
"timeStamp":"@str(time.time()).replace('.','')"
```

2. KódpéldaTimeStamp fordítása dinamikus értékre JSON objektumban

```
<exerciseKeys><Keykey="SQL_SCRIPT">KONYV</Key></exerciseKeys>
```

3. Kódpélda SQL_SCRIPT értéke meghatározva feladatleíróban

3.1.12. Csatorna feldolgozás

A csatornafeldolgozás modulnak az architektúrája szintén Pipes and Filters és itt a mintasorrendezési feltétele sincs megsértve. A csatornák lefutásának sorrendjét elsődlegesen adefiníciójukban szereplő belépési pont (*entry* attribútum) dönti el, másodsorban pedig a feladatleíróban lévő sorrendjük. Az *entry* értéke és azok jelentése a 6. Táblázatban látható.

entry	Az érték jelentése
pre	A *main* jelölésű csatornák végrehajtása előtt fut le. Ide érdemes olyan csatornákat/scripteket definiálni, melyek a szükséges környezetet állítják elő.
main	Azon csatornák, melyek a középső fázisban futnak le, a *pre* csatornák után. Ilyen jelzéssel érdemes azon csatornákat létrehozni, melyek kimenete az értékelés során használva lesz minden egyes feladatnál.
con	A main és post jelzésű csatornával egy időben futó folyamat. Ide olyan csatornát érdemes definiálni, mely valamilyen szolgáltatást nyújt a *main*-ben futó tesztekhez.
post	A main leállása után indul el az itt definiált csatorna. Itt érdemes olyan tevékenységeket elhelyezni, amik tisztítják a munkakörnyezetet vagy plagizálást vizsgálnak.

6. Táblázat A csatornák entry attribútumának értékei, azok jelentése

3.1.12.1 Csatornák inicializálása

A minta annyiban megsérül, hogy nem csak feltétlenül a következő csatorna bemenete az előző csatorna kimenete. A modul inicializálásakor a csatorna bemenete feltöltődik. A feltöltődés fajtája többféle lehet, valamint kettes kombináció is lehetséges az ebben a fejezetben lejjebb leírt okokból. A 7. Táblázat megnevezi a lehetséges csatorna bemeneti fajtákat és azok jelentését.

Közvetlenül a csatorna definícióján belül, inputstream elemekben megadott bemenet	Amennyiben nincs megnevezve az *inputType* attribútum alapján más csatornabemenet, ez lesz az egyedüli bemenete a csatornának. Ellenkező esetben párosul az attribútumban megadott bemenettel a fejezetben lejjebb leírt módon és okokból.

inputType == ’inline’	Ebben az esetben a keretrendszer összegyűjti a feladatleíróban az egyes feladatokban elhelyezett inputstream elemben lévő tartalmat és elmenti azokat strukturált tartalomban.
inputType == ’external’	Hasonló, mint *inputType == ’inline’* esetén, de ilyenkor a külső fájlnak xml formátumúnak kell lenni, ami megfelel a channel.xsd sémának.
inputType==’channelOutput’	Ebben az esetben a *fromChannel* attribútumban meghatározott csatorna azonosítója alapján a megfelelő csatorna kimenete kerül alapértelmezetten ennek a csatornának a bemenetére. Ez felülírható az *inputTo* argumentummal, mely a bemenetet a meghatározott érték felülírásával a csatorna argumentumában helyezi el.

fejezetnél leírt parancs tartalmazza mely hallgató beadott munkáját kell értékelni, az hol található és melyik feladatleíró tartozik hozzá, valamint további opcionális paramétereket, például milyen felhasználócsoport jogosultsága alapján fussanak a környezetet felállító inicializáló scriptek. Lényegében egy fájlszolgáltalátást és az értékeléshez szükséges metaadatokat nyújt egy elosztott rendszerben, melynek két eleme az LMS és az AKÉP.

A triggerelő elhelyezkedése elosztás szempontjából kétféle is lehet, jelen esetben az AKÉP-el van egy rendszerben. Az LMS-ben elmentett hallgatói feladatokat sshfs segítségével éri el. Elindít egy keresést, melyben megvizsgálja, hogy ki az a hallgató, akinek még nem lett kiértékelve a munkája és már van feltöltött állománya vagy már lett egyszer értékelve, de az kritikus hibából adódóan megszakítódott (pl. hibás feladatleíró) vagy egyszerűen újraértékelésre lett beküldve manuálisan. A keresés eredménye alapján felveszi a kapcsolatot az AKÉP-el és a fent leírt parancsot elküldi az adott találatra. A visszakapott kiértékelést pedig elmenti a munka mellé.

Adódik a kérdés, hogy milyen sorrendben kerüljenek értékelésre a munkák, mert pl. forgólódás is kialakulhat a rendszerben azáltal, hogy folyamatosan kritikus hibával megszakad egy adott állomány értékelése. A megoldás erre, hogy fel kell állítani egy konfigurálható sorrendet. Ez jelenleg úgy néz ki, hogy az újonnan érkező megoldások nagyobb prioritást élveznek, mint azok amik már egyszer lettek értékelve. Továbbá a kritikus hibával leálló beadások forgolódása egy időlimittel is el van látva, amelyen belül nem történik meg az ismételt értékelési kísérlet annak érdekében, hogy a hiba javításáig ne terheljék feleslegesen a rendszert.

3.1.13. Kommunikáció

Ez a modul egy vezető-követők (leader/followers) mintát valósít meg. Ez a minta egy hatékony konkurenciakezelési modellt definiál, ahol minden szál egy adott példányon dolgozik, de megosztott eseményhalmazt kezel [7]. Ennek előnye, hogy teljesítményben jobb, hiszen nincs szükség lockolásra, bufferezésre a szálak között. A minta szerves része az eseménydetektálás, -válogatás és -végrehajtás. Amikor parancs érkezik, a socket szerver a szál pool-ból kiválaszt egy munkaszálat és elkezdi a parancs végrehajtását. Ehhez egy AKÉP folyamatot (AKEPProcess példányt) indít, melyet körbe burkol egy hibakezeléssel. Definiált és váratlan hibák esetén is felszabadít minden erőforrást (itt lényegében az elindított alfolyamatok leállítását kell érteni, illetve ha használt a folyamat valamilyen felhasználó-jelszó párost akkor annak a visszatételét a gyűjteménybe [3.3.1]). A parancsok formáját tekintve, a JSON dekódolhatóságot várja el. A parancs szintaxisával foglalkozik csak a kommunikációs modul, a szemantikáját az AKÉP folyamat hivatott ellenőrizni. Egy példa a lehetséges parancsra az 5. Kódpélda.

```
{"ownerID":"nyikes","exerciseID":"21-sql","runUserGroup":"normal",
"solutionZip":"/home/nyikes/sol/jegyzokonyv-2.zip"}
```

5. Kódpélda Egy lehetséges parancs a kommunikációs modulnak

Amennyiben a keretrendszer leállítását kezdeményezzük (CTRL+C a CLI felületén), úgy a már elindított értékelések befejezését megvárja a rendszer.

3.1.14. Adatgyűjtés és elemzés

A modulok közül a kiértékelő, csatornakezelő és kommunikáció során különböző megfigyelések folynak. A megfigyelések minden paraméterét az adott modul gyűjti és végül az analizáló modul helyezi azt formába és fájlokba. A fájlok szerinti szétválasztás a következő alapján történik: minden eltérő feladatleíró külön mappát kap, melyekben minden kiértékelési parancs hatására egyedi nevű (az adott lefutás timestamp-je + az adott szál azonosítója) szintén új mappába kerül. Tehát például a 29-sql feladatleíróhoz tartozó 1475880710.254384 timestamp-el rendelkező futáshoz, mely a 2. szállal lett futtatva a következő mappastruktúra alakul ki: /29-sql/1475880710254384-2/. Ezekbe a mappákba jelenleg három fájl kerül:

- channel_prop.data: Az adott kiértékelés csatornáinak adatait gyűjti össze, a következő sorrendben: csatorna neve, futás kezdete, futás befejezése, hiba (amennyiben volt).
- evaluate_prop.data: Az adott kiértékelés összes tesztjének adatait gyűjti össze a következő sorrendben: feladat azonosító (amelyen belül a teszt található), teszt azonosító, értékelő algoritmus azonosítója, a tesztre meghatározott pontszám (amennyiben az sikeres volt, egyébként 0), hiba (amennyiben volt).
  A teszt azonosító felépítése: [operator]*.(elemi tesztek közötti index) pl.: and.or.2, mely egy olyan összetett értékelési feltételt fejez ki,amely esetében az adott teszt az and kapcsolaton belüli or kapcsolat második operandusa volt.
- task_prop.data: Adott kiértékelés összes feladatának szükséges adatát gyűjti össze a következő sorrendben: feladat azonosítója, kapott pontszám, elérhető maximális pontszám.

Az AKÉP folyamatokról gyűjtött adatok nem mentődnek különböző mappastruktúrába, a modul egy meghatározott fájlba (assesmentRun.data) menti sor formájában a következő adatokat:

- feladatleíró azonosítója,
- hallgató azonosítója,
- a fenti fájlokat tartalmazó mappa neve,
- futás kezdete, futás vége,
- a folyamat utolsó ismert állapota,
- hiba (amennyiben volt).

A folyamat állapotai: init, Start [ok], References [ok] (Referenciakezelés rendben), Key binding [ok] (Fordítás rendben), Channelsinitialize [ok] (Csatornák inicializálása rendben), Channelsrun [ok] (csatornák futtatása rendben), Evaluateall [ok] (Teljes kiértékelés rendben), Filter [ok] (Filter alkalmazása rendben). Ezek az állapotok az előző fejezetekben bemutatott eseteket fedik le.

Az adatokat így elmenti a modul CSV állományokba, azonban ahhoz, hogy ebből következtetéseket szűrhessünk le különböző szempontok alapján (mint egy üzleti alkalmazás esetén), további feldolgozás szükséges. Erre az egyik lehetőség, hogy ezeknek a fájloknak

a tartalmát beillesztjük adatbázis táblákba (akár napi aggregáció részeként) és megjelenítjük azokat különböző diagramokon (pl. MS Office Excel PivotChart segítségével), ezekből riportot is készíthetünk.

3.1.15. Értékelések megjelenítése

Az értékelések eredmények megjelenítéséhez fejlesztett webes kliens AngularJs-re, Bootstrap-re és a [8] forrásban található könyvtárra épül. A tesztelés utáni kiegészített feladatleíró fájl kell, hogy valamilyen módon eljusson a lefejlesztett megtekintő modulhoz. Ennek módja sokféle lehet, a legegyszerűbb, ha fájl szinten elérheti azt, mert így semmilyen más rendszer nem érintett a fájl és a megjelenítő között. Természetesen ez nem zárja ki, hogy a fájl csak adott jogosultságokkal legyen elérhető a szerverről (pl. ha a belépett felhasználó oktatói jogosultságú).

Fontos szempont volt a tervezésnél, hogy a feladatok/tesztesetek közötti navigálhatóság könnyű legyen és tükrözze a feladatleíró fa szerkezetét. A felhasznált forrás részben pont megfelel ezeknek az elvárásoknak. Annyi kiegészítés kellett, hogy egyrészt el kellett készíteni az XML struktúra átalakítását a felhasznált könyvtár „szájízére" (meghatározott struktúrájú JavaScript object), kiegészíteni az AngularJs segítségével elkészített sablonját (pl. hogy a pontok megjelenjenek a fában), másrészt stílus átalakítások (pl. szín eltérések az egyes csatornákhoz, hiba jelzéséhez).

Szerkezetileg három szekcióra bontható. Baloldalon a navigáció a fában (feladat > feladat* > teszt* > teszt). Jobb oldalon az adott elem szerinti tartalom (feladat esetén feladatszöveg, teszt esetén formális, informális elvárás kimenet, eredmény). Felül pedig egy parancsgomb az adott hallgatói munka újraértékeléséhez (ez akkor lehet hasznos, ha egy feladatleíró megváltozott és szeretnénk a változott verzióval kiértékelni a hallgató munkáját). A 6. Ábra bemutatja a Szoftver laboratórium 5 tárgy rendszerébe integrált AKÉP megjelenítőt.

6. Ábra A Szoftver laboratórium 5 tárgy rendszerébe integrált AKÉP megjelenítő

3.2. Értékelési mechanizmusok és bővíthetőségük

Első körben ismertetem a jelenleg elérhető értékelő algoritmusokat az AKÉP-hez, majd részletezem azok működését, végül pedig leírom, hogyan lehet bővíteni ezt a halmazt újabb elemekkel.

Minden értékelő algoritmus közös interfész alapján működik. Az interfész megszabja, hogy az értékelő függvényeknek három paramétere lehet és igaz vagy hamis értékkel térhetnek vissza. A paraméterek sorrendben:

1. adott feladatra kinyert hallgatói kimenet (a kiértékelő algoritmusoknak nem kell azzal foglalkoznia, hogy ez standard error kimenetből lett-e éppen olvasva, vagy a standard kimenetből),
2. elvárt követelmény, valamint
3. egyéb tetszőleges paraméterek kulcs-érték párok formában.

Az első két paraméter string típusú és a kiértékelő modul úgy adja már át, hogy megszabadította minden felesleges whitespace-től. A kiértékelő algoritmusok mindegyike szöveges inputon dolgozik (első paraméter). A követelmények (második paraméter) az adott teszt elem belső tartalmából származnak és illeszkedniük kell az adott értékelő függvény által meghatározott szintakszisra. A 9. Táblázat összefoglalja a jelenlegi értékelő algoritmusokat és röviden leírja melyik mire lett kitalálva.

Értékelő függvény	Jelentése[1]
`containAnd`	A bemenet tartalmaz-e minden, a követelményben pontosvesszővel elválasztott tartalmat.
`containOr`	A bemenet tartalmaz-e a követelményben pontosvesszővel elválasztott tartalmat.
`regexpToInput`	A bemenet megfelel-e a követelményben megfogalmazott reguláris kifejezésnek
`ColumnsEqualParam`	A bemenet CSV formátumú és a fejlécben leírt oszlopnevek halmaza lefedi-e a követelményben vesszővel elválasztott oszlopok reguláris kifejezéssel is meghatározható halmazát
`rowNumEq`, `rowNumGrEq`, `rowNumLtEq`	A bemenet CSV formátumú és a fejléc kivételével a sorok száma a követelményben megfogalmazott számmal egyenlő/nagyobb vagy egyenlő/kisebb vagy egyenlő viszonyban áll.
`cellData`	A bemenet CSV formátumú. A függvény egy táblát készít belőle, melyben többféle módszerrel szűrhetünk (nem adjuk meg a sor indexét, oszlop azonosítóját; egész sorra írunk reguláris kifejezést; csak a sor indexét nem adjuk meg…). Ha a szűrés tartalmaz legalább egy sikeres találatot a visszatérés igaz lesz.

9. Táblázat Jelenleg elérhető értékelő függvények funkciója

3.2.1. Értékelő függvények működése

Az értékelő függvények csaknem minden esetben alkalmaznak reguláris kifejezéseket, így a legtöbbjük kapcsolatban áll a *regexpToInput* függvénnyel.A közös interfész arra is jó, hogy a megkapott paraméterek értelmében egy értékelő függvényben a részproblémákat más (az interfészre épülő), már meglévő függvénnyel lehessen eldönteni. Mivel szöveges inputon dolgoznak, technológia függetlenül lehet használni a bennük lévő logikát. A tesztírónak semmilyen ismerettel nem kell rendelkeznie az adott munkával, preprocesszorral kapcsolatban. Az informális követelményt kell megfogalmaznia formális követelményként az értékélő függvények segítségével. Konfigurációjukra nem csak a formális követelményt és az adott feladat kimenetét használhatják, hanem extra parancsokat kaphatnak a harmadik paraméterben. Eddig nem esett szó arról, hogy a harmadik paraméter honnan veszi az értékét, ehhez nem kell messze tekinteni, hiszen ugyanúgy ahogy a második paraméter is, az adott *'solution'* elem definíciója tartalmazza (az *'evaluateArgs'* attribútumban). Mindegyik függvény egyedi szintaxissal, formális nyelvvel

[1] Az értékelő algoritmusok leírásában a bemenet az első paraméter a függvény fejlécében, tehát a hallgatói kimenet az adott feladatban, melyben a teszt van.

rendelkezik a követelmény meghatározásban (milyen elválasztó karakterekkel határozható meg a séma, mik az egyes elemek jelentése az elválasztások között), azonban mindegyikben közös hogy fel lett készítve a többsoros eredmények feldolgozására. Ennek érdekében a részproblémák eldöntésénél a regexpToInput függvény például reguláris kifejezésekben minden whitespace karaktert lecserél a \s+ mintára így kibővítve annak keresési egyezőségét, továbbá ellátja egy flag-gel, mely engedélyezi a pont karakter új sorra való illeszkedését is. A következő felsorolás leírja az értékelő függvények követelményben elvárt szintaxisát:

- `containXX`: <reguláris kifejezés 1>;…;<reguláris kifejezés N>
- `regexpToInput`: <reguláris kifejezés>
- `ColumnsEqualParam`: <reguláris kifejezés 1>,…,<reguláris kifejezés 2>
- `rowNumXX`: Természetes szám
- `cellData` (1):<cella kifejezés 1>|||…|||<cella kifejezés N>

  cella kifejezés= ([0-9]+|*),([0-9]+|*):<cella tartalmára reguláris kifejezés>
- `cellData` (2): ([0-9]+|*),*::<egész sorra illeszkedő reguláris kifejezés 1>|||…|||::<egész sorra illeszkedő reguláris kifejezés N>

A *`cellData`* jó példa arra, amikor már összetettebb követelményt kell leírni valamilyen formában. Jelen esetben lehetőség van ezzel a függvénnyel két módon vizsgálni a CSV-ből felépített táblázatot. Az egyik esetben (1) konkrétan (sor és oszlop indexel meghatározva) írható feltétel egy adott cellára. Ugyanebben az esetben megtehető, hogy nem határozza meg a teszt írója a sor és/vagy oszlop indexét (*-al jelöli), így csak azt kéri, hogy valamelyik sorban/oszlopban szerepeljen az adott reguláris kifejezésre illeszkedő érték, vagy csak létezzen a táblában egy olyan cella, amire illeszkedik. A másik esetben (2) nyíltan írható feltétel egy adott sorra, ilyenkor vagy meghatározzuk a sor indexét, vagy nem. Az elválasztó karakterrel megadott több követelmény mindegyikének kell teljesülnie, hogy igaz legyen a visszatérési érték ennél a kiértékelő függvénynél.

3.2.2. Új kiértékelő függvény bevezetése

Amennyiben új függvényt szeretne implementálni, az Olvasó ezt a pythonban készült evaluateFunctions.py fájlban teheti meg, a következő követelményeket és ajánlásokat figyelembe véve.

Követelmény: A kiértékelő függvény fejlécének meg kell felelnie a szülő fejezetben leírt interfésznek (három paraméter, a paraméterek szerepének ismerete, visszatérési érték bool típusú).

Ajánlás: Nem kötelező, de a keretrendszer működése és használata szempontjából hasznos, ha a függvény fejlécénél kommentelve van a követelmény paraméterének formális szintaxisa, továbbá ha a függvény működése röviden le van írva. Szintén ajánlások közé tartozik, hogy a kiértékelő függvényben ne kezeljük le a hibát, hagyjuk azt a keretrendszerre, vagy ha elkapjuk, akkor is dobjuk tovább. A harmadik paraméterben kapható kulcs-érték párokat ne kezeljük kötelező paraméterként, mert nem az a szerepe. A harmadik paraméter a feladatleíró tesztjeit író személynek szolgáltat extra lehetőségeket egy adott kiértékelési függvény esetében, ezért opcionális a használata. Például a `regexpTo-Input` függvény esetében ebben a paraméterben megadható, hogy mely karaktersorozatok ne számítsanak a feladatból származó kimenetből az összehasonlítás során (`'skipchar'`). A harmadik paraméterben a kulcs-érték megadásban az érték valójában tömb, tehát ezt ennek megfelelően kell kezelni. Miután elkészült az új kiértékelő függvény a keretrendszert újra kell indítani, és máris használható a feladatleírókban a nevével hivatkozva.

3.3. Modulárisan lecserélhető, bővíthető architektúra

A 3.1 fejezetben részletekbe menően ismertettem a keretrendszer vázát alkotó modulokat, azok szerepét és működését. Ebben a fejezetben a modulok kicserélhetőségére, bővíthetőségére fokuszálok, ebből adódóan itt már implementáció-specifikus részeket is bemutatok.

A Pipes and Filters architektúrális mintának a 3.1.1-ben felsorolt előnyei mellett a bővíthetőségnek is ugyanúgy helye van. Viszont mielőtt rátérek a részletekre, tisztázom a szereplőket. A modulok (a mintában: filterek), nem csak a feladatleíróból szerzik az adatot és a konfigurációt a működésükhöz, hiszen rendelkezésükre áll a csatornán ugyanúgy elérhető kapott kiértékelési parancs, és globális, lokális konfiguráció fájlokban leírt kulcs-érték gyűjtemény is (ugyanakkor ezeket csak olvashatják, nem módosíthatják). Ezeket az adatokat együttesen kezelő osztály a resultContent, melynek példánya mindig bemenő paramétere a moduloknak. A mintának egy módosított esetét alkalmazza a keretrendszer (kiindulva abból, hogy itt valamilyen szinten kötött a filterek sorrendje). A szűrők nem egymásnak adogatják át az adatot (resultContent példányt), hanem a parancs érkezésekor példányosított AKÉPProcess (AKÉPP) teszi ezt meg. Ennek több előnye is van, egyrészt

így a modulok nem csak a resultContent példányt kaphatják meg, hanem a korábban lefutott modulok belső publikus tagjait is (amennyiben rendelkeznek ilyennel), másrészt az új láncszem bekapcsolása is egyszerűbb (csak egy újabb elem, ami megkapja az adott modulok között elhelyezkedve a resultContent példányt).

A valamilyen szintig meghatározott sorrend és a korábbi modulok állapotának olvashatósága, valamint hogy a modulok nem aktív filternek számítanak, sajnos elrontja a minta párhuzamosítási lehetőségeit (a feladatleírókban meghatározott csatornák végrehajtásánál is ugyan ez a helyzet). Ebből adódóan a párhuzamosítás nem ezen (ezeken) a szinten (szinteken) van megoldva, hanem az AKÉPP létrehozásának a szintjén, melyre a 3.1.13-ben tértem ki.

3.3.1. Modulok kicserélhetősége

Az AKÉP architektúra és működés fejezetein végig vett részletes leírás már majdnem elég ahhoz, hogy valaki megpróbálja az adott modult lecserélni a keretrendszerben. Ehhez azonban még szükséges egy-két implementációs pontosítás is. Lévén, hogy vannak modulok, melyek másik modul belső megvalósításától függnek (hogy kinyerjen belső állapotokat az adott modul lefutásából, mert az nem része a `resultContent`-nek), kell egy minimális függőségi áttekintés.

A pythonban nem létezik privát függvény vagy változó, így csak egyfajta jelölése van a fejlesztőknek arra utalva, hogy az adott függvény, változó csak belső használatra lett kitalálva[9] és semmilyen a rendszerben található más modul nem használja azt / nem függ tőle. Ennek a jelölése egy alávonás a változó/függvény neve előtt (pl. `_my_function()`). Ezt a konvenciót követve könnyen láthatja bárki, hogy az adott modul lecserélésekor mire kell különösképp figyelmet fordítania, mert ha a belső működést másképp is oldja meg a többi modulnak a publikus részekkel továbbra is együtt kell tudnia működni.

Tulajdonképpen hat függőség nevezhető meg az AKÉPP és moduljai között. A csatornákat feldolgozó modul (`channel`) és a kiértékelő (`evaluate`) között (1), a channel és az AKÉPP között (egy változó, egy függvény) (2,3), az evaluate és az AKÉPP között két változó (4,5) továbbá a `resultContent` és az AKÉPP között (6). Az (1)-es esetben a `channel.getChannelTaskOutput(…)` függvény az, aminek a referenciáját megkapja az `evaluate` példány és így az elérheti a csatornák tartalmát a kiértékelés során. A (3),(6) kicsit kilóg a sorból, ugyanis itt nem az AKÉPP a fő cél a függőséggel, hanem hogy az

AKÉPProcesst létrehozó kommunikációs modul (*requestHandler*) hiba esetén tudjon takarítani. A (3)-as esetben az elindított csatornák lezárását elvégző függvényt publikálom (hogy hiba esetén a futó folyamatokat leállíthassa a kommunikációs modul). A (6)-os esetben pedig, amennyiben egy adott csatornához a feladatleíró igényelt felhasználó-jelszó párost, de valamiért az AKÉPProcess egy adott állapotban elhasalt, a kommunikációs modul vissza tudja helyezni a kapott párt a használható felhasználó-jelszó páros gyűjteménybe. A (4),(5),(2) mindegyike az analizálást szolgálja. A (4) esetében a *toAnalyze* változó az (5)-nél pedig a *taskAnalyze*. A (2)-nél a *channel.channels* az érintett, ennek is minden csatornának csak az analizáláshoz használt attribútumai (ezen attribútumok elérhetők a schemaSpecificAttr.py-ban).

3.3.2. Modulok bővítése

Mielőtt rátérnék a tárgyra szeretnék egy árnyalatnyi kitérést tenni, mert bár nem a keretrendszer moduljait bővíti, ugyanúgy a keretrendszer lehetőségeit szélesíti minden egyes új preprocesszor, amely azután a feladatleírók csatorna definíciójában használható. Egy új modul létrehozásánál kevés szabálynak kell megfelelni, inkább nevezném nagyrészét ajánlásnak, ahogy azt az Új kiértékelő függvény fejezetnél is tettem.

Az új modult az src/framework/moduls alá érdemes helyezni, hogy az eddigi könyvtárszerkezethez passzoljon. Minden modul, mely a kiértékelési parancs hatására inicializálódik és lefut valamikor a lánc során, az AKÉPP osztály *run()* függvényében fedezhető fel. Az AKÉPP *run* függvénye először készít egy példányt a *resultContent* osztályból majd sorra elindítja a különböző modulokat, melyeket 3.1.1-ben lehet olvasni. Végül visszatér a kibővített feladatleíróval a kommunikációs modulnak. Az elkészített új modul osztályát, tehát a *run()* függvényben úgy kell elhelyezni (inicializálni, majd futtatni), hogy a célja alapján a megfelelő két modul inicializálás/futtatás közé kerüljön. Ennek helye nagy valószínűséggel a csatornák lefuttatása után és az értékelés előtt lesz, mert még az értékelés előtt tehető be plusz dolog, ugyanakkor már rendelkezésre állnak a különböző szemszögekből kinyert hallgatói munka adatok. Ugyanakkor az is elképzelhető, hogy a kiértékelt feladatleíróval van valami egyéb cél még elküldés előtt (pl. valamilyen formában adatbázisba menteni az eredményeket).

Ajánlások:

- Amennyiben valamilyen a szálkezelés szempontjából védendő műveletet akar végezni az új modul (tehát az AKÉPP kontextusán kívül), azt tegye a védett *dataStore*

modullal, mely az AKÉPP-ben *store*-ként érhető el. Ha ott nem található a művelet céljából szükséges függvény, akkor először a modul létrehozója készítse el ott a szükséges függvényt (abban használja a *dataStore*-ban található *Lock* objektumot), majd a moduljának adja át a függvény referenciáját (ahogy ezt például a *channel* inicializálásnál is lehet látni).

- Amennyiben kezel hibákat a modul, úgy a hibaszövegeket az *exception* modulba helyezze és importálja azt a moduljába. Továbbá ha kritikus hibát kíván jelezni a modul, úgy dobjon a leírt modulban szintén megtalálható *AKEPException*-t.
- Az új modul rendelkezzen konstruktorral az inicializáláshoz, és egy *run* függvénynyel a végrehajtás elindításához.

3.4. Integráció LMS eszközbe

Ebben a fejezetben a Szoftver laboratórium 5 (Szlab5) tárgynak a házifeladat beszedő és adminisztrációs portáljába történő integrációt, továbbá a tárgy négy laborjára (Oracle, SQL, JDBC, SOA) kiterjedő teljes AKÉP bevezetését írom le, tehát egy konkrét alkalmazását a tervezett és implementált keretrendszernek.

Mielőtt a részleteket írnám, van pár fontos tényező, információ a tárgy oktatási mechanizmusában és a megvalósított rendszerrel kapcsolatban, ami szükséges a következő fejezetek megértéséhez. A tárgy oktatói gárdája különböző szerepekre oszlik. Vannak mérésguruk, akik az adott labor technológiájának ismerik minden zegzugát. Minden feladatsorhoz tartozik feladatguru, aki minden laboron átvezetően az adott feladatsor tartalmáért, megoldásaiért, problémáiért felelős. Továbbá vannak laborvezetők, akik tartják a számítógépes foglalkozásokat, elindítják a hallgatókat minden labor esetén a feladatlapjuk megoldásának elkezdésében (több szerep is lehet egy szereplőn). Minden hallgató egy adott típusú feladatlapot kap és annak határidőre való elkészítéséért dolgozik. Az ismertetett szerepekre azért volt szükség, hogy leírhassam a preprocesszorok elkészítése mögött kisebb nagyobb szerepben jelen voltam, ugyanúgy ahogy az adott laborra elkészített feladatleírók elkészítésénél is, ugyanakkor az előbbit a mérésguruk az utóbbit a feladatguruk készítették el közös megbeszéléseket követően.

3.4.1. Preprocesszorok a laborokhoz

Az első két laborhoz az SQL nyelvvel kell a hallgatóknak dolgoznia. Így egy olyan alkalmazásra volt szükség, mely képes inicializálni az adott feladatsorhoz tartozó

táblákat, végrehajtani a hallgató munkáját és visszaadni mindezt a kimenetén feladatonkénti struktúrában. A legfontosabb, hogy a lekérdezés jellegű adatokat képes legyen úgy visszaadni, hogy az oszlopok, sorok adott karakterrel elválasztódnak. Szerencsére az Oracle Database-hez készült egy SQLcl [10] nevű kliens, mely tökéletesen kielégíti a kívánalmakat. A struktúráltságot a hallgatóknak kiadott speciális felépítésű szkeleton eredményezi a kimenetben. Így a keretrendszer, miután inicializálta az adott feladatsor környezetét (erre rendelkezésre áll a hallgatóknak ugyancsak kiadott inicializáló sql fájl), odaadja az SQLcl-nek a hallgató munkáját, majd a kimenetet XML-ként parsolva fát épít belőle és mehet is a kiértékelési folyamat. Az inicializálást (táblák létrehozását, feltöltését) teljesítménybeli okokból egy másik Oracle eszköz végzi (mivel ilyenkor még nincs szükség a CSV kimenetre), ennek neve sqlplus [11].

A harmadik labor Java alapú, így kellett egy olyan preprocesszor, ami képes kommunikálni egy Java alkalmazással - ilyet nyilván nem nehéz készíteni, és nem is itt van a csavar. A hallgatók által készített grafikus felületű vastagkliens tesztelése sok olyan kérdést vetne fel, ami kívül esik a tárgy keretein. A megoldás egy CLI belecsempészése a hallgató MVC-felépítésű projektjébe, amikor a fordítás megkezdődik. A CLI a hallgatóknak kiadott interfész szerint megvalósított kontrollert hívja. Mivel megkötöttük, hogy a grafikus felület is ezen kontroller használatával valósítsa meg a funkcionalitását, a CLI-n keresztül logikailag ugyanazt az eredményt kell, hogy kapjuk, mint amit a grafikus felületen. Így az AKÉP keretrendszere a feladatleíróban megadott parancsokat küldi el a lefordított Java projekt CLI-jének, aminek eredménye kerül a preprocesszor kimeneti csatornájába.

A negyedik labor a SOA címet kapta és a hallgatóknak python-ban megírt REST-es webservice-t kell megvalósítaniuk, illetve egy erre épülő mini webes alkalmazást. Itt jön ki igazán a keretrendszer adta lehetőségek tárháza. Mielőtt a tényleges preprocessor lefutna, inicializálni kell a feladatsor adatbázisos környezetét, el kell indítani a hallgató service-t, majd futtatni a preprocesszort, mely a feladatleíró alapján különböző hívásokat intéz a hallgató service-hez, valamint teszteli a kliensoldali webes felületet is phantomJS emulátor segítségével.

Általánosságban elmondható hogy egy adott technológiával készült munka tesztelésére az AKÉP csak annyira jó, mint amennyire az ahhoz készült preprocesszor.

3.4.2. Feladatleírók a feladatokhoz

Az AKÉP feladatleíró séma nem csak azzal a céllal jött létre (rengeteg opcionális elem van benne), hogy formailag megszabja a tesztek, csatornák felépítését és hogy minden feladatleíró betöltése előtt ellenőrizni lehessen ennek betartását. Ajánlást ad arra nézve, hogy mi az a forma, ami egy jó leírást ad az oktatásban megszokott feladatok számonkéréséhez, amit könnyedén fel lehet használni több célra (pl. részhalmazában a diáknak kiadott munkára, vagy akár az oktatónak/javítónak szánt információkra, mint javítási segéd, tesztek leírása). Ebben a fejezetben részletezem, hogy is kapcsolódott be a feladatleírók világa a Szlab5 tárgy kereteibe. A feladatleírók nyelvezete, szerkezete a 3.5 fejezet alatt olvasható. Itt azt írom le miképp csatlakozik ehhez a szerkezethez a tárgy által eddig használt szerkezet.

A Szlab5 labor keretei között számos feladatlap létezik, melyek véletlenszerűen kerülnek párosításba a hallgatókkal. 2016-ig a feladatok leírása és a javítóknak szánt javítási segédlet különböző fájlokban volt tárolva minden egyes feladatlaphoz. A feladatleíró ezekből a következő szerkezet alapján jött létre (minden feladatsor azonosítón belül minden méréshez):

- Csatornák definiálása: minden olyan szükséges preprocesszor, mely célorientáltan kiveszi a hallgatói munkából, ami lényeges lehet az ellenőrzésekor és ezt átnyújtja a keretrendszernek, ahogyan az azt kéri.[3.1.5.1] (oktatónak/javítónak)
- Egész feladatsorra vonatkozó tartalom, opcionális információ (hallgatónak)
- Feladat (ebből annyi, amennyit és ahogyan (= akár feladatok csoportosítva) a feladatlap is tartalmaz), a következő elemekkel:
    - Feladatcsoporton belül kiadott opcionális információ (hallgatónak)
    - Feladat szövege (hallgatónak): Ide kerül a korábban említett egyik fájlból a tartalom.
    - Megoldás informális leírása (javítónak/oktatónak): Idekerül a másik korábban említett fájl tartalma.
    - Megoldás formális leírása tesztekben megfogalmazott követelményekkel (javítónak/oktatónak): Ez az, amit az informális leírás alapján meg kell írni az értékelő algoritmusokat felhasználva [3.2.1][3.1.6.1]

- (csak már a lefutatott feladatleíró esetén) Az adott feladatra vonatkozó hallgatói munka minden definiált csatornán (oktatónak/javítónak)

Azzal, hogy ezek a tartalmak az adott mérés adott feladatsorához kapcsolódóan együtt láthatók (az oktatói látja a hallgatónak szánt tartalmat is, de a hallgató csak a hallgatóval jelzettet kapja meg Szlab5 esetében) a tesztek ellenőrzése, bemenete, kimenete, manuális ellenőrzés mellett mindig ott van jól láthatóan a számonkérés is. Egyszerűbbé és átláthatóbbá téve az oktatóknak ehhez kötődő munkáját, arról nem is beszélve, hogy így egységes képen látható minden, egy adott kiértékelt feladatleíró megtekintésekor az AKÉP megjelenítőben.

2016-tól követően a fent leírt módon elkészített feladatleírók a tárgyhoz kialakított privát GitHub repositoryba kerültek, hogy ezáltal a csoportos munkát (oktatói gárda) és a követhetőséget biztosítani lehessen. Ennek elkészítése és a rá épülő CI (ContinousIntegration) rendszer kialakítása Marton József munkája és ebben a dolgozatban nem is kerül részletezésre, ugyanakkor mindenképp láthatjuk ennek fontosságát (feladatleírók megadott formában való elhelyezése az AKÉP-nek, tesztek, feladatleírók változása esetén azok újra töltése az AKÉP-ben, …).

3.4.3. Hallgatói feladatok eljuttatása a keretrendszernek

Ahogy azt a 3.1.8 fejezetben részleteiben is olvasni lehetett, a triggerelő az, ami fájlszolgáltatásként összekapcsolja az elosztott rendszert, melyben az egyik rendszer végzi a kiértékelést (AKÉP), a másik rendszer pedig az LMS-nek van fent tartva. A Szlab5 LMS-e a hallgatói megoldásokat a hallgatók azonosítójával fémjelzett mappában gyűjti, mindig csak az utolsó beadást megőrizve. A hallgatók a munkájukat zip állományban töltik fel. A triggerelő, amint érzékeli a megadott helyen a változást (az nem csak feltöltésből származhat, hanem a 3.4.4 fejezetben leírt manuális újraértékelés indítástól is) és olyan munkát talál, amelyre nincs még kiértékelés (nincs ott mellette a meghatározott elnevezéssel ellátott kiértékeléssel kibővített feladatleíró xml fájl), akkor azt a fájlt átmásolja az AKÉP környezetébe. Ezt követően parancsot küld a keretrendszernek, melyben átadja a szükséges paramétereket (5. Kódpélda). A keretrendszer innentől kezdve átveszi a stafétát (kicsomagolja a munkát, megkeresi benne az adott mérésre specifikus fájlokat, …) végül visszaadja a kiértékeléssel kibővített feladatleírót a triggernek a megnyitott socketen keresztül. A triggerelő ezt követően az adatot elmenti fájlba meghatározott névvel abba a mappába, amelyikből kiszedte a hallgatói munkát.

3.4.4. AKÉP Megjelenítő integrálása a hallgatói munkák értékeléséhez

A beadó és adminisztratív rendszer még PHP 4-es verzióban készült és már készül az új verziója, emiatt a lehető legkevesebb változtatást akartuk belevinni. Az ehhez szükséges információk, hogy a rendszer által megjelentett értékelő oldalt, milyen módon állítja elő a szerver oldal, a beadott hallgatói munkák hogyan érhetők el. Elmondható, hogy bármely LMS-hez történő integrációhoz lényegében erre a két információra van szükség a tervezett keretrendszer mellett.

A rendszer úgy működik, hogy az adatbázissal történt bármilyen kommunikációt követően a szerver oldal a kimenetet xml formájában állítja elő, majd ezt xslt-vel átalakítva végső formában visszaküldi a kliensnek. Kapóra jön, hogy a kiértékeléssel kiegészített feladatleíró is xml alapú. A hallgatói munka a feltöltést követően egy meghatározott mappastruktúrába kerül, itt képbe kerül a 3.4.3. fejezetben leírt fájl alapú szolgáltatás. Ezen információk alapján a következő módosításokra volt szükség, hogy a kiértékelés eljusson a javítóknak az adminisztrációs rendszer meghatározott oldalára és hogy triggerelhető is legyen újraértékelés céljából:

- Az értékelő oldalakat generáló php fájlban egy plusz függvényhívás, mely az adott hallgató beadási könyvtárából kiveszi a megfelelő értékelést (amelyik laborra éppen nézik a beadott munkát), ha ott van.
- Az így kapott XML-ből tartalom generálása az általam létrehozott xslt-ből, mely eljut a kliens oldalra. Ebbe az xslt fájlba lett elhelyezve a megjelenítést végző javaScript könyvtár is, melyről az Értékelések megjelenítése [3.1.15] fejezetben írtam.
- Végül az értékelés újra elkészítése gombnak megfelelő akció felvétele a post utasítások közé. Hatására az adott laborra leadott hallgatói munkára előállított kiértékelt feladatlap egy timestamp-el új nevet kap, így a triggerelő újra fogja értékelni az adott munkát (mert ilyenkor nem találja a meghatározott nevű kiértékelést az adott munkához). A gomb csak akkor kerül újra a megjelenített tartalom mellé, ha az újraértékelés befejeződött.

3.5. A kialakított szakterületi nyelv

A szakterületi nyelv (DSL) programozási, vagy specifikációs nyelv korlátozott kifejezőerővel, egy konkrét szakterület problémáinak és megoldásainak leírásához [12].

Az AKÉP DSL-je xml-re épülő internál, deklaratív nyelv, tehát az xml-t mint általános nyelvet speciális módon használja fel és a végrehajtás helyett a kívánt eredményre fókuszál (a végrehajtás módjával az adott technológiához készült preprocesszorok [3.4.1] foglalkoznak). Ebben a fejezetben a már sokat emlegetett feladatleírókról lesz szó. Számos eleme van az AKÉP DSL-nek és itt csak a főbb elemekre, azok szemantikájára, szerkezetére fókuszál a fejezet, azonban a nyelv sémáját leíró xsd fájl megtalálható a függelékben leírt GitHub-os repositoryban.

Mivel a keretrendszer a feladatleíró nyelvezete alapján működik, azt interpreter módon, utasításonként dolgozza fel, ha szükséges azt megváltoztatni valamilyen okból (például új modullal bővítjük a keretrendszert és ez már új elemeket is megkíván a feladatleírókban), azt ne csak a sémafájlban, hanem a keretrendszer egy erre a célra elhelyezett fájljában (schemaSpecificAttr.py) is tegyük meg, hogy a keretrendszerben absztrakt maradjon a nyelvi elemhez való kapcsolódás.

3.5.1. Előnyök és hátrányok[12]

A szakterületi nyelv legfőbb haszna, hogy a célcsoport (jelen esetben oktatók/javítók) hatékonyan tud benne dolgozni, szavai, segédeszközei illeszkednek a csoport igényeihez, azaz a szakterület absztrakciós szintjén dolgozhatnak és szakterületi fogalmakkal fejezhetik ki a problémát. A szakterületi nyelvek növelik a produktivitást, a megbízhatóságot, karbantarthatóságot és a hordozhatóságot. Szakterületi tudást hordoz magában, elősegíti ennek a tudásnak a megőrzését és újrafelhasználását (pl. a technológia függetlenül definiált tesztek, melyek formálisan várják el a követelményben megfogalmazott informális elvárásokat).

Hátránya a szakterületi nyelvnek, hogy használóit be kell tanítani, ami idővel jár. A nyelv tervezése, megvalósítása, karbantartása szintén sok idő (és pénz). Határainak megfelelő megállapítása nehéz (ha nem elég általános nem lehet megfogalmazni vele teljesen a problémákat, ha meg túl általános akkor nem növeli az absztrakciót és így a produktivitást sem) .

3.5.2. A séma főbb elemei és azok szerkezete, funkciója

A megtervezett szakterületi nyelv három fő elemre épül. Ezek a `script, task, solution`. Ha ezeket párhuzamba hozzuk a keretrendszer moduljaival, akkor világossá válik a fejezetben leírt funkcionalitásuk és szerkezetük. A `script` nyelvi elem és minden

hozzá kötődő lehetőséget a channel [3.1.5] modul dolgozza fel, míg a `task, solution` esetében az evaluate [3.1.6] modul gondoskodik az elemek által meghatározott kiértékelési végrehajtásról. Minden elem esetén igaz, hogy rendelkezhet a referencia rendszerhez [3.1.3] szükséges attribútumokkal, melyekkel vagy közvetlenül rámutat egy elemre (`reference-id`), mely rendelkezik `id`-val vagy filtert ír rá XPATH segítségével (`refChildrenFind`, ezt több elem áthozásakor érdemes használni).

A `script` nyelvi elem segítségével írható le, hogy milyen csatornákat kívánunk létrehozni, ezt számos attribútum megadásával teljes mértékben megszabhatjuk. Egyrészt kötelezően meg kell határoznunk a csatorna nevét (`name`), hogy a tesztek során hivatkozható legyen, másrészt a script/program elérhetőségét (`path`) (természetesen ha a script szerepel a környezeti változóban, akkor elérési útja helyett nevével is elég hivatkozni rá). Harmadrészt a csatorna lefutási sorrend meghatározásához az `entry` attribútumot, melynek lehetséges értékeit felsoroltam a 3.1.5 fejezetben. Minden más paraméter opcionális, ugyanakkor kevés olyan script van, mely elboldogul vagy értelmes funkcionalitást nyújt argumentumok nélkül (`arguments`), vagy input nélkül (`inputstream`). Az `inputstream` az egyedüli olyan elem a script nyelvi elemen belül, mely belekerülhet magába a script elembe és nem attribútunként szerepel. Minden egyes `inputstream` elem új és új sort jelent a script standard bemenetén. Az inputstreamnek nincs attribútum lehetősége, az elem belsejében tartalmazza az értéket. A 10. Táblázat a script opcionális (ugyanakkor teljesen hasznos) további attribútumait és azok funkcióját határozza meg.

Attribútumok	**Magyarázat**
`fromChannel`	Az adott csatorna inputjához hozzáfűzésre kerül a megadott csatorna outputja.
`inputType`	Meghatározható, hogy a preprocesszor csatorna, mely fajtája van a csatornánál. [3.1.5.1]
`inputTo`	Az megjelölt csatorna kimenete, milyen kulcsszó lecserélésével kerüljön az aktuális csatorna argumentumai közé [3.1.5.1]
`noConAfterError`	Megtiltja, hogy hiba esetén a keretrendszer próbálja folytatni a program újraindításával a munkamenetet. (Csak megfelelő `inputType` esetén folytatható a futás.) [3.1.5.1]
`taskErrorHandle`	Amennyiben szerepel ez az argumentum, úgy a keretrendszer tudja, hogy az adott alkalmazás figyeli a hibákat és azokat szeparáltan visszaadja a standard error kimeneten, tehát hiba esetén nem a keretrendszerre bízza a munkamenet visszaállítását.
`continueCondStream`	A con típusú csatornáknál lehet meghatározni, hogy az adott standard error/outputon jelentkező triggerjel hatására haladjon csak tovább a végrehajtási lánc. Hasznos akkor, amikor pl. várnunk kell egy szolgáltatás elindulására.

`taskOutput`	Amennyiben a csatorna kimenetéből minden szükséges módosítás nélkül kiolvasható a szeparált tartalom.

10. Táblázat A script nyelvi elem opcionális paraméterei

A *task* nyelvi elem inkább szerkezeti megkötést határoz meg, mint funkcionalitást. Különlegessége, hogy magában rekurzívan bármennyiszer előfordulhat (csoportostást meghatározva a feladatoknak). Kötelező attribútuma a feladat azonosító, melyet *'n'*-el lehet meghatározni (ez lehet szám, vagy tetszőleges szöveges azonosító). Az azonosító alapján köti össze a keretrendszer a feladatokat a hallgatók megoldásával, valamint a preprocesszor bemeneteket. Ennek megfelelően a feladatok azonosítójának egyedinek kell lennie az egész feladatleíróban. A *task* önmagában a következő nyelvi elemeket tartalmazhatja: *inputstream* (1), *tasktext* (2), *description* (3), *solution* (4). Az (1) az adott feladathoz tartozó preprocesszor bemenetet tartalmazhatja, kötelező attribútuma a *channelName,* mellyel meghatározza melyik csatorna bemenete akar lenni. A (2),(3) a feladat leírása és az ellenőrzések informális megadására szolgál. A (4) pedig a következő bekezdésben kerül részletezésre.

A *solution* nyelvi elem segítségével lehet a kiértékelést leírni. Ez egy összetett nyelvi elem, mert szerkezete logikai kapcsolatot határoz meg és emellett az elemi *solution*-ök követelményeket tartalmaznak formális leírásban. A *solution*, tehát rekurzívan tartalmazhatja önmagát, ugyanakkor ilyenkor a tartalmazó elemben meg lehet határozni a tartalmazott elemek logikai kapcsolatát (*operator*, alapértelmezetten *or* kapcsolat, de lehet még *and/xor*), az eredmény tagadását (*negation*). Azok a *solution* elemek, melyek már nem tartalmaznak magukban további elemet (ezeket nevezzük eleminek), meghatározzák az elemben szövegesen megadott formális követelményre, hogy mely kiértékelő függvénnyel/algoritmussal (*evaluateMode*) fusson a hallgató adott feladatára vonatkozó megoldásra, továbbá hogy az a megoldás mennyit ér (*score*). Ilyenkor kötelező meghatározni, hogy mely csatorna adott feladatra vonatkozó kimenetét vizsgálja (*channelName*). A további, opcionális paramétereket a 11. Táblázat foglalja össze.

scoreType	Értéke lehet bonus, minus. A keretrendszer az adott pontozási típus vizsgálatakor leszűri azokat a teszteket, melyek az adott pontozási kategóriába tartoznak és az alapján építi az értékelési fákat [3.1.6.2]
errorCheck	Amennyiben megadjuk ezt az attribútumot, azt várjuk el, hogy az adott tesztesetnél a hallgató munkája hibára fusson. Ilyenkor a keretrendszer az adott feladatra a megadott csatornánál az error kimenetet vizsgálja (amennyiben taskErrorHandle aktív volt a csatornánál, egyébként a keretrendszer által elkapott hiba olvasható ki ilyenkor)

`evaluateArgs`	Kulcs-érték formában átadható extra paraméter az értékelő függvényeknek. Pl. `evaluateArgs=”skipchar:kihagy1;skipchar:kihagy2;”` Az értéket ilyenkor a keretrendszer tömbként kezeli, ami megadható a fenti példában leírt módon.

11. Táblázat A solution nyelvi elem opcionális paraméterei

4. Eredmények, jövőbeli tervek

Egy olyan tárgyat, mint a Szlab5 nem könnyű átalakítani manuális értékelésről, tesztelésről automatikusra (fél-automatikusra), mert megannyi feladatsor és öt különböző labor képezi a tananyagát. Az AKÉP első bevezetése számos tanulsággal szolgált (pl. hogy mennyire kelljen pontosítani egy feladatszöveget, hogy az már kellőképp specifikálja az elvárást), ugyanakkor már használható módon tudott segítséget nyújtani a javítóknak (pl. SQL méréshez).

Ebben a fejezetben a tanulságokat (1) és az eredményeket fogom elemezni (2), végül kitérek a jövőbeli tervekre. Az (1)-es esetben a 2015/2016 tavaszi félév során előkerült problémák, tapasztalatok egy fő részét osztom meg, hogy aki hasonló szándékkal akar egy tárgyat átalakítani, ezek már a tudásalapját képezhessék. A (2)-es esetben pedig ennek a félévnek az AKÉP eredményeit különböző mérési alapok segítségével [3.1.14] vizualizálom és magyarázom.

4.1. Tanulságok, tapasztalatok

Az oktatói gárda 3.4 szerinti felosztása lehetőséget adott a bevezetéssel járó terhek elosztására. A preprocesszorok és a feladatleíró tesztsablonok előállítására a mérésgurukkal, majd a feladatleírók elkészítése a feladatgurukkal a 3.4.2 fejezetben írtak alapján. Tervben minden rendben volt, ugyanakkor a gyakorlatban kiütköztek olyan problémák, amikre első körben nem voltunk felkészülve. A teszt branch direkt azért lett kialakítva, hogy bármilyen módosítás először oda kerüljön, majd csak ha letesztelte a commit készítője a munkát, akkor kerüljön át a masterbe. Történtek olyan balesetek, amikor úgy került át egy munka, hogy nem lett tesztelve, így például megsértette az XML szintaxisát, vagy az AKÉP séma szintaxisát. Ez a probléma lappangó jellegű, hiszen addig nem okoz hibát a rendszerben, amíg azt az adott feladatlapot le nem generálják (mert a hallgatók papír alapon kapják meg a feladatlapokat ennél a tárgynál, ezzel is gátolva a közös munkájukat és a feladatoklapok terjesztését), vagy az AKÉP egy olyan munkát nem ellenőríz, amihez az adott feladatleíró tartozik. A CI rendszer megoldja ezt a problémát, mert időben jelzi és nem az esemény bekövetkezésekor, vagy csak az után derül ki a probléma.

A másik probléma az idő. Az a tanulság, hogy a feladatleírók tesztjeinek használható szintű elkészítését egy ilyen volumenű tárgynál legalább egy fél évvel előre el kell kezdeni, nem is csak a tesztek jó átgondolása érdekében, hanem a feladatok szövegének kellő

pontosságú kifejezésében. Bármely tárgy automatikus tesztelésűre való átalakításánál az első lépés a feladatok szövegének átvizsgálása és minden olyan eset ahol az elvárás nem egyértelmű, ott tisztázni, pontosítani kell mindent, mert míg egy ember elfogadja a követelmények másképp értelmezését, egy automatizmus ezt csak korlátozott mértékig képes tolerálni (amennyi tesztesetet készítünk a különböző esetekre). Az elmélettel nem volt gond, tiszták voltak a fenti elvek, ugyanakkor a gyakorlatban nem számoltunk azzal, hogy nem minden feladatguru fogja kellő odafigyeléssel végig járni ezeket és amikor ez felfedezésre kerül már nem biztos, hogy rendelkezésre áll elegendő idő a feladatleírók javítására (például azért, mert már ki lett adva a hallgatóknak). A tárgy sajátossága, hogy a „katonasága" legnagyobb számban önkéntes demonstrátorokból áll, ami nem gond és dicséretes, ugyanakkor hátrány is, ugyanis nekik is megvannak az egyetemi elfoglaltságaik, ami mellett nehéz a többletteher (tesztek elkészítése).

A harmadik probléma, hogy a hallgatók nem tartottak be minden elvárt tényezőt (a beadandó könyvtárszerkezettől elkezdve, a szöveges fájl kódolásán át, a sablon tiltott részeinek átírásáig), ami több forrásban is ki lett nekik hírdetve követelmények formájában (pl. hallgatói útmutatóban, weboldalon). Ennek érdekében jobban fel kell hívni a figyelmüket, például előírni, hogy amennyiben nem tartják be, azzal milyen következmények járnak.

4.2. AKÉP Eredmények mérési szempontok alapján

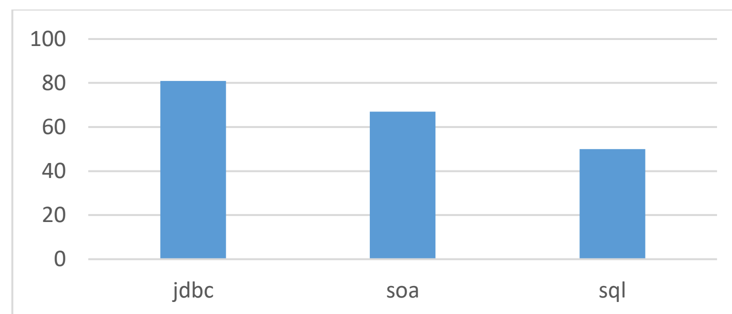
A következő mérési eredmények 2203 feladat értékelését követően alakultak ki. Az eredmények diagramként ábrázolásához az Excel kimutatásdiagramját (Pivotchart) alkalmaztam. Összesen 197 hibás beadást kapott az AKÉP, melyet hibákra lebontva részletesen a függelékben lehet megtekinteni. Itt, a 7. Ábra-n már az összegzett érték látható. A leggyakrabban előforduló hibák közé tartozik a nem megfelelő struktúrájú munkák beadása (rossz fájl/mappa szerkezet: 46 darab, fájl sémája hibás: 46 darab).

Sajnos a SOA feladatok esetén a hibák számos része abból adódott, hogy a hallgatók (pontosan 66 darab) nem az előírt service.py fájlban implementálta a funkcionalitást, hanem szétszedte azt modulokra. A hallgatók munkája „homokozóban" (sandbox) kerül futtatásra, és ezen a területen el is érhető a moduljuk, ugyanakkor az AKÉP a service.py csatlakozáshoz használt függvényüket felülírja, hogy a programjuk a tisztán inicializált értékelő-adatbázisban dolgozzon az adott feladatsor tábláival. A pythonwrapper

sajnos a service.py-ban importált egyéb modulokat nem vette figyelembe, amikor példányosította a hallgató munkáját a fenti felülírással. Ennek a hibának a jövőbeli kijavítása vélhetően nullára redukálja a soa „hibás” beadásokat.

A JDBC feladatok jó része a rossz beadási struktúrából származik (holott a meghirdetett elvárásokkal ellentétben az AKÉP csak a tényleges célfájlokat kereste és nem nézte a könyvtárstruktúrát) (27 darab), másrészt a követelményektől való eltérés a java projektben, melynek hatására nem tudott a CLI belefordulni a projektbe (25 darab).

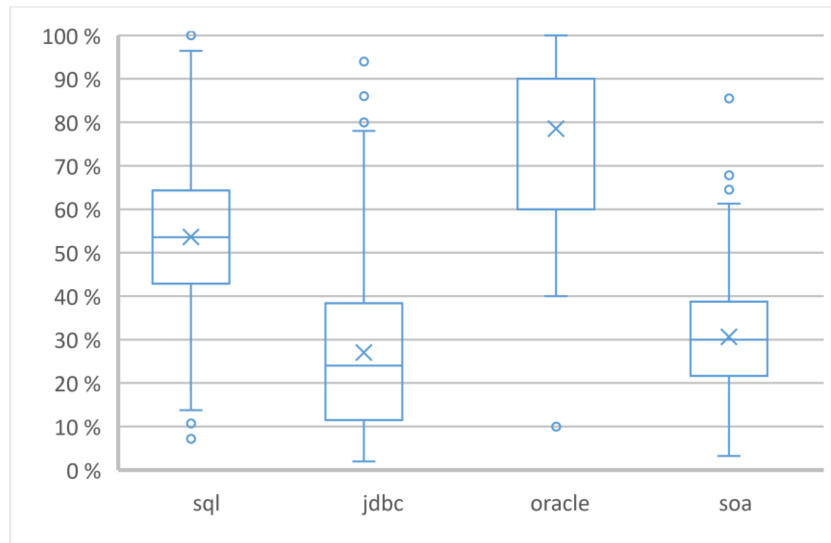
SQL esetén a kiadott szkeleton volt nehezen érthető a hallgatók számára, mint utólag kiderült. Itt a megoldások megfelelő elválasztása érdekében prompt sorok voltak beillesztve a szkeletonba, és bár nyomatékosan meg lettek kérve a hallgatók, hogy ne töröljék, ne írják és helyezzék át ezeket a fejléceket, sokan mégis megtették (45 darab). Ennek érdekében jövőre egy webes felületen fogják beilleszteni az SQL kódjaikat, hogy így garantált legyen a helyes szeparáció.



7. Ábra AKÉP által nem feldolgozható hibás beadások laboronként

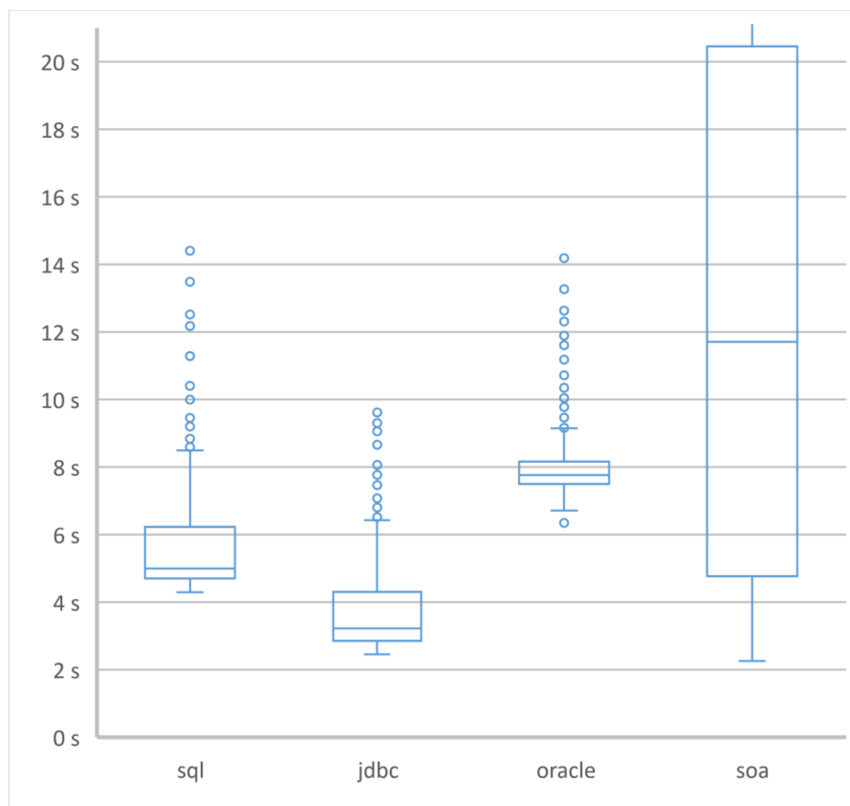
Jöjjenek az AKÉP által elvégzett javítások(doboz diagram (boksz plot) segítségével). A rajtuk szereplő x jel az átlagot, a dobozon belüli vonal a mediánt, a doboz alja az alsó kvartilist (az adatok 25%-a az adott érték alatt van), a doboz teteje a felső kvartilist (az adatok felső 25%-a az adott érték felett van) és a körök a kiugró értékeket mutatják. A 8. Ábra-n százalékosan látható a javított munkák eredménye (a 100% a kapható maximális pontszámot jelenti az adott feladatsorra). Itt megjegyezném, hogy bár az összefoglalóban leírt manuális pontozással való összehasonlítás elvárt lenne, sajnos nem lehetséges, mert az csupán egyetlen ömlesztett string érték a Szlab5 adatbázisában minden egyes javításnál a javító saját szájíze szerinti megfogalmazásban. Pár száz beadás esetén még talán neki esne kézzel az ember, de több mint kétezernél nehézkesen. Ugyanakkor általánosságban elmondható, hogy az automatizált SQL eredmények nagyon jól közelítették a valós (manuális) pontozást, az Oracle pedig teljes mértékben (itt viszont a feladatok csak

egy részét vizsgálta automatizmus, a többivel ellentétben). A SOA és JDBC pedig a sok feladatsor esetén rosszul megválasztott tesztadatokból adódóan (illetve a SOA esetén az elkövetett AKÉP hibából származóan), illetve a hallgatók követelmények betartásának hiányában csúszott lejjebb a valós értékektől a doboz diagrammon.



8. Ábra AKÉP által javított feladatok eredményei százalékban laboronként

Végül az egyes labormunkák értékelésének gyorsaságát mutatom be a 9. Ábra-án. Minden beadott munka elején átmásolódik, kicsomagolódik a hallgató munkája, lefut a hozzá tartozó feladatlaphoz a tábláit inicializáló szkript, majd folytatódik a laborspecifikus csatornák lefutásával. SQL esetén a hallgató munkáját lefuttató SQLcl, mely képes szeparáltan visszaadni a táblákat. JDBC esetén a hallgató munkájának kiegészítése a CLI-vel majd lefordítása és futtatása a hozzáadott oracle driver könyvtárral. SOA esetén a hallgató szolgáltatásának egy pythonwrapperen keresztül történő elindítása és mellette a SOA Preprocesszor elindítása. A SOA-nál látható magasabb értékeket az magyarázza, hogy a hallgatók webes felületét is teszteli egy phantomjs nevű eszköz, itt pedig 10 és 15 másodperces időtúllépésig vár, továbbá ha a hallgató nem oldott meg egy feladatot a 404 hiba érkezéséig is több másodperc is eltelhet (amennyiben ez sokszor, több feladatnál is megtörténik annak lesz az az eredménye, amit a doboz diagrammon lehet látni). Megjegyzés: a diagrammon csak a könnyebb láthatóság kedvéért vettem a skálát 21 másodpercig, a SOA maximuma 37 másodperc. Természetesen ezek az idők a teljes kiértékelést (feladatot kapott az AKÉP, végrehajtotta a feladatot) felölelik.



9. Ábra Az értékelések lefutása laboronként

4.3. Jövőbeli tervek

Elsődleges pont a Szlab5 összes feladatleírójának átnézése és javítása (elvárások pontosítása), hiszen ennek megfelelően érhető el nagyobb AKÉP hatékonyság, ebben nagy segítségre lesz a bevezetésre kerülő feladatleírók elkészítésében és tesztelésében segítő webes felületű alkalmazás. Kódanalízis, kódminőséget ellenőrző eszközök becsatolása a csatornákba, szintén egy első helyen lévő jövőbeli terv. A hallgatók általi többszöri beadások közötti különbségek elemzése, nem csak a forráskódban, hanem a kimeneti eredményben és a logok között, hogy ezekből következtethető legyen egy fejlődési minta. A tervek között szerepel még az AKÉP számára egy dashboard készítése, melyen valós időben láthatók az analizálás eredményei.

5. Jegyzékek

Ábrajegyzék

1. Ábra[2] forrásban Mooshak-ban készített tevékenységekre használt eszközök mérési eredményei 16
2. Ábrakísérleti csoport és kontrol csoport vizsgaeredménye az egyes teszteknél a [2] forrásban..... 16
3. ÁbraJupyter Notebook architektúra felszínesen, forrás: [4] 20
4. Ábra AKÉP architektúra 24
5. Ábra Az AKÉP keretrendszere és a preprocesszorok, feladatleírók, hallgatói munka kapcsolata . 31
6. Ábra A Szoftver laboratórium 5 tárgy rendszerébe integrált AKÉP megjelenítő 44
7. Ábra AKÉP által nem feldolgozható hibás beadások laboronként 61
8. Ábra AKÉP által javított feladatok eredményei százalékban laboronként 62
9. Ábra Az értékelések lefutása laboronként 63

Kódpélda jegyzék

1. Kódpélda Referencia definiálása feladatleíróban 28
2. KódpéldaTimeStamp fordítása dinamikus értékre JSON objektumban 29
3. Kódpélda SQL_SCRIPT értéke meghatározva feladatleíróban 29
4. KódpéldaPélda egy teszteset leírására 33
5. Kódpélda Egy lehetséges parancs a kommunikációs modulnak 41

Táblázat jegyzék

1. Táblázat A kiértékelés típusa szerinti osztályozások összehasonlítása. Forrás: [1]irodalom VI. táblázatának fordítása. 9
2. Táblázat Megközelítés dimenziója szerinti összehasonlítások. Forrás: az [1] irodalom VII. táblázatának fordítása. 11
3. Táblázat[2] forrásban használt eszközök 15
4. Táblázat: AKÉP és nbgrader összehasonlítása jellemzők alpján 22
5. Táblázat: AKÉP és nbgrader összehasonlítása osztályozási sémák alapján 23
6. Táblázat A csatornák entry attribútumának értékei, azok jelentése 29
7. Táblázat Lehetséges csatorna bemeneti fajták 30
8. Táblázat Kezelt hibák egy részhalmazának felsorolása az egyes modulokra 35
9. Táblázat Jelenleg elérhető értékelő függvények funkciója 45
10. Táblázat A script nyelvi elem opcionális paraméterei 57
11. Táblázat A solution nyelvi elem opcionális paraméterei 58

6. Irodalomjegyzék

[1] K. R. F. E. F. B. Draylson M. Souza, „A Systematic Literature Review of Assessment Tools For Programming Assignments,” in *2016 IEEE 29th CSEE&T*, 2016 Április.

[2] J. L. F. Aleman, „Automated Assessment in a Programming Tools Course,” *IEEE Transactions on Education,* %1. kötet54, pp. 576 - 581, 2011.

[3] U. o. P. Vreda Pieterse, „Automated Assessment of Programming Assignments,” *CSERC '13 Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research,* pp. 45-56, 2013.

[4] „Jupyter notebook,” [Online]. Available: http://jupyter-notebook.readthedocs.io/en/latest/notebook.html. [Hozzáférés dátuma: 21 10 2016].

[5] „Jupyter nbgrader,” [Online]. Available: https://nbgrader.readthedocs.io/en/stable/. [Hozzáférés dátuma: 21 10 2016].

[6] R. M. H. R. P. S. M. S. Frank Buschmann, „Pattern-Oriented Software Architecture Volume 1: A System of Patterns,” England, John Wiley & Sons Ltd., 1996, pp. 53-71.

[7] M. S. H. R. F. B. Douglas C. Schmidt, Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects, England: John Wiley & Sons Ltd, 2000.

[8] „GitHub - NavTree library,” [Online]. Available: https://github.com/nickperkinslondon/angular-bootstrap-nav-tree. [Hozzáférés dátuma: 13 október 2016].

[9] „Python.org - Privát tagok,” [Online]. Available: https://docs.python.org/3.4/tutorial/classes.html#tut-private. [Hozzáférés dátuma: 14 10 2016].

[10] „SQLcl Oracle,” Oracle, [Online]. Available: http://www.oracle.com/technetwork/developer-tools/sqlcl/overview/index.html. [Hozzáférés dátuma: 15 10 2016].

[11] „SQL*Plus Oracle,” Oracle, [Online]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14357/toc.htm. [Hozzáférés dátuma: 15 10 2016].

[12] M. G. Dr., „Elosztott rendszerek és szakterületi modellezés,” [Online]. Available: https://www.aut.bme.hu/Upload/Course/VIAUMA01/hallgatoi_segedletek/Ea05_SzovegesSzakteruletiNyelvek_2016.pdf. [Hozzáférés dátuma: 9 5 2016].

Függelék

A következő táblázat (a 4.2 fejezetet kiegészítve) az AKÉP-nek beadott hibás munkák hibáit részletezi, színezéssel jelölve a rossz fájl/mappa szerkezetet (46 darab), rossz fájl sémát (46 darab). Előbbit sárgával, utóbbit narancssárgával.

Sorcímkék	jdbc	soa	sql	Végösszeg
Element 'task': Duplicate key-sequence ['.']			1	1
Element 'task': Duplicate key-sequence ['1.1']			1	1
Element 'task': Duplicate key-sequence ['2.2']			1	1
Element 'task': Duplicate key-sequence ['2.7']			1	1
Element 'task': Duplicate key-sequence ['3.1']			1	1
Element 'task': Duplicate key-sequence ['3.3']			1	1
Element 'task': Duplicate key-sequence ['4.3']			2	2
Error in script: CLIToStudent	13			13
Error in script: convertToUTF8			1	1
Error in script: javaCompiler	25			25
Error in script: run_PythonService		30		30
Error in script: solutionFileChoose	27	1	2	30
Error in script: unzipper			1	1
Error in script: unzipper2	2			2
line 136: b'Extra content at the end of the document'			1	1
line 152: b'Extra content at the end of the document'			1	1
line 159: b'Extra content at the end of the document'			1	1
line 213: b'Extra content at the end of the document'			1	1
line 264: b'Extra content at the end of the document'			1	1
line 286: b'Extra content at the end of the document'			1	1
need more than 0 values to unpack	13			13
Output channel schema other error			31	31
Time expired script: Main		36	1	37
Végösszeg	**80**	**67**	**50**	**197**

Az elkészített AKÉP forráskód megtalálható a következő linken:

https://github.com/nyikesda/akep-framework/releases/tag/AKEP2