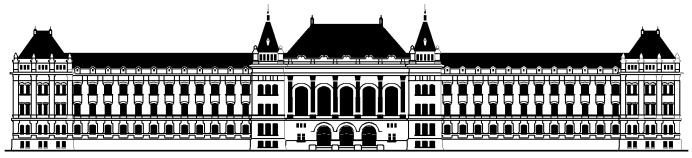


BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM

VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR

IRÁNYÍTÁSTECHNIKA ÉS INFORMATIKA TANSZÉK

A WIKIPÉDIA VALÓSIDEJŰ NYELVI FELDOLGOZÁSA OSGI ALAPOKON

SZAKDOLGOZAT

Készítette
UNICSOVICS MILÁN GYÖRGY

Konzulensek
HÉDER MIHÁLY, SIMON BALÁZS

2013. december 8.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Unicsovics Milán György*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2013. december 8.

Unicsovics Milán György
hallgató

Kivonat

A részben struktúrált adatok minél pontosabb feldolgozása rendkívül fontos a mesterséges intelligencia, természetes nyelvi feldolgozás, de az informatika más területein is. Az ilyen részben struktúrált adatforrásokat, mint például a legnagyobb kollaboratívan szerkeszthető tudásbázist, a Wikipédiát felhasználó megoldások terén mégis még mindig nagyon sok fejlesztési lehetőség található. Szakdolgozatom célja egy kutatók számára is használható rendszer tervezésének és fejlesztésének bemutatása, mely a Wikipédiát dolgozza fel, és feldolgozott formában teszi elérhetővé kutatások céljából. A rendszert egy feldolgozólánc formájában valósítottam meg, mely az OSGi keretrendszert használva, annak flexibilitását kihasználva egyszerűen kiegészíthető, illetve továbbfejleszthető, így alkalmassá válik további kutatások kiindulópontjaként is szolgálni.

A tervezés előtt megvizsgáltam a hasonló megoldásokat, hiszen azok tapasztalatait érdemes felhasználni: hibáikból tanulni lehet, a jól bevált megoldásokat pedig meg kell fontolni, hogy beépíthetőek-e az alkalmazás architektúrájába. Az elérhető részben struktúrált adatforrásokat feldolgozó alkalmazásokat végigvizsgálva mindegyikben egy alapvető hiányosság tűnt fel: egyik sem elég flexibilis, és a legtöbb program csak egy nagyon speciális célra alkalmas, így általános kutatások alapjául nem használhatóak. A megoldások közös jellemzője, hogy alapvetően három jól meghatározott, nagyobb előfeldolgozó komponenst tartalmaznak, így ezeket a komponenseket (információk gyűjtése, feldolgozása, eltárolása) terveztem meg és implementáltam én is a követelményeknek megfelelő módon.

A rendszer architektúráját nagyban meghatározza a flexibilitást és karbantarthatóságot eredményező OSGi technológia, melynek komponens modellje egy megszokottól eltérő Java nyelvű fejlesztést tesz lehetővé.

A tényleges tervezésnél az OSGi komponenseket külön-külön készítettem el, figyelve arra, hogy az alkalmazás teljesítménye minél nagyobb legyen. A gyorsaságot legnagyobbrészt a többszálú futásnak köszönheti az alkalmazás, de a feldolgozás több részének aszinkron kivitelezése és nagy teljesítményű technológiák alkalmazása is sokat segített.

Az implementációs fázisban ismertetem a kipróbált technológiákat, bemutatom, hogy melyiknek milyen előnyei, hátrányai vannak, és miért esett rájuk a választás. A technológiai elemeken kívül a tervezési minták használata is hangsúlyos az elkészített munkámban.

A kivitelezés eredményeit végül egy mérésen keresztül mutatom be, mely továbbfejlesztési lehetőségeket tár fel. Itt mérésekkel alátámasztva ismertetem, melyik részt, hogyan lehetne továbbfejleszteni a specifikációtól eltérve, hogy a követelményeknek megfelelő és egyben hiánypótló alkalmazás szülessen a részben struktúrált feldolgozórendszerek körében.

Abstract

Using of semi-structured data is very important in the field of artificial intelligence (AI), natural language processing (NLP) and in other branch of computer sciences too. Nevertheless, there are lots of opportunities in developing those softwares that are using semi-structured datasources, for example the greatest collaboratively editable knowledge base, the Wikipedia. My thesis aims to present the full lifecycle of a new research tool that can help in computer science researches and it is able to process the articles of Wikipedia and publishes the result to the researchers.

Before the design, I investigated similar solutions, because from their experience we can learn lot. We can avoid common mistakes, and for example use a good architecture. After examining the available solutions, it can be see that none of them is flexible and they can be used only at specific tasks. Common property of the applications that they all have three major, well defined components (gathering, processing and storing the data). These three components were designed and implemented in my thesis regarding to the specified requirements.

The architecture of the system, based on OSGi module system and service platform that comes with flexibility and maintainability. The OSGi framework extends Java language's abilities with a component driven development method.

At the design, phase the separate OSGi components were created one-by-one, taking care of high performance. The speed of the application comes from the multi threaded approach, the async implementation and the usage of high performance technologies.

At the implementation chapter of the thesis tested technologies are demonstrated, covering the advantages and disadvantages of each technologies. Besides the technical elements, the design patterns are cardinal in my thesis too.

The results of the implementation will be demonstrated via the measurement chapter, which leads to opportunities of further development. The further development ideas are proven by measurements, and it is explained how it is possible to modify the application to create a useful software in the field of semi-structured data processing systems.

Bevezető

Mai világunkban a tudás a legnagyobb érték. A tudás valójában kontextusba ágyazott információ, mely elemi adatokból épül fel. Hosszú ideje irányulnak kutatások és fejlesztések az informatikában a tudás minél hatékonyabb megszerzésére, azonban, ha egy nehéz és összetett problémát akarunk megoldani valamilyen módszerrel, azt tapasztaljuk, hogy a hozzá szükséges tudás megszerzése lesz mindig a szűk keresztmetszet [7] (ez az ún. *knowledge acquisition bottleneck*).

Ez a helyzet egy ideje a különféle adatábrázolási módszereknek köszönhetően kezdett megváltozni. A struktúrálatlan, nehezen feldolgozható adatok mellett megjelentek a struktúrált és szemi-struktúrált adatok, melyekből az információt sokkal gyorsabban és könnyebben lehet kinyerni, és olyan automatizált folyamatokban is felhasználhatóak már, ahol korábban emberi közreműködés lett volna szükséges. Megjelentek a kollaboratívan szerkeszthető tudásbázisok, ezek közül is a legnépszerűbb és legnagyobb a Wikipédia, mely egy részben struktúrált információforrás. Ezen tudásbázisokat főleg a mesterséges intelligencia, természetes nyelvi feldolgozás [3], valamint a számítógépes nyelvészet területén, de az informatika szinte minden ágában ugyanúgy felhasználják.

A részben struktúrált információforrások, mint a Wikipédia (továbbiak például a Flickr, Twitter vagy a Yahoo! Answers) egyesítik a struktúrált és struktúrálatlan források előnyeit, így rendkívül jó kiindulópontjai különféle kutatásoknak. A struktúrálatlan adatokkal szembeni előnye, hogy az emberek számára is olvasható és a gépek számára is értelmezhető információt tárolnak; a struktúrált adatok előállításánál és kezelésénél pedig kevésbé erőforrás-igényes a szemi-struktúrált adatok létrehozása és karbantartása.

Az informatika ezen területén történő kutatások már 2000-es évek elejétől megkezdődtek, azonban az ezek eredményeit felhasználó fejlesztések száma nem túl sok. Célom tehát az eddig elkészült és elérhető fejlesztések vizsgálata, információk és tapasztalatok összesítése, valamint ezeket továbbfejlesztve egy korszerűbb rendszer összeállítása, melyre alapozva eredményes kutatásokat lehet kezdeményezni, a legnagyobb elérhető szemi-struktúrált erőforrás a Wikipédia segítségével, a lehető legflexibilisebb módon.

Az általam készített rendszer feladata addig terjed, hogy előállítson, karbantartson és üzemeltessen egy folyamatosan frissülő tudásbázist, amelyben minden Wikipédia cikkhez tartozik egy objektum reprezentáció, amely a cikket már előfeldolgozott, könnyen kezelhető formában tartalmazza. Erre az tudásbázisra építve az MTA SZTAKI kutatói saját modulokat hozhatnak létre, amelyekben már nem kell a cikkek letöltésével, frissítésével, átalakításával foglalkozni.

1. fejezet

Hasonló megoldások vizsgálata

Mielőtt a saját Wikipédia alapú rendszer tervezésének nekiláttam volna megvizsgáltam a hasonló célú, elérhető megoldásokat, hogy utána a tapasztalatokból kiindulva láthassak neki a munkának. Megpróbáltam összegyűjteni az összes hasonló témakört feldolgozó alkalmazást, majd ezeket egyesével megvizsgáltam, milyen előnyökkel, hátrányokkal rendelkeznek.

WikiNet[21] A WikiNet egy teljesen struktúrált adatszerkezetű tudásbázist, más néven ontológiát épít fel. Ez egy Perl nyelvű szkriptek gyűjteményéből álló megoldás, a Wikipédia egy statikus, letöltött verzióját használja forrásként, a kinyert fogalmakat és kapcsolatokat külön szöveges fájlokban tárolja el.

Elérhetőség:
`http://www.h-its.org/english/research/nlp/download/wikinet.php`

DBpedia Ez a megoldás is a Wikipédia dump-ján alapul, belőle tényszerű információkat nyer ki és rögzít struktúrált formában, melyet ezután a weben közzétéve a felhasználók számára egy SQL szerű nyelvvel lekérdezhetővé tesz. A alkalmazás Scala, Java nyelven íródott, adatbázisként Virtuoso Universal Server-t használ.

Elérhetőség:
`http://dbpedia.org/`

BabelNet[18] A BabelNet egy Java nyelven írt alkalmazás, ontológiát készít más adatforrások (Wikipédia dump, WordNet) alapján és ezekből egy „enciklopédikus szótárat” készít a felhasználók számára. Egy adott fogalomra keresve a szemantikailag kapcsolódó fogalmak is megjelennek, valamint ezek szinonimái (a BabelNet elnevezése szerint *synset*-ek), és minden szinonimához tartozik egy rövid definíció (*gloss*) több nyelven.

Elérhetőség:
`http://lcl.uniroma1.it/babelnet/`

Java Wikipedia Library[20] A JWPL egy Java alapú alkalmazás, mely egy interfészt ajánl ki, amivel a Wikipédia tartalmához lehet hozzáférni. A JWPL tartalmaz egy Mediawiki Markup parser-t, mellyel a letöltött Wikipédia dump-ot beolvassák, a wikitext-ből átalakított

szövegekből optimalizált adatbázisokat készítenek, melyekhez végül hozzáférési felületet nyújtanak.

Elérhetőség:
`http://www.ukp.tu-darmstadt.de/software/jwpl/`

Wikipedia Preprocessor Ezzel az eszközzel más alkalmazások számára lehet egy Wikipédia dump-ot feldolgozhatóbb formába hozni. A statikus dump-ból kiinduló alkalmazásoknál erre szükség is van, mivel a például Wikipédia 2013. október 2-i XML formátumban letölthető tartalma is 44 GB méretű, amit nyers formában szinte lehetetlen hatékonyan felhasználni.

Elérhetőség:
`http://www.cs.technion.ac.il/~gabr/resources/code/wikiprep/`

YAGO2[13] A YAGO2 a Wikipédia egy korábban letöltött tartalmán, és egyéb online tudásbázisokon (WordNet, GeoNames) alapuló ontológia. Az ontológiához több formátumban lehet hozzáférni, ezt kisebb Java nyelven írt konvertáló eszközökkel lehet megtenni.

Elérhetőség:
`http://www.mpi-inf.mpg.de/yago-naga/yago/`

1.1. Egy ígéretes megoldás: Wikipedia Miner

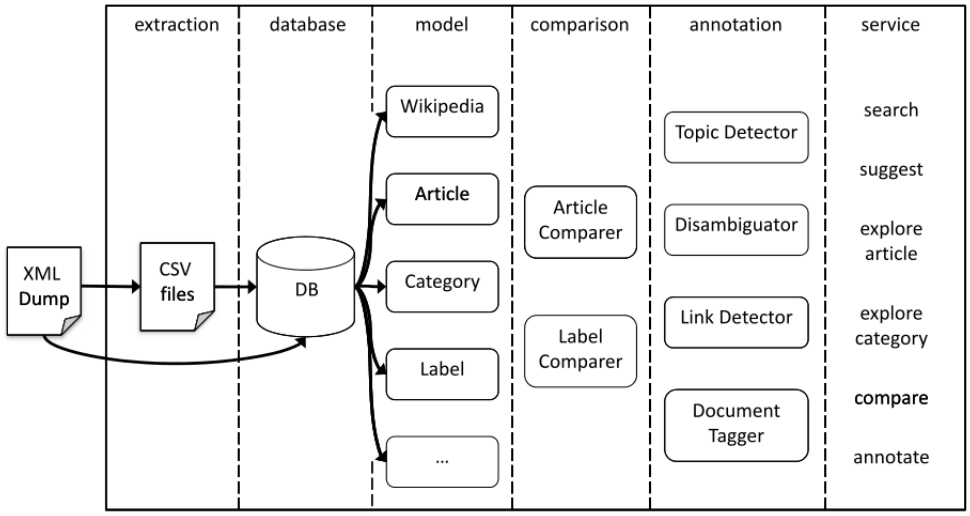
A Wikipedia Miner egy olyan nyílt forráskódú teljes egészében Java nyelven írt eszközkészlet, mellyel lehetővé válik kutatók és fejlesztők számára, hogy alkalmazásukban egyszerűen hozzáférjenek a Wikipédia tartalmához. A hozzáférési felületet Java API-n keresztül nyújtja az alkalmazás, a tudásbázis tartalmát pedig egy összegzett formában használja fel. Ezenfelül a rendszer olyan technológiákat használ fel, mint az elosztott számítási keretrendszert nyújtó Apache Hadoop és a Weka adatbányászati szoftver.

A Wikipedia Miner alkotói szerint három lehetőség van a szemi-struktúrált adatforrások felhasználására: vagy kész ontológiákat használunk, mint például a YAGO2, vagy egy nyers letöltött dump-ból elkészítjük a saját ontológiánkat, mint például a JWPL és az alapján kezdünk dolgozni; a harmadik lehetőség, hogy folyamatosan frissítjük az adatforrásunkat, így mindig naprakész forrást használunk.

A rendszer áttekintő architektúráját mutatja be a 1.1. ábra. A rendszer belépési pontjára érkezik egy nagy méretű XML formátumú fájl, mely a Wikipédia hivatalos API-ján keresztül lett letöltve és tartalmazza a naprakész Wikipédia teljes állományát.

Az *extraction* package funkciója, hogy kinyerjen az XML forrásból egy összegzett tartalmat. Az adatkinyerési folyamat használja a nagy méretű XML fájl feldolgozásához a Hadoop és MapReduce technológiákat, valamint a Google GFS fájlrendszerét. Az adatok kinyerése tehát elosztott rendszeren történik, a 27 GB méretű teljes angol Wikipédia feldolgozása a mérések szerint egy 2 magos, 2,66 GHz órajelű processzorral és 4 GB memóriával rendelkező 30 gépből álló clusternek 2,5 órájába telik.

A Wikipédia kivonatolása után a teljes XML fájl és az összegzett tartalmak is bekerülnek a *database* package adatbázisába. Adatbázisként Berkeley DB Java Edition-t használnak, mellyel



1.1. ábra. Wikipedia Miner architektúra (forrás: Artificial Intelligence, Wikipedia and Semi-Structured Resources[7])

akár egy teljes adatbázist is a memóriában lehet tartani, ezzel nagyon gyors lekérdezhetőséget elérve.

A Wikipedia Miner keretrendszert használók a tartalmakhoz a *model* package absztrakciós felületen keresztül férhetnek hozzá, mely becsomagolja a kinyert információkat és jól dokumentált osztályok formájában teszi közzé (mint például: `Wikpedia`, `Article`, `Category`).

A további csomagok (*comparison* és *annotation*) már a felhasználók számára használható szemantikus tartalom kezelését segítő eszközök, a *service* package-ben pedig konkrét szolgáltatások találhatóak, melyeket felhasználhatnak a fejlesztők, illetve kiegészíthetik őket.

1.2. Tanulságok

Amint látható a fentebb leírt Wikipédiát felhasználó eszközök szinte mind kizárólag a Wikipédia egy korábban letöltött változatán (dump) alapulnak, mely módszernek több hátránya is van. A letöltött tudásbázis mérete rendkívül nagy, így azt kezelni nehézkes, feldolgozási ideje rendkívül hosszú (a Wikipedia Processor feldolgozási ideje például körülbelül 43,5 óra). A hosszú feldolgozási idő nem mindig engedhető meg, ráadásul amíg nincs feldolgozott forrás, a rendszer sem működőképes. Újabb dump letöltésekor kezdhető előlről a feldolgozás, így többször is megakaszthatja ez a folyamat a rendszer működését, a rendelkezésre állási időt csökkentve.

A másik megoldás, melyet a Wikipedia Miner (1.1. alfejezet) is demonstrál a Wikipédia folyamatos *on-the-fly* feldolgozása. Míg az előző módszer figyelmen hagyja a közösségi tudásbázisok fő erejét, hogy rendkívül dinamikusan fejlődnek, az on-the-fly megoldás kihasználja azt, és mindig a friss adatokkal dolgozik.

További fontos megállapítások, hogy egyik rendszer sem elég flexibilis: futásidőben új komponens beépítése egyáltalán nem lehetséges, nem lehet a feldolgozólánchoz új elemet (kutatást végző modult) illeszteni, szerkezetük szinte mindegyiknek rendkívül statikus és csak arra a célra használhatóak konkrétan, amilyen speciális feladat ellátására kitalálták. Ebből adódóan rendkívül

specifikusak és bonyolultak, így újabb kutatások indítása eredményeiket felhasználva nehézkes.

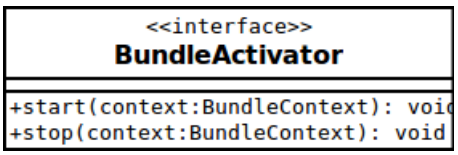
A szoftverek bármely módosítása újrafordítást, rendszerleállást eredményez, ami egy aktívan használt program számára nagy kiesést jelenthet. On-the-fly feldolgozásnál a rendszer leállása kihagyott, nem feldolgozott információkkal jár, így a tudásbázis töredezetté válhat. Ezen okok miatt szükséges egy olyan megoldás, mellyel a rendszer sokkal flexibilisebbé tehető és a fenti problémák áthidalhatóvá válnak.

2. fejezet

Az OSGi keretrendszerről

Az OSGi Alliance [1] által fejlesztett OSGi (*Open Services Gateway initiative*) egy Java nyelvű, dinamikus, szolgáltatás orientált komponens modell és keretrendszer, mellyel kiterjeszthető az alap Java nyelven készült programok funkcionalitása. Használatával elérhető válik egy komponens alapú fejlesztés, ahol alkalmazásokat (illetve komponenseket) távolról elérve telepíthetjük, elindíthatjuk, leállíthatjuk, frissíthetjük, vagy törölhetjük anélkül, hogy a teljes alkalmazást leállítanánk és újra elindítanánk. A standard Java nyelven íródott alkalmazásoknál ez egy fontos hiányosság, hiszen az információs rendszerek sok területén nem engedhetőek meg akár a pillanatnyi leállások sem.

A flexibilitás és újrafelhasználhatóság követelményeknek tehát nagyon jól megfelel az OSGi keretrendszer, ezért is esett rá a választás. Léteznek természetesen más, komponens alapú modellt használó technológiák, például: Microsoft Common Object Model, Enterprise JavaBeans, CORBA Component Model. Ezek közül azonban a legtöbb egy komplex programozási modellen alapul, amely megköveteli konvenciók követését és az OSGi-al szemben nem támogatják a komponensek dinamikus frissítését, így flexibilitás szempontjából ilyen módon elmaradnak.



2.1. ábra. BundleActivator osztálydiagram

2.1. Bundle

Az OSGi technológiát [2][16][19] használó alkalmazások kisebb komponensekre (az OSGI terminológiát használva csomagokra, azaz *bundle*-ökre) vannak bontva. Ezen csomagok elkészíthetőek, lefordíthatóak, telepíthetőek egymástól függetlenül, életciklusukat maga az OSGi keretrendszer felügyeli. A bundle-ök gyakorlatilag nem mások, mint a jól ismert JAR fájlok (Java osztályok, és egyéb erőforrások becsomagolva), azzal a különbséggel, hogy a leíró manifest állományban a szabvány által kiterjesztett módon további fejlécek találhatóak meg. Példát látunk manifest állományra a 2.1. kódrészletben. A manifest-ben lévő metaadatok nagy része embe-

ri felhasználás céljából szerepel és nem módosítják az OSGi rendszer működését, kivétel a `Bundle-Activator` és az `Import-Package` fejlécek.

A manifest állomány elemei:

Bundle-Name az elkészített bundle emberek számára olvasható neve

Bundle-SymbolicName Java package név, ez azonosítja a bundle-t (az egyetlen kötelező elem)

Bundle-Description a bundle hosszabb szöveges leírása

Bundle-ManifestVersion a bundle által használt OSGi változat verziószáma

Bundle-Version a bundle verziószáma

Bundle-Activator `BundleActivator` interfészt implementáló bundle-ben lévő osztály, mely a bundle telepítése után elindul

Export-Package más bundle-ök számára elérhetővé tett saját csomagok és verziószámaik listája

Import-Package a bundle fordításához és futtatásához szükséges külső csomagok listája

Private-Package olyan saját csomagok, amelyeket nem teszünk elérhetővé más bundle-ök számára

A bundle-ök egy OSGi példányon belül, azonos JVM-ben futnak. Ennek vannak előnyei (teljesítmény növekedés, kisebb erőforráshasználat, interprocessz kommunikációt nem szükséges használni), de hátrányai is (hozzáférési problémák), melyeket az OSGi úgy old meg, hogy minden bundle-höz saját classloader-t rendel.

2.1. Listing. MANIFEST.MF

```
1 Bundle-Name: Hello World
2 Bundle-SymbolicName: org.available.helloworld
3 Bundle-Description: A Hello World bundle
4 Bundle-ManifestVersion: 2
5 Bundle-Version: 1.0.0
6 Bundle-Activator: org.available.helloworld.Activator
7 Export-Package: org.available.helloworld;version="1.0.0"
8 Import-Package: org.osgi.framework;version="1.3.0"
9 Private-Package: org.notavailable.helloworld
```

Ha egy komponens kapcsolatba akar lépni az OSGi keretrendszerben lévő más komponenssel, akkor azt a hozzá tartozó egyedi `BundleContext`-en keresztül teheti meg. A `BundleContext` megszerzéséhez implementálnia kell a `BundleActivator` interfészt (2.1. ábra), melynek `.start()` és `.stop()` metódusainak argumentumaként megkapják a bundle kontextust, illetve előbbi metódusok meghívódnak a komponens indításakor és leállításakor.

2.2. Service, Service Registry

A bundle-ök kiajánlhatnak szolgáltatásokat (*service*), melyekre más bundle-ök feliratkozhatnak. Az OSGi specifikáció szerint a szolgáltatások normál Java objektumok, melyek egy adott interfészt implementálva lettek beregisztrálva az OSGi *Service Registry* moduljába.

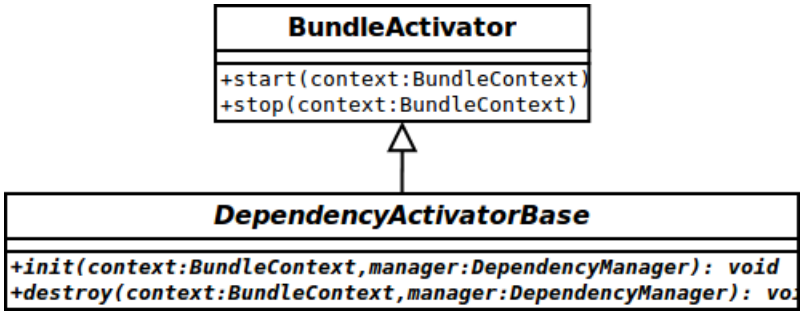
A Service Registry segítségével tudnak tehát a bundle-ök kiajánlani szolgáltatásokat, rajtuk keresztül tudják lekérdezni az elérhető szolgáltatásokat. Ha egy bundle-nek szüksége van egy másik bundle által kiajánlott szolgáltatásra, akkor az OSGi keretrendszertől lekéri a `BundleContext`-en keresztül az adott szolgáltatás referenciáját.

Még robusztusabb használat lehetséges, hogy ha implementáljuk a `ServiceListener` interfészt (`.serviceChanged(ServiceEvent e)` metódust), mellyel értesülhetünk a regisztrált illetve eltávolított szolgáltatásokról, így dinamikusan követhetjük a szolgáltatások elérhetőségét, felkészülhetünk például egy hirtelen kieső szolgáltatás kezelésére. Ugyanezt a funkcionalitást elérhetjük másképpen egy `ServiceTracker` példány használatával is.

A szolgáltatások kezelése ezen módszerek használatával azonban nem skálázódik jól, összetett rendszerekben egyszerűbb technológiák használata jobban kifizetődő. Lássunk tehát néhány példát a szolgáltatások dinamikus kezelésére:

Service Binder A Service Binder komponens létezésének ma már csak történeti okai vannak, mivel a későbbi OSGi R4 specifikációban megjelent újdonságok (Declarative Services, Dependency Manager, iPOJO) teljes mértékben helyettesítik ezt a megoldást. A Service Binder megpróbálja a bonyolult Activator példányok implementációját megkerülni és automatizált módon kezelni a komponens szolgáltatás függőségeit.

A korábbi `BundleActivator` osztályból való leszármaztatás helyett a `GenericActivator` osztályból kell leszármaztatni az osztályunkat, és egy XML fájlban deklaratív módon kell meghatározni, hogy milyen szolgáltatás példányokat akarunk létrehozni és azoknak milyen függőségei vannak.



2.2. ábra. DependencyManager osztálydiagram

Declarative Services A Declarative Services egy komponens modell, melynek felépítését befolyásolta a korábbi Service Binder megoldás. A technológia célja, hogy leegyszerűsítse az olyan OSGi komponensek készítését, melyek OSGi szolgáltatásokat használnak.

Fontos jellemzők, hogy használatával nincs szükség explicit módon implementálni a szolgáltatások kiajánlását és felhasználását, ezek deklaratív módon teljes mértékben

XML segítségével végezhetőek el. Leszármaztatásra vagy interfész implementációra nincs szükség; a szolgáltatások implementációja a *lazy loading* tervezési mintát követi; a komponensek konfigurációja bármikor megtörténhet Configuration Admin szolgáltatás segítségével.

Dependency Manager Ebben az esetben a `DedendencyActivatorBase`-ből kell leszármaztatni az osztályunkat, és implementálni az `.init()` és `.destroy()` metódusokat (2.2. ábra). A `DependencyManager` példány segítségével pedig a Decorator mintát használva adhatunk hozzá szolgáltatásokat, és azok függőségeit a komponenshez, valamint konfigurálhatjuk azokat (2.2. kódrészlet).

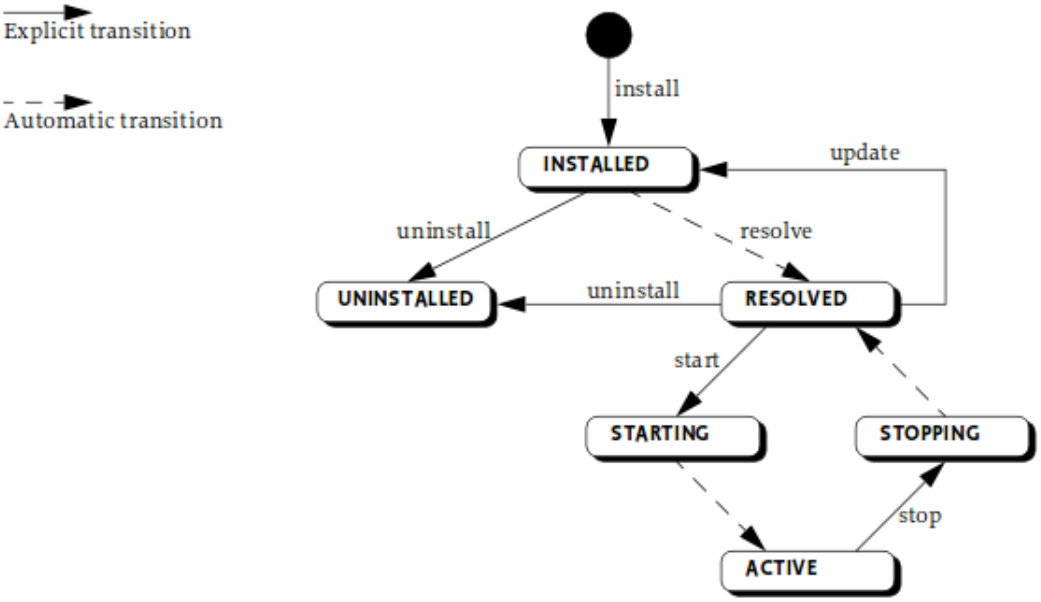
2.2. Listing. Példa a DependencyManager használatára

```
1 manager.add(createService()
2     .setInterface(WebService.class.getName(), null)
3     .setImplementation(WebServiceImpl.class)
4     .add(createServiceDependency()
5         .setService(ConfigurationAdmin.class, null)
6         .setRequired(true))
7     .add(createServiceDependency()
8         .setService(LogService.class, null)
9         .setRequired(false));
```

iPOJO Az iPOJO komponens modell központi koncepciója, hogy ne legyen szükség POJO (*Plain Old Java Object*) példányoknál bonyolultabb objektumok használatára. Használatát tekintve nagyon hasonló a Declarative Services-hez, azonban az iPOJO kicsit több eszközt kínál a fejlesztőknek. A komponenshez szükséges szolgáltatásokat és azok függőségeit szintén XML fájlban kell deklaratív módon specifikálni, és a rendszer automatikusan kezeli ezeket a függőségeket. Újdonságok a korábbi megoldásokhoz képest például a Java annotációk támogatása, API hozzáférés a belső működéshez, könnyű kiegészíthetőség és beépített távoli konfiguráció JMX segítségével.

2.3. Életciklus

Az OSGi keretrendszer dinamikusságát a *Life-cycle* (életciklus) rendszer szolgáltatja, mely által a bundle-öket futásidőben lehet telepíteni, elindítani, leállítani, frissíteni, eltávolítani más hagyományos alkalmazásokkal ellentétben.



2.3. ábra. Bundle életciklus (forrás: OSGi Service Platform Release 2 [1])

A bundle-ök futtatása előtt megvizsgálja a keretrendszer a csomag futás idejű függőségeit, és ha olyan kielégítetlen függőségeket talál, melyek szükségesek a bundle futtatásához, akkor nem indítja el a komponenst. Egy bundle életciklusának állapotai megfigyelhetőek a 2.3. ábrán, valamint az állapotok leírása a 2.1. táblázatban látható.

Állapot neve	**Leírás**
INSTALLED	A bundle sikeres telepítve lett.
RESOLVED	A bundle-nek minden függősége ki lett elégítve. Kész az elindításra, vagy már le lett állítva.
STARTING	A bundle el lett indítva, de még nincs aktiválva, .start() metódus még nem tért vissza.
ACTIVE	A bundle aktiválva lett és aktív.
STOPPING	A bundle le lett állítva, .stop() metódus még nem tért vissza.
UNINSTALLED	A bundle el lett távolítva, ez az életciklus végállapota.

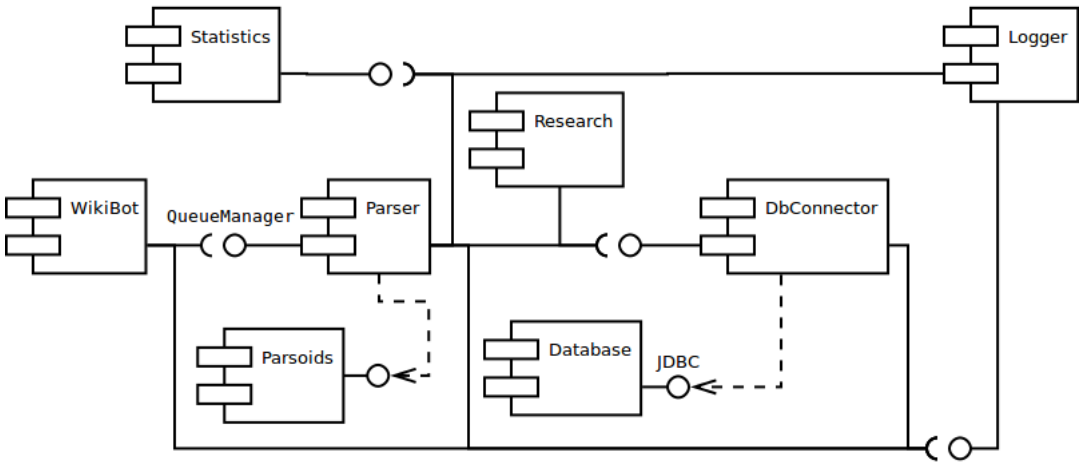
2.1. táblázat. Bundle életciklus állapotai

3. fejezet

Tervezés

Ahogy a 2. fejezetben is láttuk az OSGi keretrendszer megfelel a kiírásban specifikált szoftver által támasztott követelményeknek, felhasználásával a rendszer kellően flexibilis lesz. Az 1. fejezetben levont tanulságok alapján érdemes megtervezni a rendszert, illetve az ott említett Wikipédia Miner architektúráját fel lehet használni a tervezés során, az alapvető komponensek meghatározásában segíthet.

Első lépésként meghatároztam a leendő rendszer komponenseit (3.1. ábra), mely képes folyamatosan működve a Wikipédia tartalmát rendszerezett formában eltárolni és azt később a rá épülő alkalmazások számára elérhetővé tenni. Mivel a rendszer első indításakor az adatbázisban még semmilyen adatok nem találhatóak, és a Wikipédia frissülő cikkeinek követésével, csak az aktuális változások kerülnek be az adatbázisba, a rendszer indításakor már a Wikipédián korábban közzétett cikkek nem kerülnek be a felépített tudásbázisba. Ezt a problémát úgy lehet a legegyszerűbben megoldani, hogy az on-the-fly feldolgozás mellett a statikus dumpból való adatok importálását is lehetővé tesszük. Ezzel kombináljuk a 1. fejezetben megismert alkalmazások előnyét a követelményekben meghatározottakkal, így egy sokkal erősebb eszköz állítható elő.



3.1. ábra. Az alkalmazás tervezett komponensei

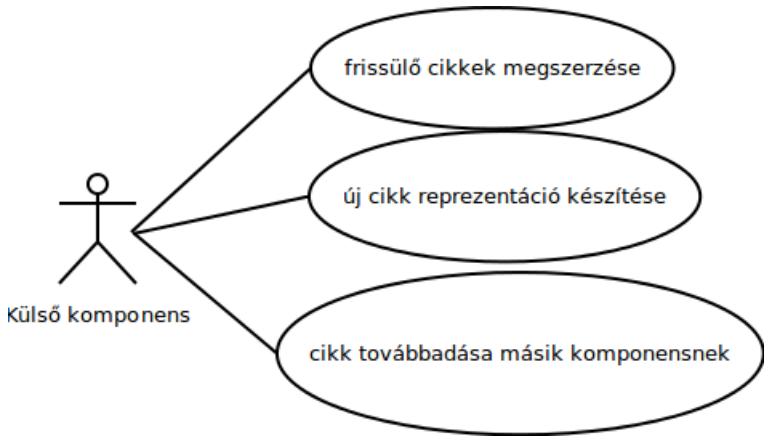
A követelményekből következően szükségszerűen meghatározott komponenseken túl az alkalmazás minőségének javítása érdekében érdemes a komponensek felügyelhetőségét is biztosítani.

Ezáltal az alkalmazás működéséből részletesebb adatok is megfigyelhetővé válnak azon kívül, hogy működik vagy sem a rendszer, illetve a funkcionalitás tesztelése és a teljesítménymérés is sokkal egyszerűbb lesz. A felügyelhetőséget naplózás és különféle metrikák segítségével terveztem lehetővé tenni.

Miután a különálló komponenseket meghatároztam, megkezdődhetett sorban az egyes komponensek megtervezése. Minden komponenst a komponensalapú fejlesztésnek megfelelően egyesével, mint különálló programokat lehetett megtervezni és implementálni. A következőkben bemutatom az egyes komponensek szerepét, alapvető működését és a tervezésük lépéseit.

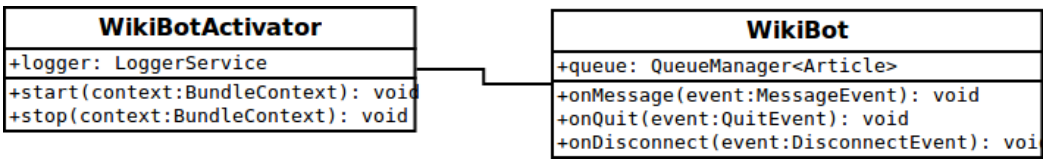
3.1. WikiBot bundle

Az adatok on-the-fly feldolgozásáért felelős ez a komponens. Ezt a tulajdonságot a frissülő cikkek megszerzésének módjával fogom biztosítani. Az adatok könnyű kezelése miatt minden cikknek egy reprezentációját is ki kell alakítani a programban, amelyet tovább kell adni a következő komponensek számára, további feldolgozás céljából. Ezek alapján meghatározott use-case-ek láthatóak a 3.2. ábrán.



3.2. ábra. A WikiBot komponens használati esetei

Megtervezését egy rövidebb előkutatás előzte meg, mely során keresni kellett valamilyen lehetőséget, amelynek segítségével folyamatosan értesülhet az alkalmazás, ha egy új Wikipédia cikk jelenik meg, vagy frissítenek egyet a hivatalos oldalon. A legegyszerűbb megoldásnak végül a hivatalos, Wikipédia által üzemeltett IRC csatorna tűnt, ahol folyamatosan publikálják az oldalon frissülő cikkeket.



3.3. ábra. A tervezett WikiBot osztálydiagramja

A tervezett komponens tehát egy IRC Bot kliens lesz, amely a fent említett IRC csatornára lesz feliratkozva. A tervezésnél figyelni kellett arra, hogy miután megszerzi az új cikk szükséges adatait, azokat azonnal tovább kell adnia a következő komponensnek. Más teendőket nem végezhet,

működését nem szabad hosszú távon feltartani, mert a gyorsan frissülő cikkek miatt elvesznek az IRC csatorna által küldött információk, míg a komponens mással foglalkozik.

A komponens két osztályból fog állni, az egyik egy OSGi BundleActivator implementáció (`WikiBotActivator` osztály), mely menedzseli a komponens indulását, leállítását, a másik pedig a Wikipédia IRC csatornájával való kommunikációt kezeli (`WikiBot`).

A tervezett működés szerint a rendszer a `WikiBotActivator` elindulásával kezdődik, melyet az OSGi keretrendszer példányosít számunkra, majd elindítja azt. A lehető leggyorsabb működés úgy érhető el, hogy ne tartsuk fel a komponens működését, ha az új cikk megszerzett adatait eltároljuk egy átmeneti tárolóban (`QueueManager` osztály). Ez a queue, ahogy a 3.1. ábrán is látható a Parser komponens egy kiajánlott OSGi szolgáltatása. A `QueueManager` osztály bemutatása a 4.2 alfejezetben folytatódik. A kiajánlott queue referenciáját az OSGi `BundleContext`-en keresztül lehet majd megszerezni.

Továbbiakban a `WikiBot` belép a Wikipédia egyik IRC szobájába, ahol az angol nyelvű cikkeket teszik közzé. Az implementált IRC Bot-ban egy eseménykezelőt kell megvalósítani (`.onMessage()` metódus), ami akkor hívódik meg, ha egy új üzenet érkezik az IRC szobába. Egy példa üzenet figyelhető meg a 3.1. kódrészletben.

3.1. Listing. Példa üzenet az angol nyelvű Wikipédia IRC csatornájából

```
1 (11.16.43) rc-pmtpa: [[History of Vietnam]]  http://en.wikipedia
    .org/w/index.php?diff=580135314&oldid=580104406 *
    124.170.231.126 * (+95)
```

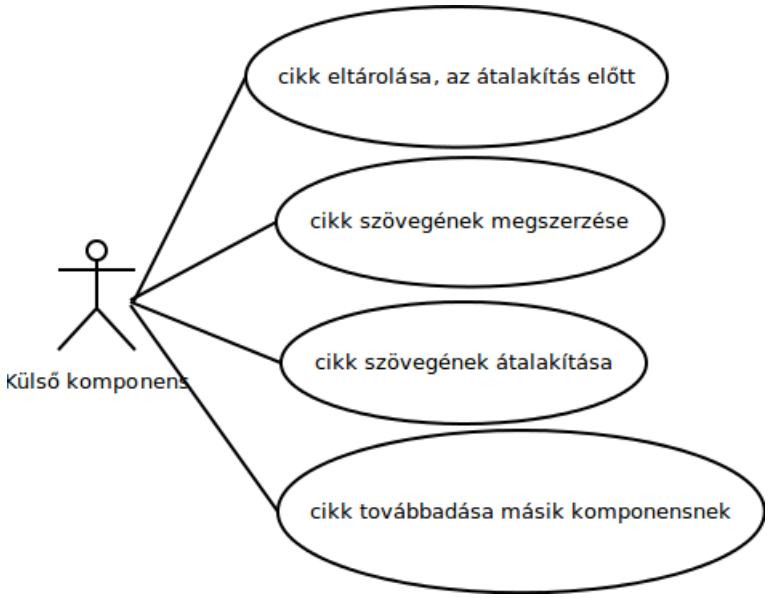
Minden egyes beérkezett üzenet alapján egy új cikk reprezentációt (`Article` osztály) hoz létre, mely később végig fog haladni a teljes feldolgozóláncon. A szükséges adatok a cikk címe, mely dupla szögletes zárójelek között található, valamint az új cikk verziószáma, mely az üzenetben található link `diff` GET paraméterében található. Végül az összeállított cikket a korábban megszerzett `QueueManager`-ben helyezi el.

Az alapvető követelményeken felül a megfigyelhetőséget is érdemes biztosítani, amelyet a `WikiBot` komponensnél naplózással oldottam meg. A `WikiBotActivator` indulásakor a `Logger` komponens (4.4. alfejezet) referenciáját is megszerzi, melyet naplózásra használhat.

3.2. Parser bundle

Ebben a komponensben kell lennie valamilyen tároló elemnek, egyfajta queue megoldásnak, melynek szükségessét az előző 4.1. alfejezetben fejtettem ki. Ennek a queue megoldásnak szálbiztosnak kell lennie, hiszen egyszerre fog a WikiBot és a Parser komponens dolgozni vele, ezenkívül a tárolónak blokkolnia is kell beszúrást, ha már túl sok elem van benne. A módosult, vagy újonnan létrehozott cikkek szövegét meg kell szereznie a komponensnek, majd azt át kell alakítania egy meghatározott formátumra. Végül a cikket tovább kell adnia egy másik komponensnek eltárolás céljából. Ezek alapján készíthető el a a use-case diagram (3.4. ábra).

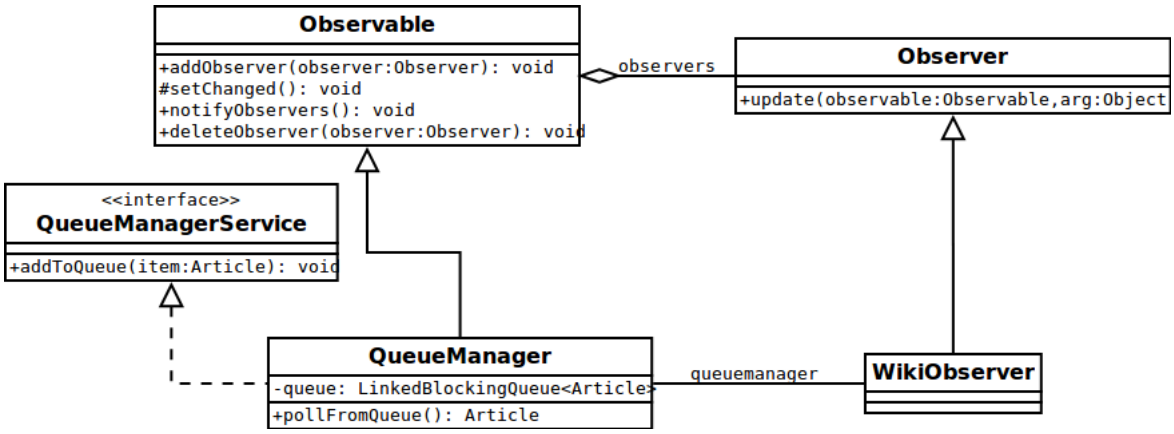
Ahhoz, hogy egy másik komponens el tudjon tárolni a Parser-ben valamit, létre kell hozni a Parser komponensben egy szolgáltatást (OSGi service), amelyen keresztül a kommunikáció megtörténhet. Ez az szolgáltatás egyben az egész program mozgatórugója is, hiszen a további



3.4. ábra. A Parser komponens használati esetei

use-case-ekben található funkciókat akkor tudjuk végrehajtani, ha legalább egy cikk már el lett tárolva.

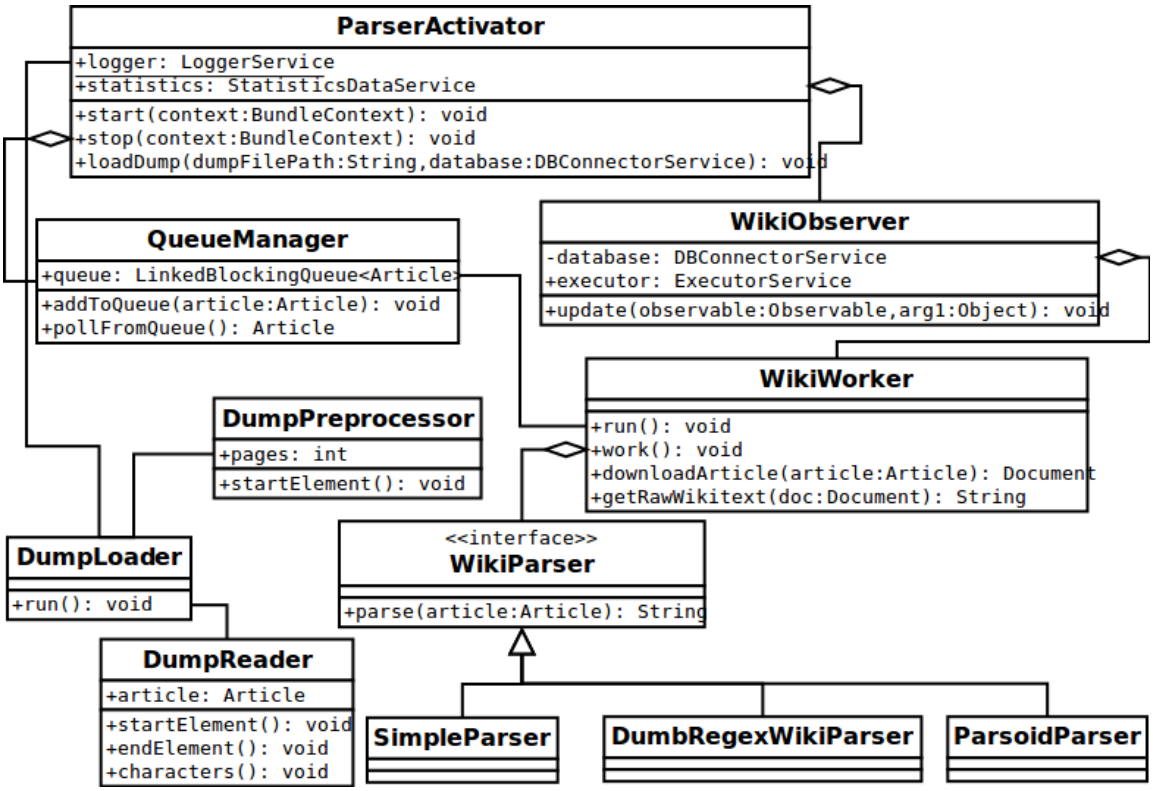
Ennél a résznél használtam az *Observer* tervezési mintát [9]: a `QueueManager` osztály `Observable` lett, míg egy `Observer`-ből származó `WikiObserver` kezelte a tárolóba került új elemeket (3.5. ábra).



3.5. ábra. Observer minta használata a Parser komponensben

Az Observer tervezési mintának köszönhetően, ha új elem kerül a queue-ba, akkor arról a `WikiObserver` értesülni fog. Mivel a cikkek feldolgozása jelentős időt vehet igénybe, azt mindenképpen külön szálakon kell megtenni. Ezt a problémát a *Threadpool* tervezési mintával oldottam meg [6]. A `WikiObserver`-nek (3.6. ábra) van egy `ThreadPool` példánya, és ha értesül a queue frissüléséről, egy új feladatot fog a `ThreadPool`-hoz hozzáadni. A ThreadPool mintának megfelelően minden feladat (`WikiWorker`) külön szálon fut.

Ebben a fázisban első lépésben először letöltődnek a hivatalos Wikipédia API-n keresztül a cikkek tartalmai. A Wikipédia a felhasználók számára könnyebben szerkeszthető *Wikitext*



3.6. ábra. A Parser komponens osztálydiagramja

formátumban teszi elérhetővé a cikkek tartalmait. A Wikitext egy egyszerű, könnyen használható jelölőnyelv, mely egyértelműen leképezhető a HTML formátumra, így könnyen készíthetőek vele webes tartalmak.

Azonban ez a Wikitext formátum kevésbé jól feldolgozható, mint például a HTML formátum, így a cikkek szövegét HTML formába célszerű alakítani. A második fázisban tehát ez az átalakítás történik meg, ha nem volt még újabb verziójú változat az adott cikkből az adatbázisban. Ehhez ún. parser-eket lehet használni, melyből három felcserélhető, különböző előnyökkel és hátrányokkal rendelkező változat is került a feldolgozóláncba.

- Sztakipedia parser [15]: Könnyen testreszabható és általános célú program, mely a MediaWiki Wikitext formátumról tud HTML formátumra átalakítani. Oláh Tibor BME mérnök informatikus hallgató 2011. évi gyakornoki munkája az MTA SZTAKI-ban. Ez a program egy könnyen kiterjeszthető *Visitor* tervezési mintára alapuló szoftver, mely önálló csomagként is használható a jól felépített API-nak köszönhetően. Használata nagyon egyszerű, használható kimenetet állít elő és sebesség szempontjából is jól teljesít, viszont nagyon nagy méretű fájloknál, például egy teljes Wikipédia dump feldolgozásánál rendkívül sok memóriát használ. Ennek a tulajdonságnak köszönhetően, csak on-the-fly feldolgozás során használható eredményesen ez a parser.

- DumbRegexWikiParser parser: A végletekig leegyszerűsített MediaWiki Wikitext átalakító Héder Mihály 2011. évi munkája, mely egy SAX parser implementáció. Az állapotgépes Wikitext feldolgozásnak köszönhetően nem fogyaszt annyi memóriát, mint a Sztakipedia

parser, sebesség szempontjából még gyorsabb is az előzőnél, viszont kevésbé használható kimenetet produkál.

- Parsoid parser: Ez az átalakító a WikiMedia Alapítvány által, 2011 óta fejlesztett és használt szoftver, mely a Wikitext egy ekvivalens HTML / RDFa kimenetét készíti el, mely automatikus feldolgozásra rendkívül jól használható. Az RDFa szabványban meghatározott attribútumokkal kiegészített HTML kód, sokkal nagyobb jelentéstartalommal bír, mint az alap HTML dokumentum, így a szöveg sokkal jobb lesz további felhasználhatóság szempontjából. A Parsoid egy NodeJS-ben írt szoftver, így felhasználása Java nyelvű alkalmazásokban sokkal nehezebb, mint a korábbi parser megoldásoké, viszont kimenete a célnak leginkább megfelelő, így ez lett az ajánlott átalakító mechanizmus a feldolgozóláncban.

Végül a cikk feldolgozásának utolsó, harmadik fázisában eltárolódik a cikk a rendszer adatbázisában. Ebbe a fázisba akkor ér el a feldolgozás, ha nem volt újabb verziójú változat az adott cikkből még az adatbázisban. A verziók meghatározása a cikkek verziószáma alapján történik. Ha régebbi verziószámú cikk már volt az adatbázisban, akkor azt csak frissíteni kell, ellenkező esetben az adott cikknek még semmilyen változata nem létezik az adatbázisban, így azt újonnan kell beszúrni.

Ennél a komponensnél a megfigyelhetőséget naplózással és különféle metrikák lekérdezhetőségével oldottam meg. A `ParserActivator` indulásakor a `Logger` komponens (4.4. alfejezet), és a `Statistics` komponens (4.5. alfejezet) referenciáját is megszerzi.

3.2.1. Wikipédia dump feldolgozás

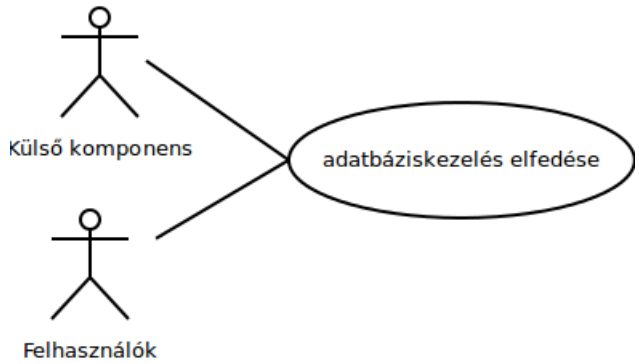
A feldolgozólánc indulásakor, lehetőség van az éppen frissülő cikkek on-the-fly feldolgozásán és adatbázisba való eltárolásán túl a már korábban a Wikipédián elérhető cikkek eltárolására is, melynek forrása a havonta frissülő Wikipédia állománya, amit XML formátumban tesznek elérhetővé.

A dump feldolgozása nagyon hasonlít az on-the-fly feldolgozáshoz, szintén egy `WikiObserver` példány irányítja a feldolgozást és egy `QueueManager`-t használ a cikkek ideiglenes eltárolásához. A hatalmas méretű (2013. októberében 44 GB) XML fájl, melyben a Wikipédián olvasható cikkek vannak eltárolva, a feldolgozás során először egy előfeldolgozáson megy keresztül, ahol gyorsan néhány statisztikai adatot gyűjt a program az adatbázismentésről. Ilyen statisztikai adat például a dump-ban lévő cikkek száma, így követhető a feldolgozás állapota a tényleges beolvasás során. Ekkora méretű fájlt csak valamilyen állapotgép alapú megoldással lehet feldolgozni, ilyen például *SAX Parser* (Simple API for XML). A parser a megszerzett cikket egy queue-ban tárolja el, ahonnan a `WikiObserver` kiszedi, annak szövegét átalakítja HTML formátumra és eltárolja az adatbázisban.

3.3. DatabaseConnector bundle

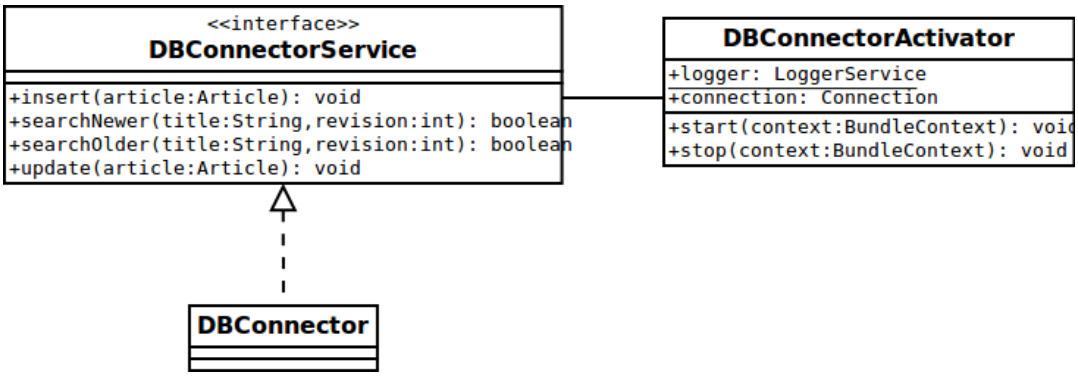
Ebben a komponensben lesz az adatbázisban eltárolva a Parser komponens által feldolgozott minden cikk. Feladatai közé tartoznak tehát az adatbáziskezeléssel kapcsolatos műveletek elfedése

más rétegek elől: cikkek eltárolása, frissítése, törlése és különböző szempontok alapján való keresés a cikkek között, azaz a CRUD műveletek (create, read, update, delete). A feldolgozóláncra alapuló további rétegek számára – melyet külső fejlesztők (a rendszer felhasználói) készítenek – ez a komponens egy csatlakozási pont, így kiajánlott szolgáltatását alacsonyabb és magasabb rétegben lévő komponensek is használhatják, az adatbázissal való kommunikáció teljesen el van fedve előlük.



3.7. ábra. A DBConnector használati esetei

A fenti use case-ek csak általánosan fogalmazzák meg a komponenssel kapcsolatos elvárásokat. Az egyes feldolgozóláncra alapuló komponensek további követelményeket is támaszthatnak a komponens iránt, például különböző szempontok alapján való keresés.



3.8. ábra. A DBConnector komponens osztálydiagramja

A komponens indulásakor (`DBConnectorActivator` osztály `.start()` metódusa), olyan állapotba hozza az adatbázist, hogy az később képessé váljon a fenti feladatokra, valamint egy OSGi szolgáltatást kell kiajánlani, amin keresztül a többi komponens kommunikálhat a DBConnector-ral.

3.4. Database

Az adatbázis komponens alapjául szolgáló adatbázis szoftver kiválasztását a WikiBot komponenshez hasonlóan egy rövidebb előkutatás előzte meg. Fő szempont volt, hogy nagy teljesítményű, gyors és lehetőleg az OSGi keretrendszerrel könnyen integrálható legyen a választott

adatbázis megoldás. Két különböző adatbázis lehetőséget vizsgáltam meg, és a tapasztalatok alapján választottam ki a legmegfelelőbbet.

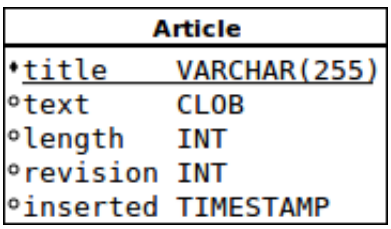
1. Prevayler: Egy nagyon egyszerű és gyors adatbázis, melynek lényege, hogy minden objektumot egyszerű Java objektumokként (POJO) tartsunk a memóriában, és a tranzakciókat naplózzuk, az adatbázis pedig a használó programba beágyazottan fut. A tárolandó objektumokra egyetlen megkötés, hogy implementálják a `Serializable` interfészt, és a műveleteket is szerializálható objektumok reprezentálják. A műveleteket a Prevayler naplózza, és időnként teljes mentést (*snapshot*) készít az adatbázisról, így egy rollback esetén a snapshot-ot visszaállítja és lefuttatja még a szükséges tranzakciókat.

   Felépítéséből adódóan sokkal gyorsabb mint bármelyik RDBMS, viszont OSGi környezetben a használata egyelőre nem megoldott. A probléma oka, hogy minden OSGi komponensnek saját classloader-e van, így osztályok dinamikus betöltése, például szerializációval nem lehetséges az OSGi programokban [17]. A megoldás lehetne új `BundleClassLoader` implementálása, azonban ez a Prevayler használatának egyszerűségét rontaná el, így inkább másik megoldást választottam.

2. H2DB: Az összegyűjtött tapasztalatok alapján arra jutottam, hogy OSGi környezetben kevésbé érdemes beágyazott adatbázist választani, ha több komponens között is meg kell oldani a kommunikációt, hiszen belső működésüket tekintve ezen adatbázisok szinte mindig használnak valamilyen szerializációs eljárást.

   A H2DB egy objektum relációs adatbázis, mely tud beágyazott és kliens-szerver módban is működni. A készítők teljesítménytesztjei alapján gyorsabb a legtöbb népszerű adatbáziskezelőnél (HSQLDB, Derby, MySQL, PostgreSQL), valamint tervezésekor megpróbálták a HSQLDB és a Derby adatbáziskezelők előnyeit is egyesíteni benne.

   A H2DB további előnyei még, hogy nem szükséges telepíteni, teljes egészében Java nyelven íródott és például Java nyelvű tárolt eljárásokkal is kiegészíthetjük funkcióit. Az eddig felsorolt pozitív tulajdonságok miatt esett a választásom a H2DB-re, bár igaz, hogy beágyazott módban OSGi-al nem használható effektíven, csak úgy, mint a Prevayler. A H2DB szerver módban is használható, így JDBC adatbázishozzáférési API-val lehet kapcsolódni hozzá. A szerver módban való futtatás előnye, hogy sokkal kisebb a csatolás az adatbázissal, így például a DBConnector komponensben, ha a JDBC API-t használjuk, az adatbázis könnyen lecserélhető a rendszerben másik adatbázis megoldásra.



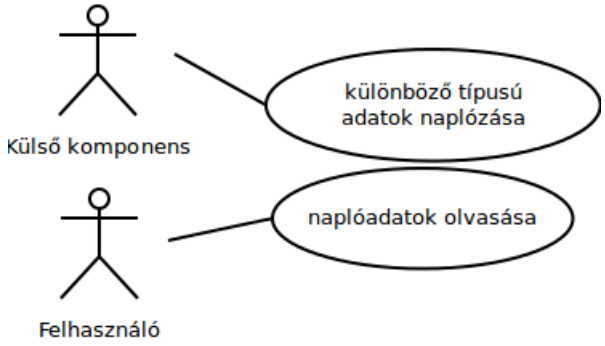
3.9. ábra. A tervezett adatbázis Article táblája

Az adatbázis kiválasztása után, az adatbázis tábláinak megtervezése következett. Az egyszerű specifikáció miatt egyed kapcsolat diagram rajzolására nincs szükség, hiszen egyetlen egyed van az adatbázisban, mely a cikk reprezentációja (`Article`). A Wikipediában minden cikket annak címe azonosítja (3.9. ábra), nem lehet két azonos című, de különböző cikk. Így elsődleges kulcs a cím (title) lesz, további szükséges attribútumok a szöveg (text), szöveg hossza (length), szöveg verziószáma (revision), illetve a beillesztés dátuma (inserted).

A rendszerben természetesen további adatbázisok, táblák is lehetnek, amelyeket a kutatómodulok hozhatnak létre. Ezek a kutatómodulok a rendszer által előfeldolgozott Article példányokat elemzik tovább. A DatabaseConnector komponens könnyen kiegészíthető, hogy a kutatómodulok igényeit teljesítse, azonban ez már nem az én feladatom volt.

3.5. Logger bundle

Ha a rendszer állapotait meg akarjuk figyelni, illetve szeretnénk, ha jelezné a hibákat, mindenképpen érdemes az alkalmazást felügyeletre tervezni. Az egyik ilyen lehetőség a naplózás, mely ennek a komponensnek a fő feladata (3.10. ábra).



3.10. ábra. A Logger komponens használati esetei

Első lépésben meg kell tervezni a rendszer felügyeleti modelljét, mely egy állapotgép formájában ábrázolható a rendszer fontosabb állapotaival. Az állapotgép átmenetei események, az átmenetek feltételei hibalehetőségek, a tranzakciók pedig naplózást jelentenek, a használt jelölés:

$$esemény(argumentum)[feltétel]/tranzakció \tag{3.1}$$

Naplózás során a rendszer eseményeit több kategóriába soroltam be súlyosság és következmények szempontjából, ezek láthatóak a 3.1. táblázatban.

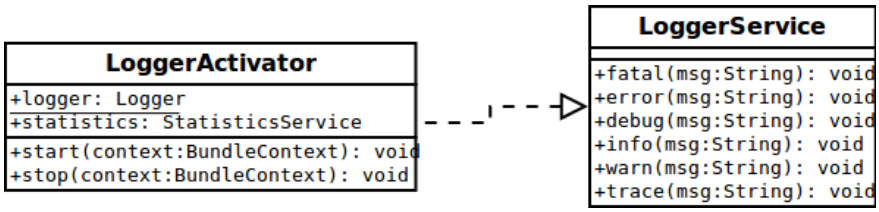
A fent meghatározott naplózási szinteknek megfelelő naplózási eljárásokat teljesítő komponens egy OSGi szolgáltatás lesz, mely képes a naplózást a követelményeknek megfelelően elvégezni (3.11. ábra).

3.6. Statistics bundle

Az alkalmazás felügyeletének másik lehetőségével meghatározott metrikák lekérdezésére és vizsgálatára van lehetőség. A többi komponens állapotait ennek a komponensnek a segítségével

naplózási szint	rövid leírás
FATAL	Súlyos hiba, amely a rendszer leállásához vezet.
ERROR	Súlyos hiba, azonban a rendszer működése nem feltétlenül áll meg.
WARN	Esetlegesen káros hatással bíró esemény.
INFO	Magasszintű információ a rendszer állapotáról.
DEBUG	Részletes információk a rendszer állapotáról, hibakereséshez használható.
TRACE	Legrészletesebb információk a rendszer állapotairól.

3.1. táblázat. A naplózás szintjei



3.11. ábra. A Logger komponens osztálydiagramja

teheti elérhetővé, így a rendszer által végzett feladatról készített statisztikák megfigyelésére is van lehetősége a felhasználóknak (3.12. ábra).

3.12. ábra. A Statistics komponens használati esetei

A feldolgozólánc működését átvizsgálva meghatároztam és kiválasztottam megfigyelés céljából a rendszer állapotának szempontjából fontos mérőszámokat (3.2. táblázat).

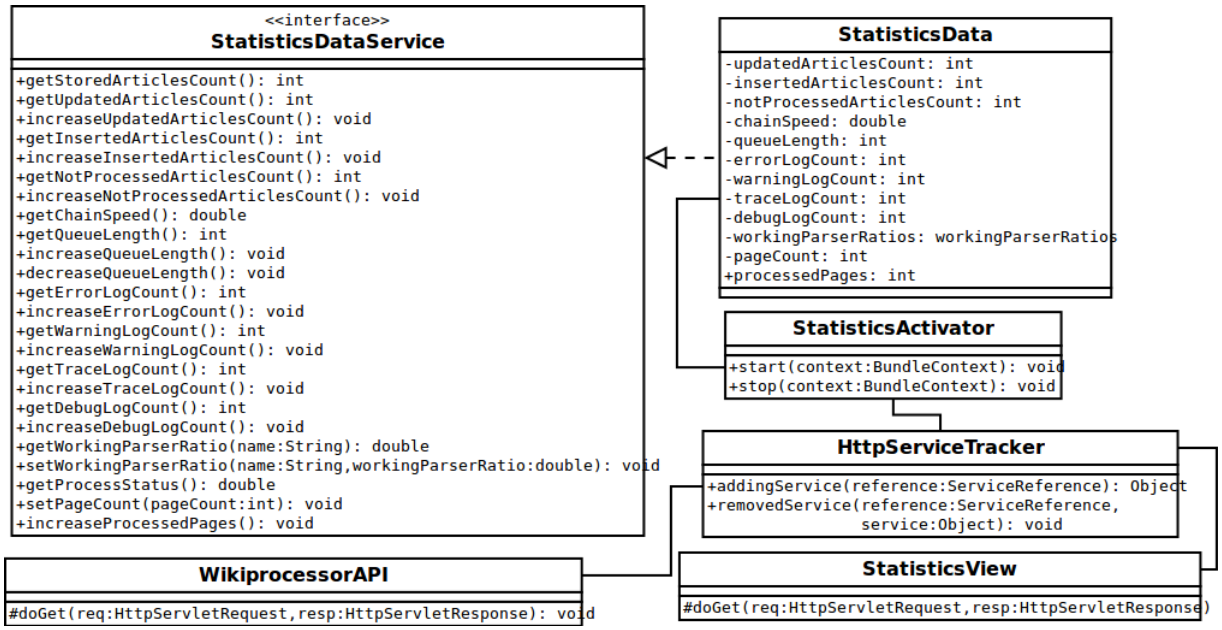
Az eddigi követelmények alapján úgy tűnhet, hogy a legjobb választás a JMX (Java Management Extensions) technológia lenne a feladatra, azonban több szempont alapján sem ezt választottam. Bár később a rendszer valamilyen szintű menedzselése szükségessé válhat, mégis OSGi környezetben nem biztos, hogy a JMX a legjobb megoldás. Egyrészt a JMX nem építhető be a már korábban említett OSGi tulajdonságok miatt a rendszerbe, így egy másik implementációt a MOSGi-t (Managed OSGi) kellene használni; másrészt a MOSGi egy sokkal bonyolultabb architektúrát használ és a felhasználók számára sem annyira egyszerű a kezelése.

metrika neve	**metrika leírása**	**metrika mértékegysége**	**metrika számításának módja**
eltárolt cikkek száma	ennyi cikk van az adatbázisban	darab	frissített cikkek száma + újonnan beillesztett cikkek száma
frissített cikkek száma	korábban már az adatbázisban szereplő, frissített cikkek száma	darab	ha kisebb verziószámú azonos című cikk szerepel az adatbázisban, akkor eggyel nő a száma
újonnan beillesztett cikkek száma	korábban még az adatbázisban nem szereplő cikkek száma	darab	ha még nem volt az adatbázisban az adott című cikk akkor eggyel nő a száma
nem feldolgozott cikkek száma	ennyi cikket nem dolgoztunk fel, mert újabb változat van az adatbázisban	darab	minden esetben mikor újabb verziójú cikket dolgoz fel a száma nő eggyel
hibák száma	FATAL, ERROR, WARN, DEBUG, INFO, TRACE üzenetek száma a naplóban	darab	log üzenet írása esetén a statisztikák száma is nő
queue hossza	QueueManager által karbantartott queue-ban lévő elemek száma	darab	QueueManager-be íráskor értéke nő eggyel, elem kivételekor csökken eggyel
feldolgozás sebessége	ennyi cikket dolgoz fel 1 perc alatt a feldolgozólánc	cikk/perc	1 percenként elmentett cikkek számából kivonja az előző elmentett értéket
dolgozóparserek száma	adott nevű observerben ennyi parser dolgozik	százalék	dolgozó parserek száma / az összes elérhető parserek számával
dump feldolgozás állapota	ekkora százaléka lett beolvasva a dump-nak	százalék	beolvasott cikkek száma / összes cikk a dump-ban

3.2. táblázat. A Statistics komponens által megfigyelhető metrikák

A fent felsorolt indokok alapján döntöttem saját menedzselő komponens fejlesztése mellett, mely egy sokkal egyszerűbb architektúrát követ, és nagyon könnyen kiegészíthető. A menedzser komponens egy OSGi szolgáltatásból áll (`StatisticsDataService` a 3.13. ábrán), melybe a többi komponens rögzítheti az állapotait, illetve két servlet segítségével tud kapcsolatba lépni a felhasználókkal, melyek közül az egyik JSON formátumban elérhetővé teszi a statisztikai adatokat (`WikiprocessorAPI`), a másik pedig egy HTML oldalt készít a statisztikai adatokkal (`StatisticsView`). Az elkészített HTML oldalon grafikonok készülnek el dinamikusan Javascript + Ajax segítségével, így az elkészült weboldalon friss statisztikai adatokat lehet megfigyelni folyamatosan a rendszer állapotáról és a futó folyamatokról.

A Statistics komponens többféleképpen is megszerezheti a feldolgozólánc többi komponensének állapotait. Az egyik lehetőség, hogy saját maga gyűjti az adatokat az eseményekről, a különböző komponensek feliratkoznak a kiajánlott statisztika készítő szolgáltatásra és így frissítik



3.13. ábra. A Statistics komponens osztálydiagramja

a metrikákra vonatkozó adatokat. Emiatt ezen komponensek (például a Parser) függeni fognak a Statistics komponenstől. Ekkor, ha a parserben lévő queue-ról is szeretnénk adatokat, az egyik lehetőség, hogy a queue minden módosulásánál tájékoztatja a statisztika készítő komponenst a megváltozásról, ez azonban overhead-del jár a queue számára. A másik megoldás, hogy a statisztika készítő komponens kéri le a queue hosszát, ha szüksége van rá, így azonban a Statistics komponens is függeni fog a Parser-től. Emiatt, ha a második lehetőséget választanánk, ami nem jár akkora overhead-del, más a Parser-nél alacsonyabb szintén lévő komponensek nem használhatnák a Statistics komponenst, hiszen a függőségek alapján az OSGi keretrendszer nem tudna előállítani olyan komponens indítási sorrendet, amelyben minden indítási függőség kielégülne (függőségi hurok alakulna ki). A megmaradt választási lehetőség tehát az, ha minden komponens folyamatosan tájékoztatja a megfigyelhető állapotairól a Statistics komponenst.

3.7. Research bundle-ök

Ezen kutató bundle-ök tervezése nem tartozott a feladataim közé. Működésüket tekintve sokféle céljuk lehet, például mondattani elemzés, szemantikus annotálás, szó együttes előfordulás analízis (ko-okkurencia), vagy egyéb természetes nyelvi feldolgozás témakörébe eső feladat.

A kutatókomponensek dolgozhatnak direkt módon a H2DB adatbázison, vagy használhatják a kiajánlott OSGi szolgáltatásokat és akkor értesülhetnek az éppen frissülő cikkekről, megkaphatják az elkészült Article példányokat. A cikk példányokkal tetszőleges műveleteket hajthatnak végre, de fontos, hogy ne tartsák fel a rendszer működését.

A rendszer funkcióinak tesztelésekor (5. fejezet) egy MTA SZTAKI által készített teszt kutatómodult használtam, mely a cikkekben szereplő link és kategóriahivatkozásokat és a hozzájuk köthető szöveges megjelenési formákat elemezte. További kutatómodulok fejlesztése a későbbiekben várható még, ezért a feldolgozólánc többi elemét is fel kell erre készíteni.

4. fejezet

Implementáció

A komponensek fejlesztésekor Eclipse fejlesztő környezetet használtam, verziókezeléshez pedig a Git szoftvert. Az alapértelmezett Eclipse IDE azonban nem tökéletes eszköz az OSGi alapú fejlesztéshez, ezért a Bndtools [4] OSGi fejlesztő keretrendszert Eclipse kiegészítő modulként feltelepítve kezdtem neki a komponensek fejlesztésének.

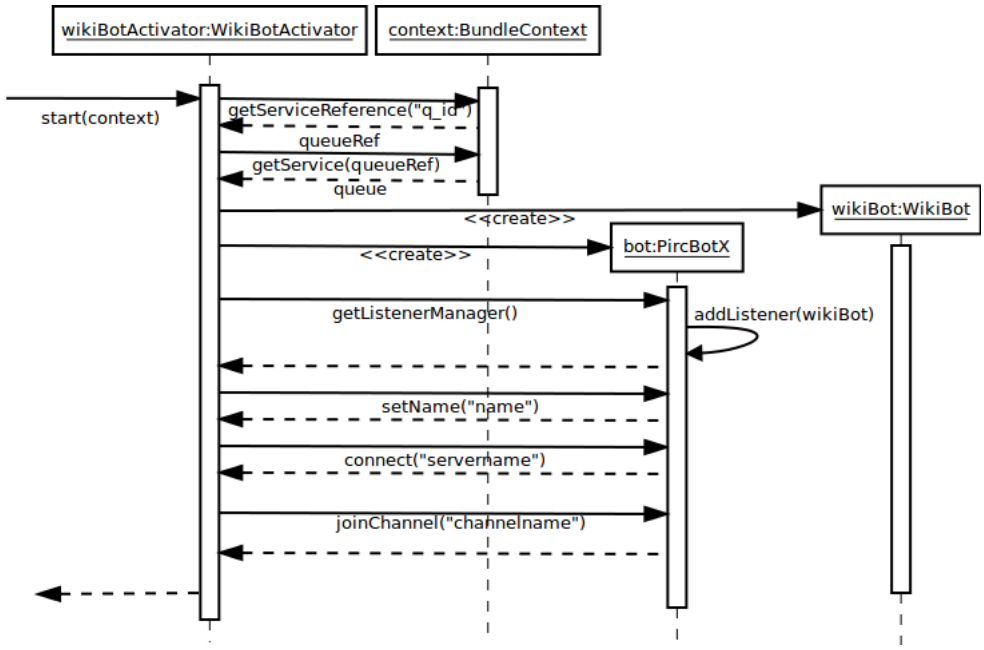
4.1. WikiBot bundle

A WikiBot komponens architektúráját teljes mértékben a fejlesztéskor használt IRC Bot készítő keretrendszerek határozták meg. A kezdeti változatban a PircBot [14] keretrendszert használtam, azonban hosszan tartó használata alatt kiderült, hogy a rendszernek több hibája is van. Hosszabb ideig tartó futáskor ismeretlen eredetű hibák kerültek elő a PircBot könyvtárban, melyek okait a rossz tervezés miatt felderíteni sem volt lehetséges. A keretrendszert Java 1.1 verzióra tervezték, nem használja ki az azóta a nyelvben megjelent újdonságokat, így teljesen elavultnak számít technológiailag, hiába fejlesztik azóta folyamatosan. A hibák nagyon sok esetben teljesen elnyelődnek a rendszerben, így sok esetben lehetetlen az alkalmazás hibáinak megfejtése. Egyes metódusok, melyek öröklésekor felüldefiniálva jól használhatóak lennének, private vagy final hozzáférés-vezérlési kulcsszavakkal vannak ellátva. További rossz tervezési döntés a *God Object* antipattern-nel való visszaélés.

Ezen hátrányok miatt egy idő után az újabb, kevésbé elterjedt, de jobban karbantartott verziójára tértem át a PircBot-nak, mely a PircBotX [5] nevet viseli. Ez a váltás azonban csak kis mértékben befolyásolta a fejlesztés menetét, hiszen mindkét keretrendszernél egy esemény alapú architektúra kialakítása szükséges, melyben különféle események esetén implementálni kell a rendszer viselkedését.

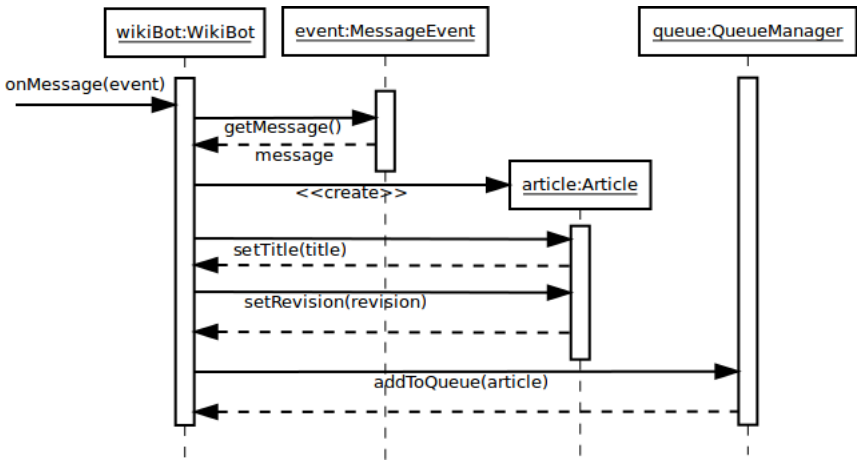
A bundle-ök implementációja során a komponensek függőségeinek, melyek külső könyvtárakban voltak, szintén el kellett készíteni a bundle változatait, hogy azok hiánya ne jelentkezzen hibaként az OSGi függőségkezelőjében. Ez a lépés mind a PircBot, mind a PircBotX esetén a külső könyvtárak OSGi komponens formára való alakítását jelentette, melyet a Bndtools nevű eszközzel hajtottam végre. Ebben a lépésben gyakorlatilag a kész JAR kiterjesztésű library-t kellett egy OSGi specifikáció szerint meghatározott manifest állománnyal újracsomagolni.

A komponens működése látható a 4.1. ábrán: először el kell kérni a `QueueManager` referen-



4.1. ábra. WikiBot komponens indulása

ciáját, majd egy PircBotX példányhoz hozzá kell adni egy `Listener`-t, amely implementálja az eseménykezelőt függvényeket. Ezután meg kell adni az IRCBot nevét, csatlakozni kell egy szerverhez, végül be kell lépni egy IRC csatornába. Ettől a ponttól kezdve, ha egy üzenet érkezik a csatornába (4.2. ábra), a `WikiBot` kiszedi a szükséges információkat reguláris kifejezésekkel és elhelyezi a `QueueManager`-ben.



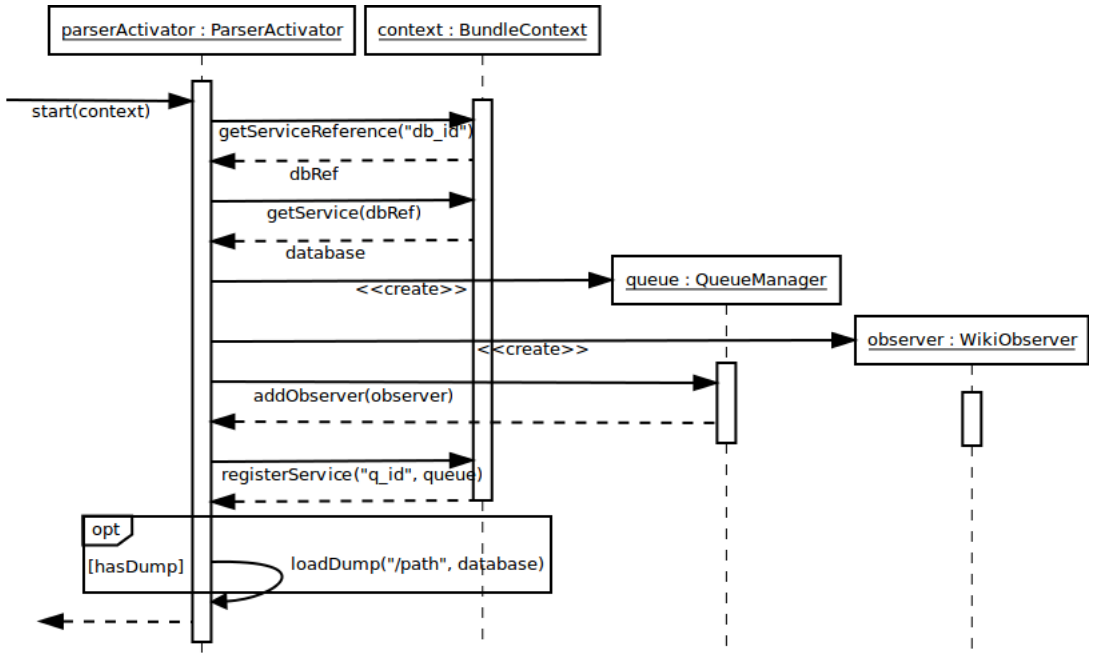
4.2. ábra. Eseménykezelés a WikiBot-ban

4.2. Parser bundle

Az összes komponens közül a Parser felépítése a legösszetettebb, többszálú működése miatt egyszerre történik sok minden, ezért a működésen keresztül fogom bemutatni, hogy teljesíti a Parser bundle a tervezésben meghatározottakat.

A komponens indulásakor először megszerzi a kiajánlott adatbázis szolgáltatás referenciáját, a

feldolgozás után itt fogja eltárolni a feldolgozott cikkeket azok adataival. Létrehoz ezenkívül egy `QueueManager`-t, beállít (`.addObserver()` metódus) számára egy újonnan létrehozott megfigyelőt (`WikiObserver` osztály), és beregisztrálja a queue-t, mint OSGi szolgáltatást. Végül ha elérhető Wikipédia adatbázis mentés (dump), akkor megkezdi annak feldolgozását is. Az indulás folyamata figyelhető meg a 4.3. ábrán.



4.3. ábra. A Parser komponens indulása

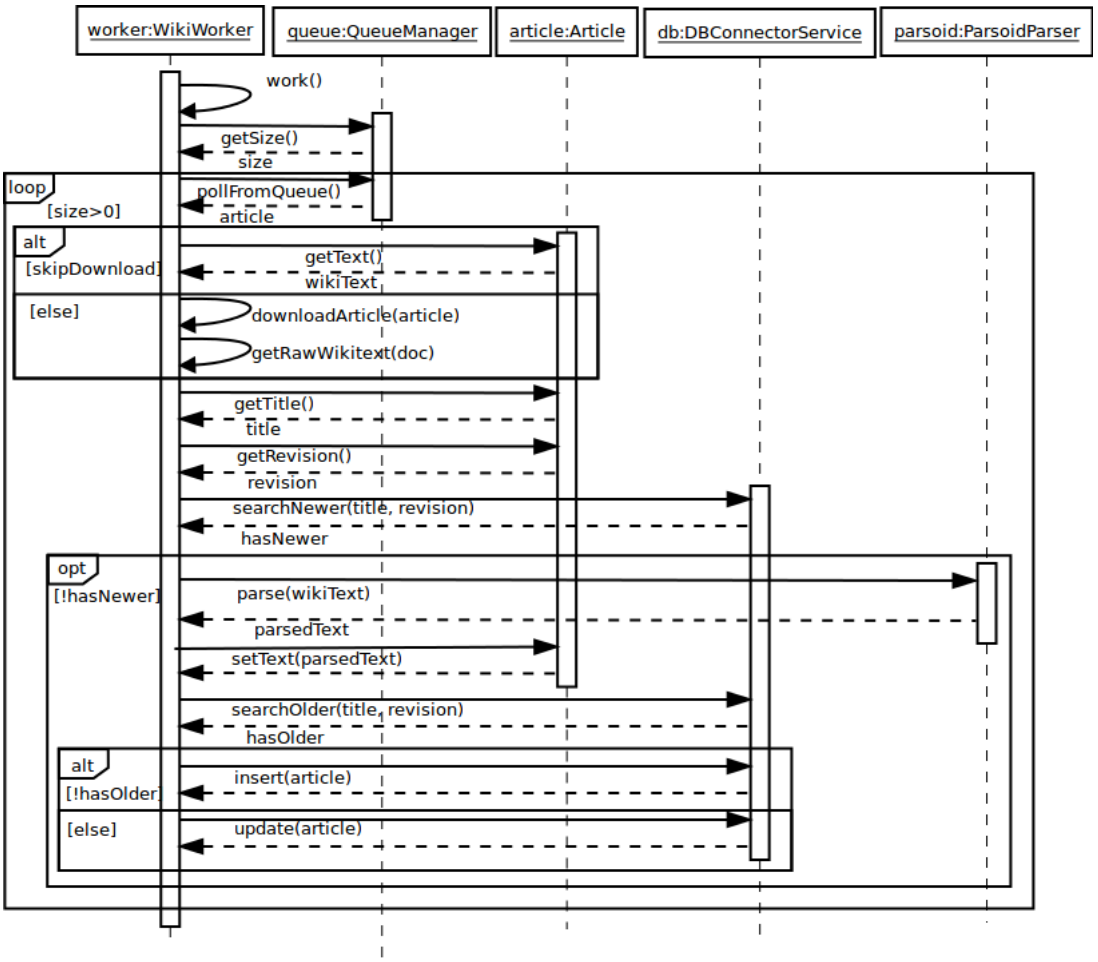
A cikkek feldolgozása külön szálakon történik meg, a minél gyorsabb működés érdekében. Minden cikkhez külön Thread objektum példányosítása rendkívül nagy overhead-del járna, ezért a 3 fejezetben már említett ThreadPool mintát használtam. Ez a minta a Thread objektumok újrahasznosításával működik, így nem szükséges mindig új szálat példányosítani. Használata a következő 4.1. kódrészletben figyelhető meg, futtatáskor a szálak nevéből látszik, hogy mindig újrahasználjuk a kezdeti 10 darab szálat:

4.1. Listing. Példa a ThreadPool használatára

```
1 public class ThreadPoolExample {
2     public static void main(String args[]){
3         ExecutorService service = Executors.newFixedThreadPool
              (10);
4         for (int i = 0; i < 100; i++){
5             service.execute(new Runnable() {
6               public void run() {
7                   System.out.println(Thread.currentThread().
                        getName());
8                try {
9                Thread.sleep(1000);
```

```
10              } catch (InterruptedException e) {
11                e.printStackTrace();
12              }
13                 }
14               });
15          }
16      }
17  }
```

A Parser működése az eddigieket felhasználva a következő:



4.4. ábra. A WikiWorker szál működése

1. Ha egy új cikk kerül a `QueueManager`-be, akkor az Observer mintán keresztül a `WikiObserver` arról értesül.

2. A `WikiObserver` egy új `WikiWorker` szálat rendel a ThreadPool-ból a feladathoz, mely megkezdi a cikk előfeldolgozását (4.4. ábrán `.work()` metódus).

3. A szál addig fut, amíg van feldolgozatlan cikk a queue-ban. A dump feldolgozásban fut a szál, akkor nem kell letölteni a cikkek szövegét, mert azokat ilyenkor a dump-ból már

megszereztük. Ellenkező esetben le kell tölteni a hivatalos Wikipédia API-n keresztül a cikket XML formátumban, és ki kell szedni a Wikitext formátumban lévő szövegét.

4. Ha van az adatbázisban már (`.searchNewer()` metódus) újabb verzió az adott cikkből, akkor nem kell tenni semmit, hiszen valószínűleg egy dump feldolgozásban vagyunk és korábban már on-the-fly megszereztük a cikk egy újabb verzióját.

5. Ha nincs újabb verziójú cikk, akkor a Wikitext formátumról HTML (RDFa) formátumra kell alakítani a szöveget (`.parse()` metódus).

6. Ha az adatbázisban van már régebbi verziójú cikk (`.searchOlder()` metódus) az adott cikkből, akkor valószínűleg on-the-fly feldolgozásban vagyunk és korábban már megszereztük a cikk egy korábbi verzióját. Ekkor az adatbázisban csak frissíteni kell a cikk adatait, máskülönben újonnan kell beilleszteni a cikket.

Az előfeldolgozáskor kapott eredmények meghatározzák a feldolgozólánc használhatóságát, így fontos, hogy a megszerzett cikkek Wikitext formátumú szövegét milyen formában tároljuk. A 3. fejezetben láttuk, hogy a feldolgozáskor három cserélhető parser használható a rendszerben, melyek különböző képességekkel bírnak. Ezen parserek a Sztakipedia parser, DumbRegexWikipediaParser és a Parsoid parser. A Sztakipedia parser-nél szükség volt az MTA SZTAKI által készített Java nyelvű könyvtár OSGi komponenssé alakítására.

További felhasználás szempontjából a Parsoid készíti a legjobb kimenetet, azonban egyben a leglassabb is. Ezen lassúság kiküszöbölése miatt különösen jó választás a ThreadPool minta, mert a Parsoid szoftver működése jelentősen függ a bemeneti állományoktól. A Parsoid egy NodeJS-ben írt modul, így a feldolgozólánchoz való illesztését egy szerver oldali NodeJS alkalmazással oldottam meg, mely a HTTP POST kérések törzsében küldött Wikitext szöveget HTML / RDFa formátumra tudja alakítani és azt adja vissza a klienseknek. Ezzel a megoldással a feldolgozólánc HTTP-n keresztül kommunikálhat a Parsoid példányokkal, melyek akár külön szervereken is futhatnak (Parsoids komponens a 3.1. ábrán).

A dump beolvasása szintén ebben a komponensben van implementálva. A Parser indulásakor, ha van elérhető dump file, akkor megkezdi a dump beolvasását. Ezt a folyamatot a `DumpLoader` osztály vezérli, két nagyobb fázisban történik meg a dump beolvasása. A Wikipédia dump-hoz hasonló méretű, óriás XML adatok olvasása semmiképpen nem történhet olyan eszközzel, amiben a teljes adat reprezentációját egyszerre kell tárolni, mint például a DOM Parser. Az én választásom egy állapotgép alapú megoldás lett, a SAX (Simple API for XML) Parser. A feldolgozás első fázisában a tényleges folyamat nyomonkövethetősége érdekében a `DumpPreprocessor` megszámolja a dump-ban lévő cikkek (page csomópontok) számát. A `DumpReader` pedig végigolvassa a dump-ot, és ha beolvasott egy teljes cikket, azt beilleszti egy QueueManager-be. Innentől a feldolgozás megegyezik a főszálon történő folyamattal.

4.3. DatabaseConnector bundle

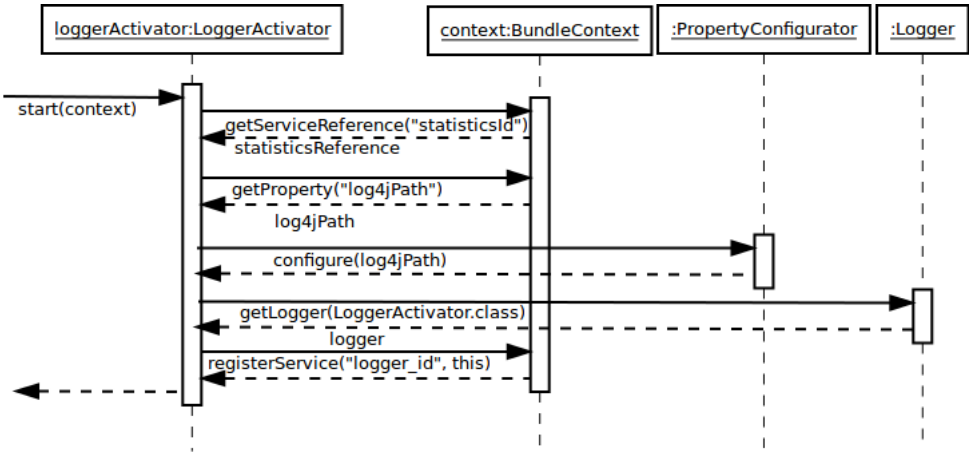
Ez a komponens a tervezésnek megfelelően elfedi az adatbáziskezelést a többi komponensek elől úgy, hogy felhasználja a kiválasztott H2DB adatbázis JDBC API-ját. A DBConnector kompo-

nens indulásakor először az adatbázis állapotát kell ellenőrizni és szükség szerint olyan állapotba kell hozni, hogy a feldolgozólánc dolgozni tudjon vele. Első lépésben csatlakozni kell a H2DB-hez, majd létre kell hozni hozzá egy adatbázisfelhasználót, egy adatbázist, valamint a szükséges táblákat. Ezzel az adatbázis készen áll a feldolgozólánccal való kommunikációra.

A feldolgozásra kerülő rendkívül nagy adatmennyiség miatt érdemes megfontolni indexek létrehozását is az adatbázisban az Articles táblán. Egy ilyen nagy teljesítményű rendszernél számolni kell az indexhasználat hátrányaival is, ugyanis ha több az írások száma, mint az olvasások száma, az indexek készítése jelentős overhead-et jelenthet az adatbázis és a diszk alrendszer számára. Az alkalmazás fő célja, hogy adatokat lehessen kiolvasni belőle, ezért elsődlegesen nem az írások, hanem az olvasások sebességét kell megnövelni. Másrészt az alkalmazás logikája is megköveteli, hogy az olvasás gyors legyen, hiszen ahhoz, hogy egy cikket beillesszünk mindenképpen meg kell nézni, hogy az adott cikk szerepel-e már az adatbázisban. A legtöbbet használt attribútum olvasáskor a cikk címe, így azon hoztam létre indexeket a H2DB-ben.

4.4. Logger bundle

A tervezés részben meghatározott funkcióknak megfelelő módon implementáltam a komponenst. Nem kezdtem saját naplózási rendszer kidolgozásába, hanem a már kiforott *Apache log4j* megoldást építettem be a komponensbe. Az Apache log4j előnyei, hogy a tervezésénél ügyeltek arra, hogy ne legyen túl nagy hatással az instrumentált kód működésére, és ne kelljen az alkalmazást újrafordítani a naplózás beállításainak módosításához. A log4j beállításait egy konfigurációs fájlban tárolja, ahol széleskörű beállítási lehetőségek használatára van mód.



4.5. ábra. A Logger komponens indulása

OSGi környezetben nem ilyen egyszerű az Apache log4j használata. Alapértelmezett esetben a log4j konfigurációs fájlját (`log4j.properties`) az alkalmazás gyökérkönyvtárába kell helyezni, OSGi környezetben nyilvánvaló módon ez nem működik. A legegyszerűbb megoldás *fragment bundle* készítése lenne, melynek nincs `BundleActivator` osztálya és megosztja classloaderét egy szülő komponenssel. Ennek azonban nagy hátránya van és az Apache log4j egyik előnyét veszítenénk el, mert fragment bundle készítésénél a konfigurációs fájlt a forráskóddal együtt be kellene csomagolni a komponensbe. Így nem lesz konfigurálható a naplózás formája,

szintjei, be lesz égetve egy bundle-be minden beállítás.

A fenti hátrányokat elkerülhetjük, ha *runtime property*-ként megadjuk a helyét a merevlemezen tárolt log4j.properties fájlnak a Felix konfigurációjában. A naplózás beállítását egy szöveges fájlban leírhatjuk és azt a rendszerben bárhol elhelyezhetjük. Az Apache log4j beállítását és OSGi szolgáltatásként való kiajánlását a komponens indulásakor kell megtennünk (4.5. ábra).

4.5. Statistics bundle

A Statistics komponens implementálásakor a tervezést követően két servletet és egy adatokat gyüjtő és tároló megoldást kellett létrehozni. A tároló felépítése nagyon egyszerű, a tervezéskor meghatározott metrikákat reprezentáló adatokat tárolja attribútumokban, és implementálja egy kiajánlott OSGi szolgáltatás által meghatározott metrikák módosítására és lekérdezésére szolgáló metódusokat.

A servletek készítése egy kicsit bonyolultabb volt, mert ahhoz, hogy a servlet HTTP-n keresztül elérhető legyen be kell regisztrálni egy servlet container-be. A JEE/J2EE fejlesztés során megszokott `web.xml` fájlon keresztüli konfigurációt nem használhatjuk, OSGi környezetben, ezért az `org.osgi.util.tracker.ServiceTracker` osztályban kell beregisztrálnunk servletünket. Ehhez az előbbi osztályból kellett leszármaztatni egy osztályt (`HttpServiceTracker` a 3.13. ábrán) és felülírni az `.addingService()` és `.removedService()` metódusait, ahol be kellett regisztrálni a servlet osztályokat.

4.2. Listing. A WikiprocessorAPI által szolgáltatott adatok

```
1 {"articlesData":{"eltárolt cikkek száma":2785,"frissített cikkek
      száma":877,"újonnan beillesztett cikkek száma":1908,"nem
    feldolgozott cikkek száma":13},
2 "logsData":{"error üzenetek száma":3,"warning üzenetek száma"
    :11,"debug üzenetek száma":4},
3 "queuelength":6013,"mainworkingparsoids":0.5714285714285714,"
    dumpworkingparsoids":0.8,"dumpprocess":0.11469880217730487}
```

A `WikiprocessorAPI` servlet a metrikák alapján JSON formátumban (4.2) teszi közzé a feldolgozólánc adatait, így ez a servlet gyakorlatilag egy webes API-nak tekinthető. A `StatisticsView` egy statikus HTML oldalt szolgál ki, ahol az előbbi API-t felhasználva JavaScript segítségével diagramokat rajzol ki a servlet. A létrehozott diagramok a metrikákat alakítják a felhasználók számára sokkal jobban megfigyelhető formába.

5. fejezet

Tesztelés és mérések

5.1. A teszteléshez használt eszközök

A feldolgozólánc teszteléséhez elő kellett állítani először a komponensek futtatható változatát. A komponensek implementációjakor használt Bndtools eszköz ebben is nagy segítségemre volt, mert az alkalmazás futtatásakor a Bndtools automatikusan előállította a telepíthető csomagokat. Az egyes komponensek könyvtári függőségeit, amelyek nem részei az alap OSGi futtató környezetnek (tipikusan ezek a harmadik féltől származó könyvtárak) magamnak kellett előállítanom szintén a Bndtools segítségével.

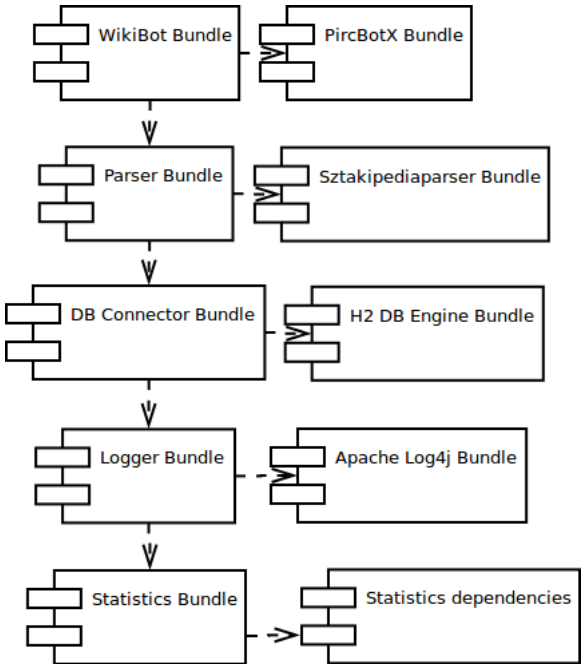
A fordításnál előállított JAR formátumú bundle-ök módosítás nélkül telepíthetőek valamilyen OSGi alapú rendszert futtatni képes környezetbe (*bundle repository*-ba). Fejlesztés során a gyorsabb tesztelhetőség miatt az Eclipse saját OSGi implementációs megoldását az Equinox-ot használtam. Egy stabil változat elkészítése után pedig, bizonyos szinten automatizáltan, szkriptek segítségével telepítettem az alkalmazást a SZTAKI Cloud-ban létrehozott virtuális gépre, melyet az alkalmazás futtatására készítettem elő tesztelés céljából.

A telepítési környezetben már a könnyen kezelhető Apache Felix [12] OSGi implementációját használtam, mert annak WebConsole eszközével könnyedén kezelhetőek a komponensek menedzselése, valamint parancssoros hozzáférés is lehetséges, ahol interaktív módon menedzselhetőek a komponensek, sőt még egy egyszerűbb Bash szintaktikáját utánzó szkriptelési lehetőséget is használhatnak a fejlesztők. Teszteléskor a virtuális gépre SSH-n belépve, az Apache Felix konzolos felületét használva telepítettem az elkészített bundle-öket és függőségeiket.

Telepítéshez, illetve a feldolgozólánc komponenseinek frissítéséhez az Apache Felix egy beépített megoldását az Apache Felix Gogo-t használtam, mely egy egységes, szabványos shell felületet határoz meg az OSGi alapú környezetek számára. Amint már említettem az Apache Felix Gogo-val a Unix rendszerekben ismert Bash szintaktikához hasonló parancsokat lehet használni. Ezeket a parancsokat akár szkript formájában is futtathatjuk, ez a Gogo Shell szkript, másnéven *gosh* szkript.

A csomagok függőségeit ábrázolja a 5.1. ábra. A függőségi fa leveleiben található bundle-öket kell először, majd a fában a gyökér felé visszafelé haladva kell a többi bundle-t telepíteni.

A bundle-ök menedzselését a konzolos Apache Felix Gogo Shell-en kívül, egy webes felületen Apache Felix Web Console segítségével is lehet végezni. Segítségével a rendszer állapotát, tulaj-



5.1. ábra. Komponensek függőségei

donságait lehet beállítani és megfigyelni, valamint a bundle-ök részletes tulajdonságait, kapcsolatait lehet felderíteni, illetve beállításait módosítani. A webes felületről a bundle-ök telepítésével, indításával kapcsolatos funkciók szintén elérhetőek.

5.2. A feldolgozóláncra épülő kutatómodul

A tesztelés során egy lehetséges felhasználási módot próbáltam ki, mikor a feldolgozólánchoz egy MTA SZTAKI által készített kutatómodult illesztettem. A modul segítségével gépi tanulás, természetes nyelvi feldolgozás témakörökhöz használható program készíthető.

A kutatómodul a feldolgozólánc által készített adatbázisból szerzi meg a bemenő adatokat, ehhez a DB Connector komponens kiegészítése volt szükséges, az OSGi szolgáltatást a lekérdezésekhez szükséges metódussal kellett ellátni.

Az kutatómodul az *WikiZaba* nevet viseli, és az *Annotare* könyvtár képességeit valósítja meg. Célja, hogy XML szabvány szerint jól formázott HTML dokumentumokat nyelvileg feldolgozhatóvá és annotálhatóvá tegyen, így a feldolgozóláncban keletkező Article példányokat fel tudja dolgozni. Az Annotare könyvtárnak három fő funkciója van:

1. Dokumentum beolvasása SAX Parser segítségével, illetve annak átalakítása. Az átalakítás után keletkezik egyrészt egy plain text dokumentum, amely átadható egy nyelvi feldolgozó modulnak, másrészt egy vagy több lista, amely az XML elemeket, mint annotációkat tartalmazza.

   Az Annotare az *UIMA* terminológiáját használja (de az UIMA kódjait vagy könyvtárait egyáltalán nem, azoktól független). Ebben a terminológiában a szóban forgó plain text dokumentum a *Subject of Annotation* (SOFA), az annotációk pedig *FeatureStructure*-ok reprezentálják. A SOFA alapértelmezett konfigurációban az XML elemek tartalmának konka-

tenálásából áll össze, de az XML tag-ek elhagyásával. A FeatureStructure-ok egy begin és end pozícióval rendelkeznek, amely pozíció azt jelöli ki, hogy a SOFA-ban hol kezdődik és végződik egy adott annotáció. A feldolgozáskor alapesetben minden FeatureStructure egy XML elemnek felel meg, így a begin és end azt mutatja, hogy hol kezdődik és hol végződik az adott elem a SOFA stringhez képest. Ez a szám nem azonos azzal a karakterpozícióval, ahol az eredeti dokumentumban az adott XML elem kezdődik, hiszen a SOFA stringbe az XML tag-ek által elfoglalt karakterek nem számítanak bele.

5.1. Listing. Példa az XML SOFA és FeatureStructure kapcsolatára

```
1 XML: <a>x<b>y</b><c>z</c></a>
2 SOFA: xyz
3 FS-ek: (a) 0..11 ; (b) 4..8; (c) 9..11
```

A fentieken túl egyedi konfigurációval megoldható, hogy az Annotare több, közvetlenül egymás után következő pozíciójú, és azonos típusú FeatureStructure-t egybe olvasson.

2. A második fontos funkció a FeatureStructure lista kiegészítése újabb annotációkkal. Ezek rendszerint egy NLP rendszer (például OpenNLP) kimenetéből származnak. Meg kell adni hozzájuk, hogy milyen XML taggel legyen reprezentálva és hogy mi a begin és end koordinátájuk.

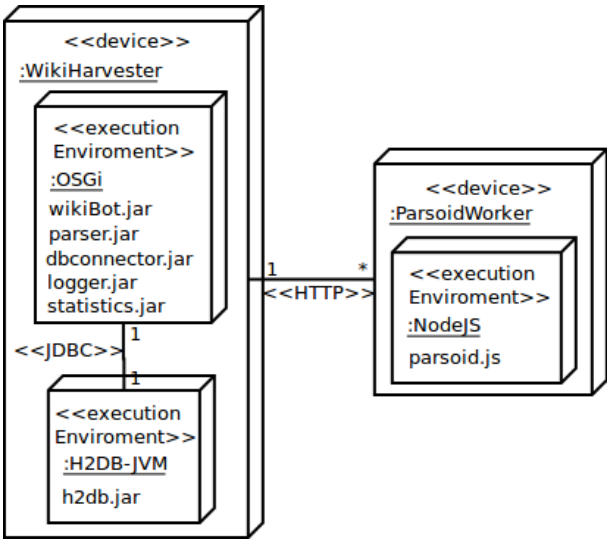
3. Végül az Annotare egy FeatureStructure listát képes XML formátumba szerializálni. Mivel a FeatureStructure-ök átlapolódhatnak, nem kell szigorú tartalmazási viszonyban lenniük (szemben az XML elemekkel), ennek a feladatnak fontos része, hogy szükség esetén a rendszer egy FeatureStructure-t több XML elemre daraboljon fel. Azt, hogy melyik elem lesz több részre bontva az elemek prioritása határozza meg, amely egy konfigurálható paraméter a FeatureStructure-ok esetében.

Az Annotare célja, hogy XML-ként értelmezhető dokumenumokban szereplő szövegeket NLP rendszerekkel feldolgohatóvá tegyen. Tipikusan HTML dokumentumoknál (például a Parsoid parser kimenete) vagy struktúrált dokumentumokat leíró XML fájlok esetén hasznos az alkalmazása. Ezen dokumentumok esetén az a probléma, hogy az XML tag-ek jelenléte teljesen elrontja az NLP rendszerek teljesítményét. Ha ezeket egyszerűen kitöröljük, akkor elveszítünk fontos információkat, ráadásul meg kell oldanunk azt is, hogy az egyik XML elem végén található szó ne olvadjon össze a másik XML elem elején találhatóval. Az Annotare fent részletezett funkciói ebben segítenek. Segítségükkel tetszés szerint vehetünk ki szövegrészeket a dokumentumunkból, feldolgozhatjuk azt, majd az eredményt visszaillesztethetjük a strukturált dokumentumunkba.

Az Annotare segítségével tehát a WikiZaba kigyűjti a cikk és kategória hivatkozásokat, majd megállapítja ezek szövegkörnyezetét (azt a bekezdést, amiben szerepelnek), valamint a hivatkozások megjelenési formáját, azaz azt a szövegrészt, amely hivatkozássá alakul.

5.3. Mérési eredmények

A mérési összeállítás a 5.2. ábrán figyelhető meg. A használt virtuális gépek OpenNebula sablonjának legfontosabb paraméterei a 5.2. kódrészletben olvashatóak. A virtuális gépek egy 16 ma-



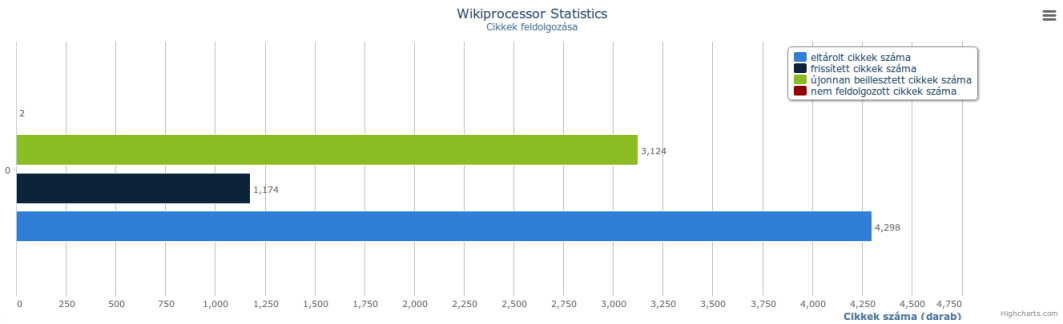
5.2. ábra. A mérési összeállítás deployment diagramja

gos AMD 7272 processzorral rendelkező gépen futottak, storage szerverként egy DELL MD3600 rendszer üzemelt. A WikiHarvester gép 4 processzort használ 16 virtuális CPU maggal és 10 Gb RAM érhető el számára. Ezen a gépen fut az OSGi keretrendszer benne a fejlesztett komponensekkel és azok függőségeivel, illetve a H2 adatbázis is itt fut külön folyamatban saját JVM-mel (a H2 adatbázissal JDBC kapcsolaton keresztül történik a kommunikáció). A WikiHarvester gép 30 darab ParsoidWorker virtuális gépet tud használni a cikkek átalakítására a Parser komponensben. A ParsoidWorker gépek 1 darab 2 magos processzort használnak fejenként, 512 Mb RAM memóriával; mindegyikre fel van telepítve a NodeJS, ahol a saját szerveroldali Parsoid megoldás fut.

5.2. Listing. Részlet a használt VM-ek sablonjából

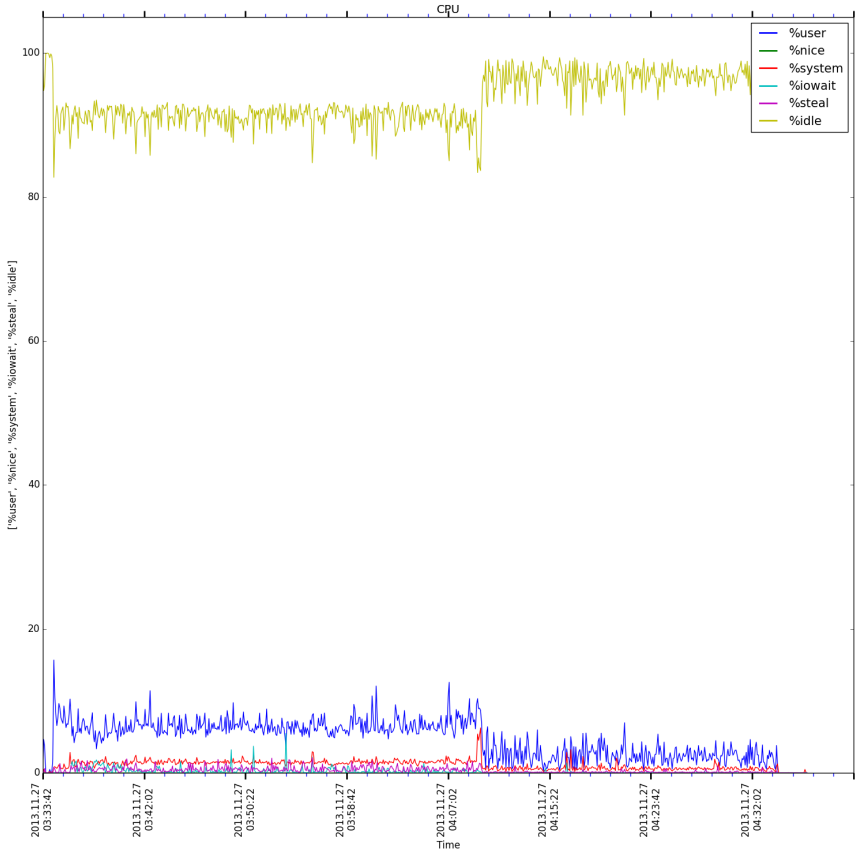
```
1 NAME   = [APP]WikiHarvester
2 CPU    = 4
3 VCPU   = 16
4 MEMORY = 10240
5
6 NAME   = [APP]ParsoidWorker
7 CPU    = 1
8 VCPU   = 2
9 MEMORY = 512
```

A mérés megkezdésekor letöltöttem az angol Wikipédiáról egy adatbázis mentést (dump), melynek mérete 42 GB volt. A rendszer indításakor megkezdi a feldolgozólánc a dump feldolgozását egy külön szálon, míg a főszálon on-the-fly követi a Wikipédiába kerülő cikkek változásait. A Wikipédia dump feldolgozása egy előfeldolgozó fázissal kezdődik, ahol a későbbi tényleges feldolgozófolyamat követhetőségéért először meg kell számolni, hogy hány cikket kell beilleszteni a dump-ból.



5.3. ábra. A mérés során feldolgozott cikkek statisztikája

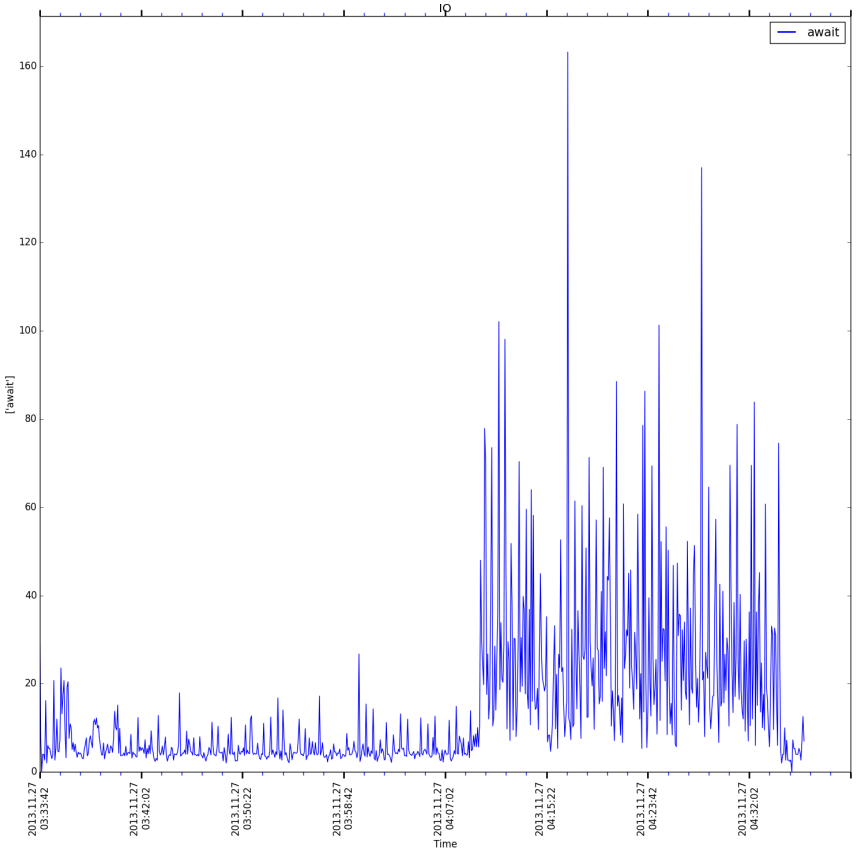
A fent említett előfeldolgozás a mérés során használt Wikipédia mentéssel körülbelül 36 percig tartott, ezalatt 1,36 milliárd cikket olvasott be. Az on-the-fly feldolgozást végző ParsoidWorker-ek száma 25 darab volt, míg a későbbi dump feldolgozást 5 darab ParsoidWorker segítette. Az egy órás mérés végén az adatbázis mérete, üres adatbázissal indítva a mérést 644 Mb lett, ezalatt 4298 cikket dolgozott fel a rendszer. A Statistics komponens által készített részletes feldolgozási statisztikák közül egy látható a 5.3. ábrán.



5.4. ábra. A mérés során mért CPU használat

A teljesítménymméréseket az *iostat* nevű ingyenes és nyíltforráskódú monitorozó szoftverrel végeztem, mely képes a processzor és a diszk használatot is monitorozni, valamint a különböző szálak működésének megfigyeléséhez a *top* programot használtam. A processzor használatot ábrázoló diagramon (5.4. ábra) világosan elkülönül az előfeldolgozás fázisa és az a fázis, amikor

az on-the-fly és dump feldolgozás egyszerre történik. A mérés eredménye az lett, hogy az alkalmazás által használt processzoridő (`%user`, mely a felhasználói módban eltöltött processzoridőt mutatja) körülbelül 2 magot használ ki a rendelkezésre álló erőforrásból, utóbbinál nyilvánvalóan visszaesik kicsit a teljesítmény és átlagosan 1 magnyi erőforrást tud kihasználni a feldolgozólánc.



5.5. ábra. A mérés során mért lemezhasználat

Kérdés lehet, hogy egy ilyen többszálú alkalmazás, amely több szálat is használ miért nem használja ki a bőséges erőforrást? A válasz az, hogy kihasználja az erőforrásokat, csak a grafikonon ez nem figyelhető közvetlenül meg. Az alkalmazás futásakor a top programban világosan látszott, hogy a feldolgozólánc mind a 16 elérhető magot kihasználta a ThreadPool-oknak köszönhetően, azonban a program futásakor legnagyobb részben a processzor más műveletek elvégzésére vár (ezért ilyen magas az `%idle`, azaz tétlenségi idő). Ezen időigényes műveletek hálózati kommunikációt jelentenek: a cikk szövegének letöltése Wikipedia API-n keresztül, kommunikáció a ParsoidWorker-ekkel és várakozás a visszakapott eredményekre. A processzor használat visszaesésére a második részben magyarázat még, hogy a gyorsabb és processzorigényesebb dump feldolgozást végző szálak jobban lefoglalják a processzort a többi szál elől.

Másik kérdés lehet, hogy miért csak ennyi cikket dolgozott fel? A válasz az, hogy egyrészt az on-the-fly érkező cikkek 100%-a fel lett dolgozva, a dump feldolgozásra állított 5 darab virtuális gép teljesítménye pedig egy teljes dump feldolgozásából ennyit tudott elvégezni a lassú Parsoid használatával 25 perc alatt. Továbbfejlesztési lehetőségekről továbbiakban a 6. fejezetben írok.

A lemezhasználatot bemutató grafikon a rendszer fentiekben meghatározott két állapota még jobban meghatározható, mint a processzorhasználatnál. A lemezhasználati statisztikákból olyan metrikát választottam ki, melyből jó következtetéseket lehet levonni. Ez a metrika az átlagos várakozási idő milliszekundumban, amíg az IO művelet végződik. Látható, hogy az első fázisban viszonylag egyenletesen használja az alkalmazás a diszket, míg a dump feldolgozás fázisában hatalmas kilengések vannak és jobban meghatározza a teljesítményt a hálózati forgalomra való várakozás. Összesítve tehát látható, hogy sem a processzor, sem a diszk teljes erőforrását nem lehet kihasználni a kívánt módon implementált rendszerrel, a hálózati kommunikációra való várakozás miatt; a processzor teljesítménye nem összehasonlítható a hálózati teljesítménnyel.

6. fejezet

Értékelés, továbbfejlesztési lehetőségek

A feladatkiírásban specifikált követelményeknek megfelelően elkészítettem a rendszert:

- Áttekintettem az elérhető részben struktúrált tudásbázisokat, mint például a Wikipédiát feldolgozó megoldásokat.
- Ismertettem az OSGi rendszer felépítését, legfontosabb tulajdonságait és használatát. Bemutatásra kerültek az OSGi keretrendszerrel való fejlesztéshez használható eszközök, mint például a gosh szkriptek, és az Apache Felix program.
- Elkészítettem OSGi komponens formájában egy IRC botot, mely a Wikipédia változásairól folyamatosan értesül, így folyamatos bemenetet szolgáltat az on-the-fly feldolgozáshoz.
- Elkészítettem OSGi komponens formájában egy MediaWiki Parsoid parsert használó többszálú megoldást, mely a Parsoid eredeti sebességének többszörösére képes és cluster-t létrehozva a parserekből egy jól skálázható, használható megoldás keletkezett.
- Integráltam az MTA SZTAKI által készített teszt kutatómodult, melynek könnyű beépítése is szemlélteti a rendszer könnyen továbbfejleszthetőségét, kiegészíthetőségét.
- Teszteltem a rendszer képességeit, funkcionalitását egy méréssel, úgy hogy az on-the-fly feldolgozás és egy teljes Wikipédia dump feldolgozása is el lett indítva.

A tesztelés és a mérések során arra jutottam, hogy az elkészített feldolgozólánc képes ellátni azokat a feladatokat, és teljesíti a meghatározott követelményeket, melyek a rendszer tervezése előtt célul lettek kitűzve. Az elkészített alkalmazás úgy lett megtervezve, hogy a korábbi megoldások hiányosságait próbálja pótolni és az on-the-fly feldolgozás miatt hiánypótló is lehet a részben struktúrált adatforrást feldolgozó rendszerek között.

A hasonló megoldások vizsgálatánál láttuk, hogy egy Wikipédiához hasonló mennyiségű adathalmaz feldolgozása semmilyen esetben sem triviális, a használható feldolgozáshoz idő és számítási kapacitás szükséges. A teszteléskor rendelkezésemre álló erőforrások elegendőek voltak a funkcionalitás ellenőrzéséhez, de számítási kapacitásuk a használhatóságot nem érte el. Korábbi vizsgálatok azonban azt mutatták, hogy a rendszer lineárisan skálázódik, így több erőforrás alkalmazásával a feldolgozólánc minden előnyét ki lehet használni.

A rendszer az OSGi keretrendszer előnyeit teljes mértékben kihasználja, a különböző feladatot ellátó komponensek, a WikiBot, Parser és a DBConnector csatolása rendkívül kicsi és azok könnyen menedzselhetőek. Felhasználhatóság szempontjából sokat javított a komponensek funkcionalitásán a felügyeletre tervezés implementálása, a Logger és Statistics komponensek elkészítése.

A SZTAKI által készített elemző kódok integrálása a rendszerbe nagyon könnyű feladat volt. Ez is mutatja, hogy a komponensek jól lettek megtervezve, újabb, más funkcionalitással rendelkező komponens integrálása a rendszerbe sem lenne túl nagy feladat. Remélhetőleg más kutató modulok beépítésére is sor kerül a jövőben, és hasznos segítsége lehet a kutatásoknak a WikiProcessor feldolgozólánc. További kiegészítések és továbbfejlesztési lehetőségek lehetőségeit a következő pontokban foglalom össze.

Parser komponens hálózati művelet nélkül A mérés során láttuk, hogy a rendszer rendkívül sokat vár hálózati kommunikációra. Egy ilyen nagy teljesítményű rendszernél ez hátrányosan befolyásolja a működést, így érdemes lenne a rendszer leglassabb részét a Parsoid parsert beépíteni a feldolgozóláncba, illetve a hálózati kommunikációt minél jobban kikerülni. Ehhez egyrészt a ParsoidWorker-eket a WikiHarvester géppel azonos gépre kell költöztetni, de hálózati kommunikáció megszűntével elért gyorsítás miatt valószínűleg kevesebb példány futtatása is elegendő lenne.

A kommunikáció azonban a NodeJS-ben írt Parsoid és az OSGi platform között nem egyszerű feladat, így több alternatívát is érdemes lesz kipróbálni. A különböző technológiát használó alkalmazások kommunikációjára egy lehetséges megoldás lehet az interprocessz kommunikáció (*IPC*), amelyet mindkét nyelvből natív módon is el lehet érni, például *MessageQueue* használatával. Másik lehetőség lehet a NodeJS C++-ban implementált V8 JavaScript motorja és az OSGi közötti kommunikáció *JNI* segítségével. A NodeJS-hez létezik is már JNI wrapper a Java API-hoz, így a kommunikáció megoldható lenne, ennek a modulnak a segítségével.

Feldolgozóláncok készítése egyszerűbben, illetve dinamikusan Az eredeti feldolgozólánc architektúrája, melyben nem szerepelt még a dumpfeldolgozás, nagyon egyszerű és átlátható volt, így a célnak megfelelt. A jó tervezésnek köszönhetően a dump feldolgozás is csak két meglévő osztály példányosítását és egy a feladatot végző rész implementációját igényelte. Ebben a helyzetben még mindig elég egyszerű és átlátható a rendszer működése, de látható, hogy az ilyen jellegű kiegészítések szempontjából nem skálázódik egy idő után túl jól a rendszer. Újabb feldolgozási láncok kialakítására a jelenlegiek mellett tehát egyre nehezebbé válik. Ennek a problémának a megoldása lehet, ezen feldolgozóláncok létrehozásának általánosítása.

Egyik lehetőség lehet az *Apache Stanbol* [11] nyíltforráskódú és moduláris szoftver stack kiegészítése az eddig elkészített feldolgozólánccal, mely az Apache Stanbol részeként használhatná az teljes software stack egyes komponenseit. Az Apache Stanbol szintén Apache Felix-et használ OSGi futtatókörnyezetként, így ilyen szempontból nem lenne bonyolult a megoldás kivitelezése. A feldolgozóláncok dinamikus előállítása a Stanbol-ban

elérhető *Enhancement Chain*-ek segítségével válna elérhetővé. Ennek a megoldásnak a hátránya, hogy az Apache Stanbol funkcióit nem egészítené ki a feldolgozólánc, hanem inkább csak használna azokból néhányat.

A fent felsorolt okok miatt érdemes lehet, saját implementációt készíteni a feldolgozóláncok készítéséhez, könnyen kiegészíthető és továbbfejleszthető architektúrával. Ilyen problémák megoldására készült az *Apache Commons Chain* [10] implementációja, mely a *Chain of Responsibility* és *Command* tervezési minta kombinációja.

A Chain of Responsibility minta alapgondolata az, hogy szétválasztja a kérést küldõket a fogadóktól azáltal, hogy több objektumnak adja meg a lehetõséget a kérés lekezelésére. A kérés egy objektumokból álló lánc láncszemeit járja be egészen addig, amíg valamelyik láncszem le nem kezeli.

A Command minta parancsok, kérések objektumba ágyazását, és ezen keresztül feladatok delegálását fogalmazza meg. Különféle parancsokat rendelhetünk így klienseinkhez (akár dinamikusan is), a parancsokokat akár sorba is állíthatjuk, naplózhatjuk, illetve segítségével visszavonható műveleteket kezelhetünk.

Feldolgozólánc perzisztenciája A jelenlegi rendszer leállásával a működéssel kapcsolatos több információ is elveszik azokban a komponensekben, melyek működése nem állapotmentes. Az egyik ilyen kiegészítés lehetne, hogy ha már egyszer feldolgozott a rendszer indulás után egy teljes Wikipédia dump-ot, egy következő indításnál nem érdemes még egyszer megtenni azt.

Leállításkor lehetőség van az OSGi komponensekben az aktuális futás közben használt információk elmentésére az adatbázisban. Például a QueueManager példányokban tárolt információkat ilyenkor lehetne elmenteni, vagy a Statistics komponens által készített statisztikák eltárolását is ekkor kellene megoldani.

A félév során folyamatosan újabb és újabb technológiákat ismertem meg, ezekben sikerült kellőképpen elmélyedni, így azt hiszem rengeteget tanultam az elvégzett munka alatt. Ezen kívül sikerült egy olyan szoftverrendszer teljes életciklusát végigkövetni és megvalósítani, mely megfelel a kitűzött céloknak és egy használható keretrendszert nyújt azok számára, akik egy ilyen hatalmas tudásbázist akarnak egyszerűen felhasználni, mint a Wikipédia.

Ábrák jegyzéke

Táblázatok jegyzéke

Forráskódok jegyzéke

Irodalomjegyzék

[1] OSGi Alliance. Osgi service platform release 2. *OSGi Alliance Specifications*, pages 1 – 288, 2001.

[2] Marco Tulio Valente Andre L. C. Tavares. A gentle introduction to osgi. *ACM SIGSOFT Software Engineering Notes*, 33:1 – 5, 2008.

[3] Peter Murray-Rust CJ Rupp Advaith Siddharthan Simone Teufel Ben Waldron Ann Copestake, Peter Corbett. An architecture for language processing for scientific texts. *In Proceedings of the 4th UK E-Science All Hands Meeting*, 1:1 – 8, 2006.

[4] Neil Bartlett. Bndtools Tutorial. `http://bndtools.org/tutorial.html`, 2011. [Online; hozzáférés 2013.04.25].

[5] Leon Blakey. PircBotX Java IRC Bot. `https://code.google.com/p/pircbotx/`, 2013. [Online; hozzáférés 2013.11.11].

[6] Joshua Bloch-Joseph Bowbeer David Holmes Doug Lea Brian Göetz, Tim Peierls. *Java Concurrency In Practice*. Addison Wesley Professional, 2006.

[7] Simone Paolo Ponzetto Eduard Hovy, Roberto Navigli. Artificial intelligence, wikipedia and semi-structured resources. *Artificial Intelligence*, 194:1 – 252, 2013.

[8] H2 Database Engine. H2 Database Engine. `http://www.h2database.com/`, 2013. [Online; hozzáférés 2013.11.12].

[9] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Desgin Patterns: Elements of Reusable Object-Oriented Software*. Kiskapu Kft., 2004.

[10] The Apache Software Foundation. Apache Commons Chain. `http://commons.apache.org/proper/commons-chain/`, 2010. [Online; hozzáférés 2013.11.12].

[11] The Apache Software Foundation. Apache Stanbol. `http://stanbol.apache.org/`, 2010. [Online; hozzáférés 2013.11.12].

[12] Richard S. Hall. Apache Felix OSGi Tutorial. `http://felix.apache.org/site/apache-felix-osgi-tutorial.html`, 2011. [Online; hozzáférés 2013.09.25].

[13] Klaus Berberich Edwin Lewis-Kelham Gerard de Melo Gerhard Weikum Johannes Hoffart, Fabian M. Suchanek. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. *Proceeding WWW '11 Proceedings of the 20th international conference companion on World wide web*, pages 229 – 232, 2011.

[14] Paul Mutton. PircBot Java IRC Bot. `http://www.jibble.org/pircbot.php`, 2013. [Online; hozzáférés 2013.11.11].

[15] Solt Illés Farkas Tamás Oláh Tibor, Héder Mihály. Sztakipédia Parser. `https://code.google.com/p/sztakipedia-parser/`, 2012. [Online; hozzáférés 2013.11.11].

[16] Humberto Cervantes Richard S. Hall. Challenges in building service-oriented applications for osgi. *Communications Magazine, IEEE*, 42:144 – 149, 2004.

[17] Humberto Cervantes Richard S. Hall. An OSGi implementation and experience report. In *Proceedings of the Consumer Communications and Networking Conference (CCNC 2004)*, pages 394 – 399, 2004.

[18] Simone Paolo Ponzetto Roberto Navigli. BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193:217 – 250, 2012.

[19] Da Qing Zhang Tao Gu, Hung Keng Pung. Toward an osgi-based infrastructure for context-aware applications. *IEEE Pervasive Computing*, 3:66–74, 2004.

[20] Iryna Gurevych Torsten Zesch, Christof Müller. Extracting lexical semantic knowledge from wikipedia and wiktionary. *Proceedings of the 6th International Conference on Language Resources and Evaluation*, pages 1 – 7, 2008.

[21] Benjamin Borschinger Cacilia Zirn Anas Elghafari Vivi Nastase, Michael Strube. Wikinet: A very large scale multi-lingual concept network. *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pages 1 – 8, 2010.

Függelék

6.1. WikiProcessor

A feldolgozólánchoz tartozó kódok Apache License 2.0 licensz alá tartoznak. A forráskódok, és a működtetéshez szükséges segédprogramok és szkriptek szintén ebben a repository-ban találhatóak meg.

Webes elérés:
`https://bitbucket.org/thesnapdragon/wikiprocessor`
Git URL:
`git@bitbucket.org:thesnapdragon/wikiprocessor.git`
Licensz:
Apache License, Version 2.0