Általános információk, a diplomaterv szerkezete

A diplomaterv szerkezete a BME Villamosmérnöki és Informatikai Karán:

1. Diplomaterv feladatkiírás
2. Címoldal
3. Tartalomjegyzék
4. A diplomatervező nyilatkozata az önálló munkáról és az elektronikus adatok kezeléséről
5. Tartalmi összefoglaló magyarul és angolul
6. Bevezetés: a feladat értelmezése, a tervezés célja, a feladat indokoltsága, a diplomaterv felépítésének rövid összefoglalása
7. A feladatkiírás pontosítása és részletes elemzése
8. Előzmények (irodalomkutatás, hasonló alkotások), az ezekből levonható következtetések
9. A tervezés részletes leírása, a döntési lehetőségek értékelése és a választott megoldások indoklása
10. A megtervezett műszaki alkotás értékelése, kritikai elemzése, továbbfejlesztési lehetőségek
11. Esetleges köszönetnyilvánítások
12. Részletes és pontos irodalomjegyzék
13. Függelék(ek)

Felhasználható a következő oldaltól kezdődő LaTeXdiplomatervsablon dokumentum tartalma.

A diplomaterv szabványos méretű A4-es lapokra kerüljön. Az oldalak tükörmargóval készüljenek (mindenhol 2,5 cm, baloldalon 1 cm-es kötéssel). Az alapértelmezett betűkészlet a 12 pontos Times New Roman, másfeles sorközzel, de ettől kismértékben el lehet térni, ill. más betűtípus használata is megengedett.

Minden oldalon – az első négy szerkezeti elem kivételével – szerepelnie kell az oldalszámnak.

A fejezeteket decimális beosztással kell ellátni. Az ábrákat a megfelelő helyre be kell illeszteni, fejezetenként decimális számmal és kifejező címmel kell ellátni. A fejezeteket decimális aláosztással számozzuk, maximálisan 3 aláosztás mélységben (pl. 2.3.4.1.). Az ábrákat, táblázatokat és képleteket célszerű fejezetenként külön számozni (pl. 2.4. ábra, 4.2. táblázat vagy képletnél (3.2)). A fejezetcímeket igazítsuk balra, a normál szövegnél viszont használjunk sorkiegyenlítést. Az ábrákat, táblázatokat és a hozzájuk tartozó címet igazítsuk középre. A cím a jelölt rész alatt helyezkedjen el.

A képeket lehetőleg rajzoló programmal készítsék el, az egyenleteket egyenlet-szerkesztő segítségével írják le (A LaTeX ehhez kézenfekvő megoldásokat nyújt).

Az irodalomjegyzék szövegközi hivatkozása történhet sorszámozva (ez a preferált megoldás) vagy a Harvard-rendszerben (a szerző és az évszám megadásával). A teljes lista névsor szerinti sorrendben a szöveg végén szerepeljen (sorszámozott irodalmi hivatkozások esetén hivatkozási sorrendben). A szakirodalmi források címeit azonban mindig az eredeti nyelven kell megadni, esetleg zárójelben a fordítással. A listában szereplő valamennyi publikációra hivatkozni kell a szövegben (a LaTeX-sablon a BibTeX segítségével mindezt automatikusan kezeli). Minden publikáció a szerzők után a következő adatok szerepelnek: folyóirat cikkeknél a pontos cím, a folyóirat címe, évfolyam, szám, oldalszám tól-ig. A folyóiratok címét csak akkor rövidítsük, ha azok nagyon közismertek vagy nagyon hosszúak. Internetes hivatkozások megadásakor fontos, hogy az elérési út előtt megadjuk az oldal tulajdonosát és tartalmát (mivel a link egy idő után akár elérhetetlenné is válhat), valamint az elérés időpontját.

Fontos:

- A szakdolgozatkészítő / diplomatervező nyilatkozata (a jelen sablonban szereplő szövegtartalommal) kötelező előírás, Karunkon ennek hiányában a szakdolgozat/diplomaterv nem bírálható és nem védhető!
- Mind a dolgozat, mind a melléklet maximálisan 15 MB méretű lehet!

Jó munkát, sikeres szakdolgozatkészítést, ill. diplomatervezést kívánunk!

FELADATKIÍRÁS

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell beleszerkeszteni ezt a feladatkiírást.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Elektronikus terelők

Szakdolgozat

Készítette
Csortán Szilárd

Konzulens
dr. Molnár Vince

2020. december 7.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Csortán Szilárd*, 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, 2020. december 7.

Csortán Szilárd
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

1. fejezet

Bevezetés

1.1. Rendszer modellezés

Napjainkban a modellezés fogalom nagyon elterjedté vált az ipari világ valamennyi ágában. A rendszermodellezés egy olyan folyamat, amelyben egy rendszert írunk le különböző absztrakciós szinteken. Minden modell egy különálló nézőpontból mutatja be a rendszert.

A modellek több nézőpontból írják le a rendszert:

- Külső perspektíva, ahol a rendszer kontextusát és környezetét írjuk le
- Belső perspektíva, ahol a rendszer belső komponensei közti és a környezettel való kapcsolatot írjuk le
- Strukturális perspektíva, ahol a rendszer felépítését és a feldolgozott adatok struktúráját ábrázoljuk
- Viselkedési perspektíva, ahol a rendszer viselkedését és különböző eseményekre való reakcióját részletezzük

Mindehhez valamilyen grafikus jelölésre van szükségünk, a legismertebb ábrázolási szemantika a Unified Modeling Language (UML). Az öt legelterjedtebb UML diagram:

- Működési (Activity) - folyamatokat és adatfeldolgozási lépéseket modellezhetünk
- Használati eset (Use case) - a rendszer és a környezete közti interakciókat írjuk le
- Szekvencia (Sequence) - aktorok és rendszer továbbá belső komponensek közti kommunikációt ábrázolják
- Osztály (Class) - objektumokat és ezek közti hierarchikus kapcsolatokat tudjuk leírni
- Állapot (State) - a rendszer viselkedését modellezhetjük különböző belső vagy külső események hatására

1.2. Gamma keretrendszer

A Gamma keretrendszer[1] (Gamma Állapotgép Kompozíciós Keretrendszer) egy olyan eszköz amivel komponens alapú reaktív rendszereket tudunk modellezni, verifikálni vagy akár kódot generálni. A gamma az egyetem Méréstechnika és Információs Rendszerek tanszékén készült, fő fejlesztője Graics Bence.

[1] A keretrendszer weblapja: `gamma.inf.mit.bme.hu`

Reaktív rendszernek nevezzük azt az architekturális stílust, amely segítségével lehetőségünk van különálló alkalmazások egységként való kezelésére úgy, hogy ezek a komponensek továbbá képesek egymás eseményeire és a környezetükre reagálni.

A keretrendszer Yakindu (Yakindu Statechart Tools) alapokon készült, ami egy nyílt forráskódú állapotgépeket modellező eszköz. A gamma ezt továbbviszi és egy magasabb modellezési réteget biztosít a felhasználók számára, amely segítségével hierarchikus állapotgép hálózatokat tudnak kialakítani. A Gamma képes egyszerű állapotgépek és komplex állapotgép hálózatok modell verifikációjára, mindehhez felhasználja az UPAAL-t ami egy modell ellenőrző eszköz. Továbbá, a Gamma segítségével lehetőségünk van a teljes rendszerhez kódot generálni, jelenleg a Java nyelv támogatott.

1.3. Feladat áttekintés

A Gamma által nyújtott műveletekre manapság a modellalapú fejlesztések elterjedése miatt egyre nagyobb igény van, ilyen műveletek a kódgenerálás, modelltranszformáció vagy modellellenőrzés. Továbbá, a korszerű eszközök közt a felhő alapú megoldások nagyon elterjedté váltak. Az említett funkciók arányos erőforrás igénye és a folytonos integráció lehetősége miatt, ideális lenne ha a két világot ötvöznénk.

A Gamma egy Eclipse-alapú eszköz, így jelen formájában nem tud felhőben futó szolgáltatásként létezni. A feladat ennek megfelelően az, hogy az alkalmazás szolgáltatásként is használható részeit parancssoron keresztül, illetve OpenAPI segítségével webes szolgáltatásként is elérhetővé tegyük.

1.4. Motiváció

Ebbe a fejezetben részletezni fogom a feladatban említett felhő és webes szolgáltatások előnyeit.

1.4.1. Felhő alapú megoldások

Napjaikban egyre gyakrabban merül föl a Felhő/Cloud fogalom, ez annak köszönhető, hogy az informatika világában valamennyi kis- és nagyvállalat egyaránt vezet be különböző felhő alapú szolgáltatásokat. Három fő kategóriába tudjuk csoportosítani a felhőalapú szolgáltatásokat:

1. **Software as Service (SaaS)** leggyakrabban használt megoldás, alkalmazások disztribúciója az interneten keresztül felhasználók számára
2. **Infrastructure as Service (IaaS)** Skálázható és automatizált erőforrások
3. **Platform as Service (PaaS)** Olyan környezetet biztosít a fejlesztők számára, amelyen kényelmesen tudnak alkalmazásokat fejleszteni

Számos előnyei vannak a fenti felhő alapú szolgáltatásoknak, 3 fő kategóriába tudjuk ezeket sorolni:

- **Flexibilitás** : Igény szerint skálázható, publikus és privát adattárolás lehetősége, vezérlési lehetőségek (egy szervezet eldöntheti, hogy milyen szinten szeretné kontrollálni az igényelt szolgáltatását), gazdag portfólió (számos létező eszköz és megoldás közül választhatunk) és nem utolsó sorban biztonsági szempontból is rengeteg létező vagy új megoldást lehet implementálni az adatok titkosításához

- **Hatékonyság** : Elérhetőség - bárhonnan elérhető egy szolgáltatás az interneten keresztül, idő megtakarítás szempontjából fejlesztők gyorsabban tudnak dolgozni, Hardware biztonság - egy esetleges hardware meghibásodás során nem veszítünk adatokat, anyagi megtakarítás - vállalatoknak nincs szüksége szerverekre és más hasonló felszerelésre
- **Stratégiai érték** : Számos terhet vesz le egy felhőalapú szolgáltatás a szervezetek válláról, így több idő marad a fejlesztésekre, mindig friss az igénybe vett szolgáltatás verziója. Az elérhetőség lehetővé teszi, hogy a világ bármilyen részéről emberek közösen tudjanak dolgozni ugyanazon a problémán

1.4.2. Webes szolgáltatások

Egy webes szolgáltatás lehetővé teszi, hogy a meglévő szoftverünket elérhetővé tegyük a világ számára. HTTP protokollt használva nem csak kliensek felé oszthatjuk meg, hanem különböző más létező szoftver komponensekkel való kommunikációt is kitudunk alakítani.

Számos előnye van: Rengeteg már létező megoldás van már belőle, így nehéz elakadni. Jól előredefiniált protokollok lettek meghatározva, így bármilyen szoftveres világ tud kommunikálni egy teljesen más világgal.

1.5. Megoldás

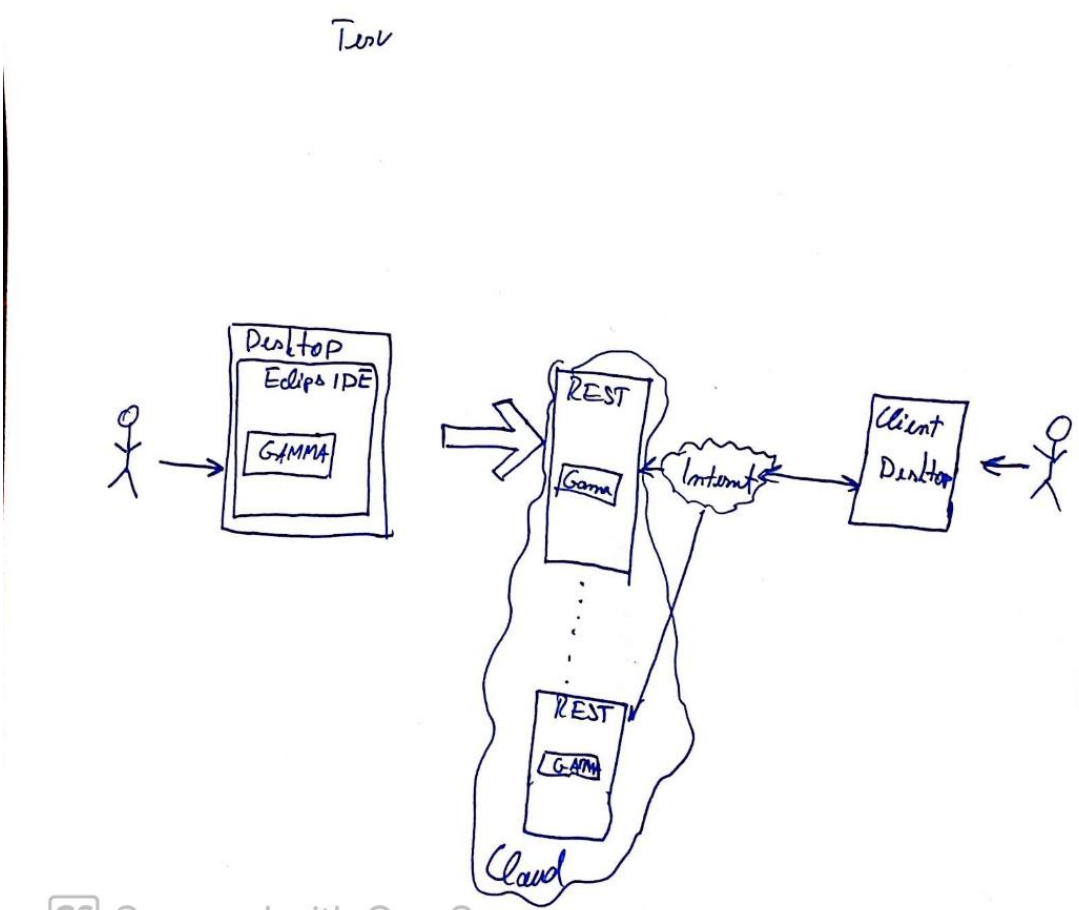
A fentebb leírtak alapján a Gamma egy Eclipse IDE-n belüli eszköz, ennek a transzformációját webes szolgáltatásba az 1.1 ábra írja le:

Alapvetően a Gammát mint Eclipse IDE Plug-in ki kellet emelni, hogy egy fekete szoftver doboz legyen, ami bemeneti paraméterként kap egy műveleteket leíró fájlt, ezután legyártja a különböző műveletek által definiált eredmény fájlokat. A gamma fölé kellet építeni egy webes kiszolgálót, ami képes bejövő kéréseket értelmezni, majd átfogalmazva tovább küldi a Gamma komponensnek. A megoldás során olyan mellék/segítő komponenseket is megkellet tervezni és implementálni, amelyek az adatok átalakításáért vagy mozgatásáért felelősek.

1.6. Dolgozat felépítése

A szakdolgozat keretin belül a fentebb leírt megoldás specifikációját, fejlesztési folyamatát, tesztelési útmutatóját és továbbfejlesztési lehetőségét részletezem.

A következő fejezetben bemutatom az általam választott technológiákat amelyekkel a feladatot megoldottam, itt bemutatásra kerülnek a OSGi, Eclipse RCP, REST API stb. fogalmak. Továbbá, a feladatot specifikálni fogom, megfogalmazott követelményeket mutatok be és végigvezetem a fejlesztési folyamaton és felmerülő nehézségek megoldásában az olvasót. Ezt követően bemutatom a fejlesztést egy konkrét példa segítségével, zárásként pedig a továbbfejlesztési lehetőségekről fogok beszámolni.



1.1. ábra. Gamma infrastruktúra

2. fejezet

Háttér ismeretek

2.1. Modern technológiák

A megoldás folyamán számos modern technológia bevezetésére, implementációjára volt szükség, ezek közül a legfontosabbak:

2.1.1. OSGi

Az OSGi (Open Services Gateway Initiative) egy Java keretrendszer, amely segítségével moduláris szoftvereket és könyvtárakat tudunk fejleszteni.

A technológia alapvetően egy specifikáció halmazból, referencia implementációkból és minden specifikációhoz tartozó teszt halmazból áll össze, mindezek együtt leírnak egy dinamikus moduláris rendszert. A bevált szolgáltatás modellje lehetővé teszi, hogy az infrastruktúra és az alkalmazás komponensek úgy lokálisan, mint hálózaton keresztül is kommunikáljanak egymással, hogy egy koherens és konzisztens architektúrát alkossanak.

Az ipar bármely területén megtaláljuk az IoT-tól kezdve a telekommunikációs megoldásokig, viszont számos esetben nincs kiemelve, azaz ritkán van feltüntetve a használt technológiák listáján.

Kompozícióilag két részből áll az OSGi. Az első egy olyan specifikáció, ami a moduláris egységeket írja le, ezeket bundle-knek vagy plug-in-eknek nevezzük. A specifikáció leírja, hogy milyen infrastruktúra szükséges az említett bundle-k számára és, hogy egyes bundle-k milyen életcikluson mennek át (Telepítve, Elindítva, Aktív, Elakadt, Felfüggesztett), továbbá meghatározza, hogy milyen kapcsolat lehet egyesek bundle-k közt és ezek hogyan kommunikálhatnak egymással. A második pedig egy JVM szintű nyilvántartó, amibe minden bundle tud bejegyzéseket tenni, hogy neki milyen publikálni kívánt objektumai vannak.

Az OSGi a fejlesztési ciklus majdnem minden területén csökkenti a komplexitást. Kódírás és tesztelés jelentősen kevesebb erőforrást fog elnyelni, támogatja az újrahasznosítást (létező komponenseket egyszerűen fel lehet használni egy új modulban). Az üzemeltetési feladatokat is könnyíti, lehetővé teszi a moduláris frissítést vagy telepítést, nem kell minden funkció bevezetésekor az egész rendszert leállítani. Könnyebben lehet lokalizálni egyes hibákat, így a támogatói feladatkör is egyszerűsödik. Az alkalmazásunk moduláris függőségi fájában egyszerűbben tudunk bizonyos elemeket bevezetni, helyettesíteni vagy akár kivenni.

További részletes leírást az OSGi specifikációban lehet olvasni: **OSGi Specification**

Eclipse világban az Equinox OSGi implementációt lehet használni, a fentebb részletezett szolgáltatásokat és szükséges infrastruktúrát alakítja ki számunkra.

2.1.2. Eclipse Plug-in fejlesztés

Eclipse egy nyílt forráskódú, platform független fejlesztői környezet (IDE). Egy alap munkaterület (workspace) és egy plug-in rendszert tartalmaz, aminek segítségével könnyen tesztre szabhatóvá lehet tenni. Az utóbbit használja ki a Gamma is és az általa használt mellék komponensek is.

Az Eclipse IDE egy speciális Eclipse alkalmazás ami lehetővé teszi a fejlesztőknek, hogy egyedi alkalmazásokat tudjanak fejleszteni. Egy Eclipse alkalmazás egy vagy több szoftver komponensből tevődik össze ezeket nevezünk plug-in-eknek. Ezek a modulok bővíthetőek vagy felhasználhatóak más komponensek létrehozására. Összefoglalva, az Eclipse IDE egy több plug-in-ekből (JDT) álló Eclipse alkalmazás.

Lehetőségünk van az így létrehozott alkalmazásainkat összecsomagolni és az Eclipse IDE- től függetlenül futtatni és tesztelni (**headless Eclipse**). Ehhez definiálnunk kell egy termék leírást (product configuration), aminek tartalmaznia kell az alkalmazásunk plug-in-jét, minden felhasznált függőséget és minden más olyan konfigurációt ami szükséges ahhoz, hogy egy külön komponenseként tudjuk hordozni a szoftverünket. Ilyen konfigurációk, az alkalmazás belépési pontja (extension point), komponensek indulási sorrendje, esetleges ikonok és függőségi fa, ami meghatározza, hogy milyen komponens mitől és hogyan (kötelező, opcionális) függ valamilyen más modultól.

A termékleírásunk egy alkalmazás osztályra kell mutasson ami alapértelmezetten két fajta lehet:

```
org.eclipse.ui.ide.workbench - IDE alapú alkalamzások

org.eclipse.e4.ui.workbench.swt.E4Application - RCP alkalmazások, függetlenek az IDE-őtl
```

Ezektől el lehet térni és egy teljesen egyedi alkalmazást írni, ehhez a következő interfészt kell implementálni és behivatkozni a termék konfigurációnkba :

```
org.eclipse.equinox.app.IApplication
```

A dolgozatban leírt fejlesztés ezzel a megoldási opcióval lett megvalósítva.

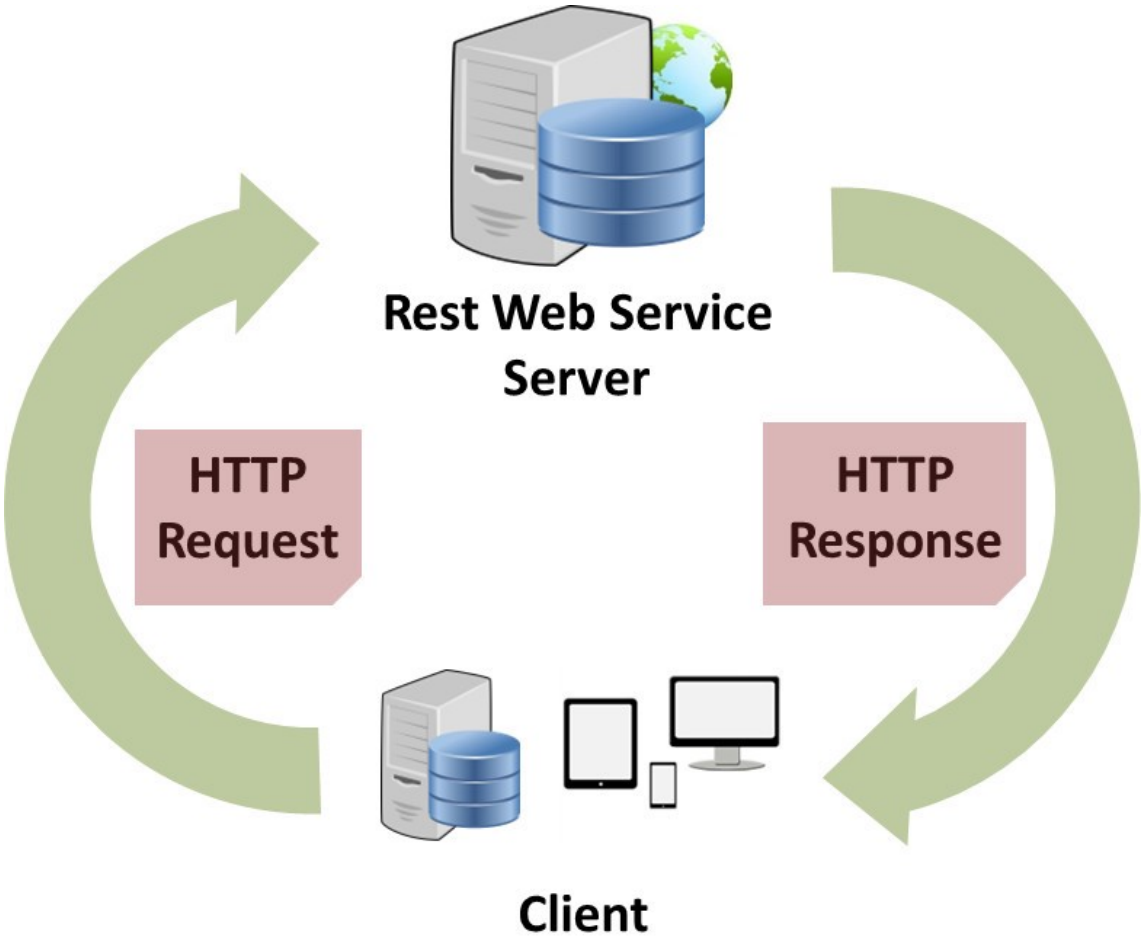
2.1.3. REST API

Az internet megszületése óta, mikor Charley Kline és Bill Duvall összekötöttek két számítógépet és elküldték egymásnak az ”L” és ”O” karaktereket, számos internet alapú adatmozgatási megoldás született ilyenek a CORBA, RPC vagy SOAP. Ezek mind nagyon komplex mechanizmusok és a mai világban elavultnak számítanak. Itt jön szóba a REST a legújabb iteráció az internetes adatszállítási megoldások körében.

REST vagy **RE**presentational **S**tate **T**ransfer egy tervezési stílus, amellyel különböző rendszerek közti kapcsolatot tudunk megvalósítani. Olyan rendszereket, amelyek megfelelnek a REST szabványnak RESTful rendszereknek is nevezünk, alapvetően ezeket két fő jellemző írja le, az első, hogy nem tartják nyilván az egyes kérések állapotát, a második pedig az, hogy elkülönítik a kliens és szerver fogalmakat.

Ahhoz, hogy egy szolgáltatást RESTful-nak nevezzünk meg kell felelni egy szabványnak ami hat alapelvből áll.

Szerver és Kliens szeparáció a tervezési minta előírja, hogy a szerver és a kliens két független és önállóan egységként létező komponens kell legyen. Ez lehetővé teszi, hogy a két rendszerünket külön tudjuk fejleszteni, telepíteni, javítani, karban tartani. Bármilyen kódmódosítást kell végezni bármelyik oldalon a másikat nem befolyásolja mind addig amíg az interfész nem módosul, ilyenkor a szerver egy master szerepet vesz át és az elvégzett módosítást kezelni kell kliens oldalon is. Ez az elv támogatja a moduláris gondolkodást, ezzel flexibilissé válik a komplex rendszerünk. Továbbá, skálázhatósági lehetőségek is felmerül-



2.1.1. ábra. REST egyszerű architektúra

nek, hiszen az erőforrás igényesebb komponens alatti erőforrásokat igény szerint bővíthetjük. A szétválasztás lehetővé teszi, hogy akár egy szerverhez több kliens is csatlakozzon, így sokszínű rendszereket tudunk létrehozni. Ennek egy vizuális reprezentációja a 2.1.1 ábrán látható.

- **Szerver és Kliens szeparáció** - a tervezési minta előírja, hogy a szerver és a kliens két független és önállóan egységként létező komponens kell legyen. Ez lehetővé teszi, hogy a két rendszerünket külön tudjuk fejleszteni, telepíteni, javítani, karban tartani. Bármilyen kódmódosítást kell végezni bármelyik oldalon a másikat nem befolyásolja mind addig amíg az interfész nem módosul, ilyenkor a szerver egy master szerepet vesz át és az elvégzett módosítást kezelni kell kliens oldalon is. Ez az elv támogatja a moduláris gondolkodást, ezzel flexibilissé válik a komplex rendszerünk. Továbbá, skálázhatósági lehetőségek is felmerülnek, hiszen az erőforrás igényesebb komponens alatti erőforrásokat igény szerint bővíthetjük. A szétválasztás lehetővé teszi, hogy akár egy szerverhez több kliens is csatlakozzon, így sokszínű rendszereket tudunk létrehozni. Ennek egy vizuális reprezentációja a 2.1.1 ábrán látható.
- **Állapot nélküliség** - Ez az alapelv annyit von magában, hogy a szerver semmilyen információval nem rendelkezhet a kliens állapotáról ennek a kijelentésnek viszont fordítva is teljesülnie kell, azaz a kliens se ismerheti a szerver állapotát. Minden küldött/kapott üzenet egységként lehet kezelni, független attól, hogy a múltban milyen

jellegű üzenet váltások történtek. Ennek eredménye az, hogy a csak kliens tárolja a saját adatainak az állapotát (nem a szerver adatait).

- **Egységes interfész** - Ez az alapelv négy szabályt határoz meg:

  1. A kliens által kezdeményezett kérésnek tartalmaznia kell a fent definiált erőforrásnak az azonosítóját. Az erőforrást (egyed/objektum) hívhatjuk a webes kontextus alanyának is.
  2. A szerver válasza elegendő információt kell tartalmazzon az erőforrásról, pontosabban annyit, hogy a kliens tudja majd módosítani ezt.
  3. Minden egyes hívás az API felé részletes és tartalmilag teljes kell legyen. Továbbá a szerver válaszának is tartalmaznia kell elegendő információt, úgy, hogy a kliens értelmezni tudja ezt.
  4. Az utolsó szabálytól gyakran eltekintenek a fejlesztők, ez azt mondja, hogy a válasz, ami a szervertől érkezik tartalmazhat, olyan útmutatókat a kliens számára amivel az alkalmazás állapotát módosítani tudja. Például, ha egy objektumot lekér a kliens az API-tól, akkor a szerver a válaszba becsomagolhat olyan mellékes információkat amelyek azt írják le, hogy milyen módon módosítható a visszaadott objektum.

  Az egységes interfész lehetővé teszi, hogy a kliens típusától (böngésző, alkalmazás) eltudjunk tekinteni a szerver oldalon.

- **Cache lehetőség** - Cache támogatást úgy lehet elérni, hogy az adatokat verziózzuk, ha a kliens lekér egy adatot akkor valamilyen metaadatban tároljuk azt is, hogy ez milyen verziójú, így ha ugyanazt az objektumot szeretné lekérni, előtte tudja ellenőrizni lokálisan, hogy már a legfrissebb adattal rendelkezik és nem kell a szervert megint megszólítania. Ezzel terheléscsökkentést tudunk elérni.

- **Rétegelt architektúra** - A szerver és az erőforrást lekérő kliens között számos más rendszer helyezkedhet el, ezek többfajták lehetnek de legtöbbször a biztonsági, cache, terhelés egyensúlyozó rétegekkel találkozhatunk.

- **Igény szerinti kód (opcionális)** - Az egyedüli elv ami opcionális, annyit von magában, hogy a kliens bármikor kérhet kódrészleteket a szervertől és ez a válaszban valamilyen HTML vagy szkript formátumba becsomagolja és elküldi ennek , amit majd a kliens futtatni tud. Az alapelv teljesülése nélkül is tud egy rendszer RESTful lenni.

A fentebb többször említett kérés/válasz fogalmakat a REST világban HTTP kérésekbe csomagoljuk, ezek tipikusan a következőket kell tartalmazzák :

1. egy HTTP ige, ami leírja a művelet típusát, jelenleg használt igék:
   (a) **GET** erőforrást tudunk lekérni, nagyon fontos, hogy nem módosíthat állapotot a szerveren
   (b) **POST** erőforrás létrehozásra használják, többszörös hívás esetén különböző erőforrásokat fog létrehozni
   (c) **PUT** már létező erőforrások állapot frissítésére használják, többszörös hívás ugyanazt kell, hogy eredményezze
   (d) **DELETE** mint ahogy a neve is sugallja, erőforrás törlésre használják
   (e) **PATCH** parciális frissítési műveletekre tudjuk használni

(f) a maradék HTTP igéket nem szokták gyakran implementálni a REST világban

2. fejléc, ami metaadatokat tartalmaz az üzenetről, többek közt olyan jellegű információkat, mint az üzenet hossza, a tartalom típusa, a válasz HTTP kód száma és üzenete.

3. az erőforrás elérési útja

4. opcionális üzenettörzs, ami további adatot tartalmazhat

2.1.4. OpenAPI és Swagger

OpenAPI Specifikáció egy API leíró szintaktika REST API-k számára. A specifikáció YAML és JSON formátumban is írható, az így létrehozott fájl a teljes API-t meghatározza, jellemzően a következőket tartalmazza:

- Endpointok halmazát és az ezeken meghatározott műveletek leírását, mint például GET /api/students, ami a tanulók listáját adja vissza vagy a POST /api/students, ami a kliens oldalról felküldött adatok alapján létrehoz egy tanulót a szerver oldalon.
- Műveletek bemeneti és kimeneti paraméterét
- Autentikációs metódusokat
- Licenc, felhasználói feltételek vagy bármilyen adminisztratív információ ami az API-hoz tartozik

Az OpenAPI specifikációt korábban Swagger specifikációnak hívták, amíg a SmartBear Software felajánlotta az OpenAPI iniciatívának. Azóta az OpenAPI 3.0 verziónál tartunk, az új verzió részletes dokumentációjáról és az OpenAPI-ról itt lehet részletesebben információkat olvasni: **OpenAPI 3.0 Documentation**

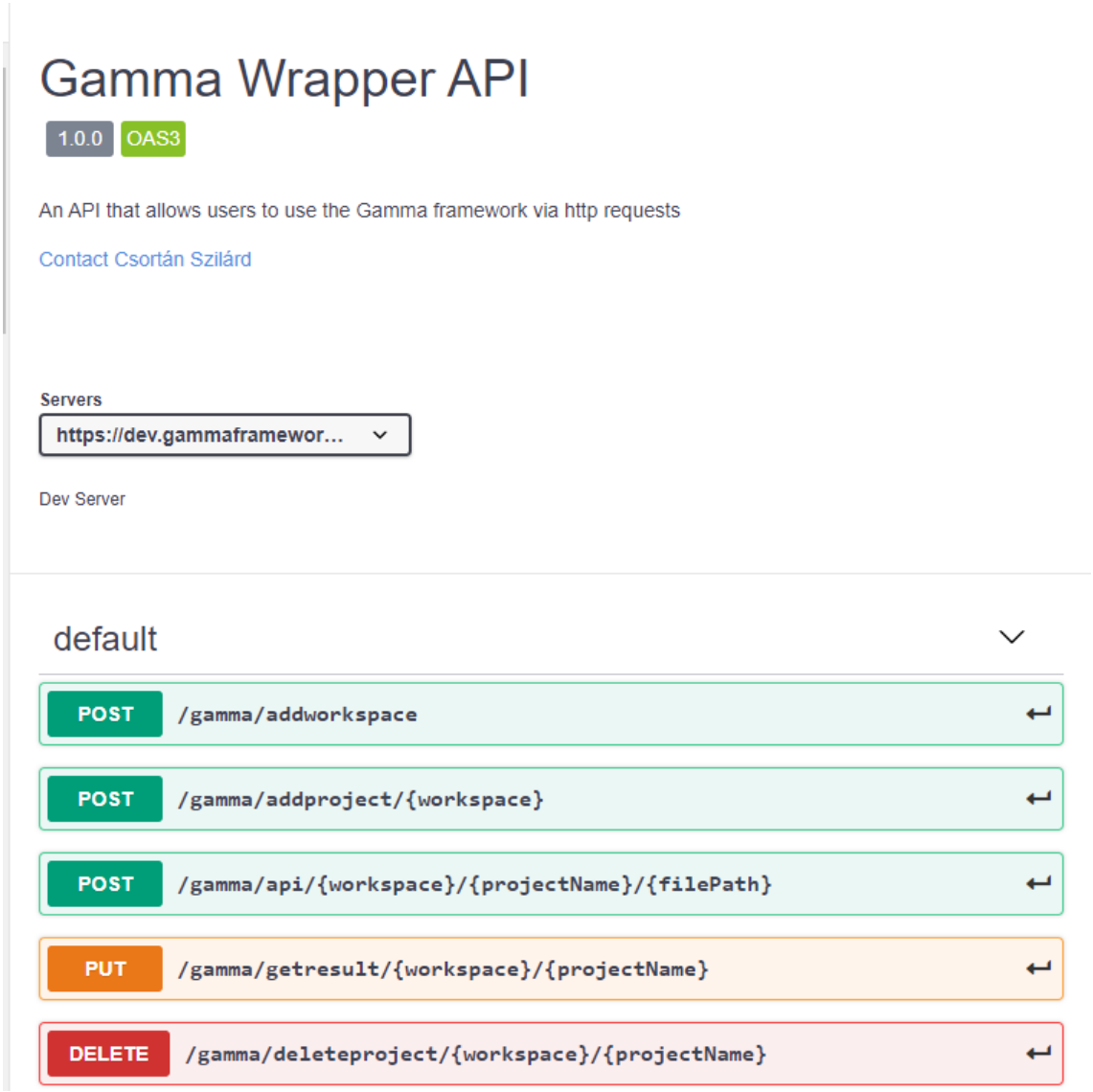
A Swagger egy olyan eszköztár ami az OpenAPI köré épült, hogy segítse a REST API tervezést, implementálást, tesztelést. Számos ilyen eszköz tartozik ide, ezek közül a legfontosabbak:

- **Swagger Editor** Egy böngészőből elérhető felhasználó barát felületet biztosít, ahol lehetőségünk van megírni a specifikációnkat. Számos előnyt kínál egy hagyományos szerkesztőhöz képest, ilyenek a szintaktikai ellenőrzések, bizonyos REST API szabályok betartását is megköveteli, pl: egy GET metódusnak nem lehet törzse.
- **Swagger UI** Egy olvasható, részletes és vizuálisan látványos dokumentációt generál az Editorban elkészített API specifikációnkhoz, a 2.1.2 ábrán egy ilyent lehet megtekinteni.
- **Swagger Codegen** Kliens oldali könyvtárakat és szerver oldali keretet tud generálni.

A specifikációnak tehát rengeteg előnye van, többek közt az egységes API leírás, ami platform független, a kódgenerálás miatt több erőforrás marad az üzleti logika implementálásra vagy az automatizált dokumentáció készítés.

2.1.5. Fontosabb Java könyvtárak

A Vert.x egy eszköztár, amelynek segítségével JVM alapú alkalmazásokat tudunk készíteni. Nagyon flexibilis, azaz nagyon széles az alkalmazás típus spektrum, ahol használni



2.1.2. ábra. Gamma OpenAPI Specification

lehet. HTTP/REST mikroszolgáltatások, komplex webes alkalmazások vagy eseményvezérelt back-end fejlesztése során is használható.

Flexibilitása miatt a piac számos területén használják, például valós idejű játékokban vagy banki rendszerekben.

JVM alapú programozási nyelvek bármelyikében használható (Java, Kotlin, JavaScript, Groovy, Ruby, Scala).

OpenAPI specifikáció integrálását is támogatja. Nagyon egyszerűen lehetőségünk van behivatkozni az API-t leíró fájlunkat, ami alapján a Vert.x legenerál egy interfész réteget, amely mögé nagyon egyszerűen letudjuk implementálni az üzleti logikánkat.

2.2. Gamma

A Gamma egy tanszéken fejlesztett eszköz, amely segítségével komponens alapú reaktív rendszereket tudunk modellezni. Több funkcióval rendelkezik, ezek kerülnek rövid bemutatásra ebben a fejezetben, részletes dokumentációt számos publikáció tartalmaz ezekről itt lehet olvasni: **Gamma Keretrendszer**

2.2.1. Gamma funkciók

Támogatott mérnöki modellek integrációja **modell transzformáció** segítségével történik. Ez a funkció egy létező mérnöki modellt fordít le a Gamma állapotgép nyelvére, ezzel a felhasználót segíti, mivel automatizálva történik a forrás nyelv és a gamma közti leképezés. Jelenleg a Yakindu és UPAAL modellezési nyelveket képes a Gamma transzformálni, viszont a plug-in koncepció további nyelvek leképezését is lehetővé teszi.

A **validáció** egy statikus analízis technika, mely visszajelzést ad a létrehozott modellek szintaktikai helyességéről. A Gamma az önálló modelleken és a kompozit állapotgép komponenseken is végez validációt.

A Gamma képes Java programozási nyelvre **automatizáltan kódot generálni**. Interfész definíciókat a Gamma interfészekből generálja a funkció, a komponens implementációkat pedig a kompozit állapotgép modellezés során definiáltak alapján készíti el.

Két **verifikáció** funkcionalitást támogat a Gamma :

- formális verifikáció, amely formális modellezési nyelvek integrációjával valósul meg. Jelenleg a Gamma keretrendszerbe az UPAAL van integrálva.
- visszavetítés, amely felhasználja a formális verifikáció eredményét és ezt felhasználva generál egy végrehajtási láncot amely alátámasztja vagy megcáfolja az egyes rendszer követelményt.

A Gamma az UPAAL modell ellenőrzőt felhasználva képes **teszteket generálni**, ehhez létrehoz egy absztrakt Gamma modellt, amely alapján legenerálja a teszt eseteket.

2.2.2. Gamma hiányosságok

A Gamma egy Eclipse IDE-ben létező keretrendszer, így jelentős kihívást jelent, hogy egy felhasználó kipróbálja esetleg használja. Kezdve az Eclipse világ rejtelmeitől a különböző függőségi nehézségekig számos hibalehetőség lép föl, ha használni szeretnénk az alkalmazást. Továbbá, vannak olyan modellek amelyek több ezer akár millió nagyságrendű állapottal rendelkeznek, ilyen modelleken a fentebb leírt műveletek futtatása igenis erőforrásigényes. Sajnos az IDE-hez kötöttség nem segít ezen problémák megoldásában, nehéz a skálázhatóság és a disztribúció kérdésekre választ kapni jelen kontextusban.

3. fejezet

Specifikáció és implementáció

Ez a fejezet tartalmazza a 2.2.2 fejezetben leírt hiányosságokra adott megoldásom specifikációját, architektúra tervét és implementációját.

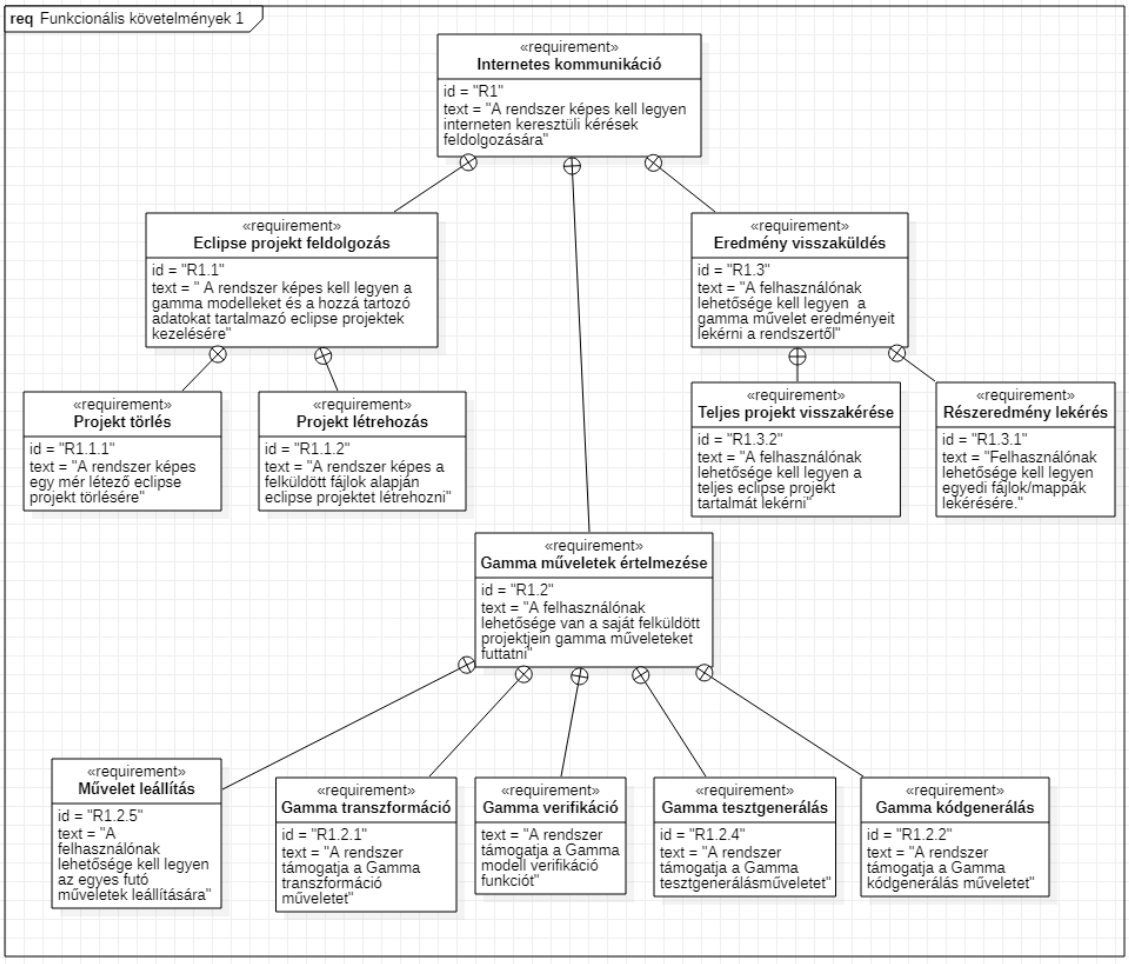
3.1. Specifikáció

A specifikálást a rendszert meghatározó követelmény halmaz kialakításával kezdjük. A követelményeket alapvetően két csoportra lehet osztani, funkcionális és nem funkcionális. Kulcsfontosságú absztrakt követelményeket az alábbi lista foglal össze, viszont nem tartalmazza a teljes körű követelmény hierarchiát, ezt a 3.1.1 és a 3.1.2 ábrákon lehet megtekinteni.

- A rendszer képes kell legyen a gamma modelleket és a hozzá tartozó adatokat tartalmazó eclipse projektek kezelésére.
- A felhasználónak lehetősége kell legyen felküldeni a rendszer számára a saját eclipse projektjeit.
- A felhasználónak lehetősége kell legyen a saját felküldött projektjein gamma műveleteket futtatni.
- A felhasználónak lehetősége kell legyen a gamma művelet eredményeit lekérni a rendszerünktől.
- A rendszer képes kell legyen felhasználók azonosítására.
- A rendszer képes kell legyen inaktív projektek automatizált törlésére.

A további nem funkcionális követelményeket négy kategóriába soroljuk:

- **Megbízhatóság:** Egy projekten nem futtathatunk két különböző Gamma művelet halmazt
- **Biztonság:** Minden felhasználó csak a saját projektjeit szerkesztheti / saját projektjein futtathat gamma művelet halmazokat / A rendszer képes kell legyen kiszűrni az egyes rosszindulatú felhasználók által megadott fájl elérési utakat /
- **Teljesítmény:** A rendszer képes kell legyen különböző projekteken ugyanabban az időben Gamma műveletek futtatására. / A rendszer skálázható kell legyen. / A rendszernek nem szabad fölösleges adatot tárolnia. / A rendszer muszáj töröljön minden olyan adatot, amely neki vagy a felhasználó számára nem releváns. / A rendszer képes kell legyen asszinkron módban működni.



3.1.1. ábra. Követelmény hierarchia 1

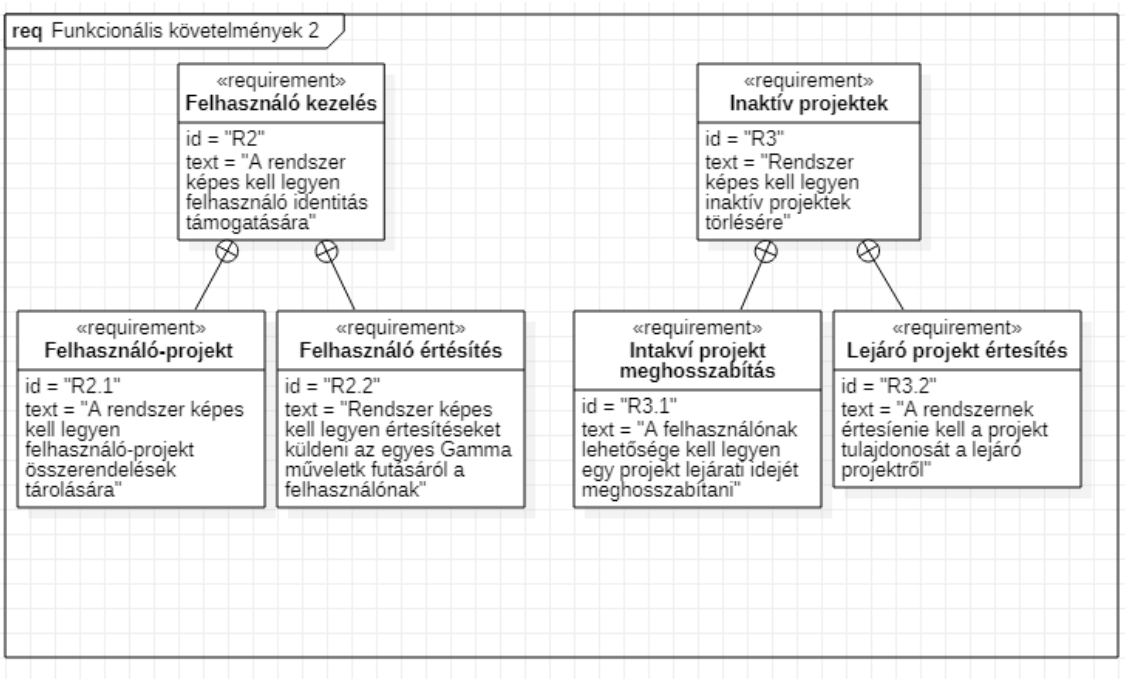
- **Felhasználhatóság:** A rendszer megfelelő, HTTP szabvány által előírt válaszokat adni az egyes kérésekre. / A rendszer beszédes értesítéseket kell küldjön a projekt tulajdonosának az egyes Gamma műveletek futási állapotáról.

A fentebb leírt követelmények alapján bizonyos technológiai döntéseket kellet hozni a projekt iniciális fázisaiban. Ahhoz, hogy az alkalmazásunk interneten keresztül is elérhető legyen egy webszervert kellet kialakítanunk, ez lesz a rendszer belépési pontja. A webszerver a REST API modern architektúra stílus szabályait betartva lett kialakítva. Mindez a *R1.** követelmény teljesítése teszi szükségessé.

Ahhoz, hogy az *R1.2.** követelmény teljesüljön, a Gamma keretrendszert egy *headless Eclipse*-be kellet becsomagolni. Ezzel megtudjuk oldani azt a problémát, hogy a Gamma az Eclipse IDE-hez van kötve, az így becsomagolt keretrendszert parancssoron keresztül lehet elérni.

Az *R2.** követelményt eclipse munkaterek (workspace) használatával teljesítjük. A rendszerünk biztosít munkatér létrehozás funkciót a felhasználói oldalon álló kliens szoftvernek, ehhez rendel egy egyedi azonosítót amit visszaküld a kliensnek és mostantól, ezzel az azonosító megadásával tud a kliens további funkciókat elérni. Mindezzel azt érjük el, hogy majdnem semmilyen információt nem kell tárolni a felhasználóról, ezt a feladatot rábízzuk a kliens szoftverre.

Összefoglalva a követelményeket és a technológiai döntéseket, olyan rendszert tervezünk, amely a Gamma keretrendszert elérhetővé teszi a világ számára, oly módon, hogy



3.1.2. ábra. Követelmény hierarchia 2

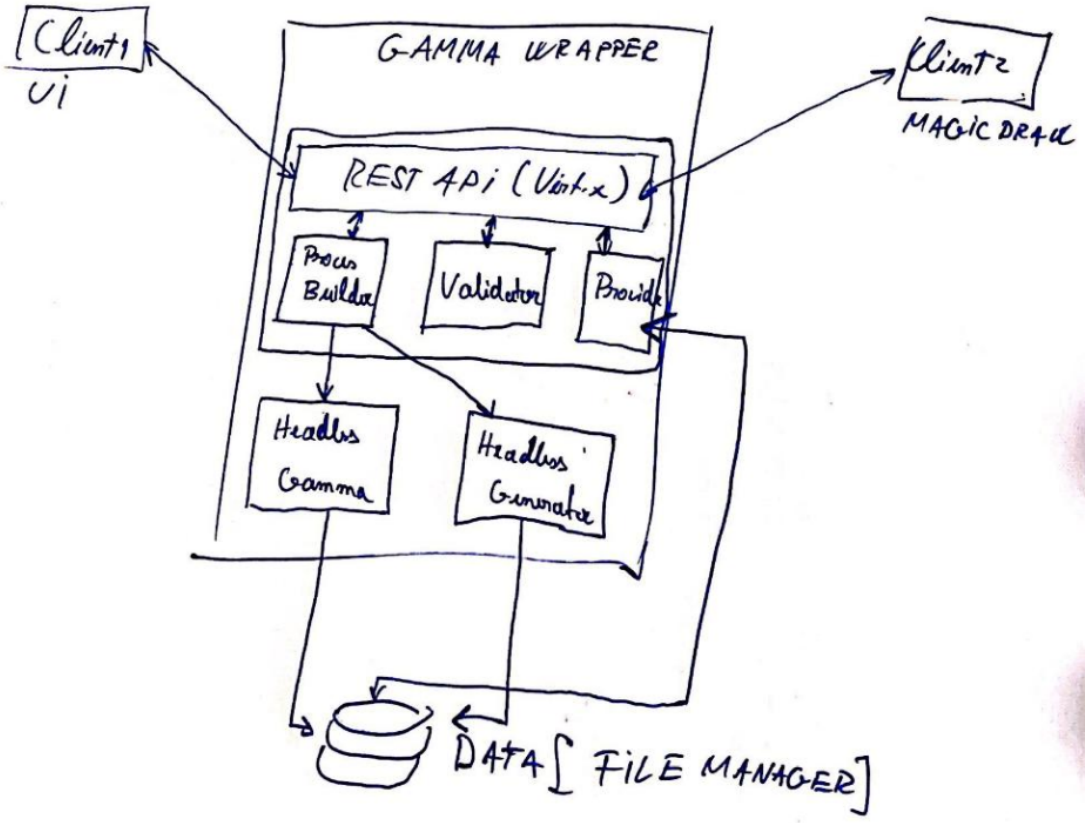
közben több felhasználói rendszerlogikát támogat az adatok tárolásán és funkciók elérésén. Továbbá, olyan mellék funkciókat is biztosítania kell, mint inaktív projektek automatizált törlése vagy felhasználók értesítése.

A rendszerben érdekelt személyek listája (**stakeholders**), azaz olyan aktorok, akik valamilyen módon kapcsolatba léphetnek a rendszerrel:

- Komplex modellekkel foglalkozó **mérnökök** (NASA)
- **Egyetemi kutatók**
- Olyan **fejlesztő** aki integrálni akarja a rendszerünket egy harmadik párti alkalmazásba (MagicDraw)
- **Gamma fejlesztő** ,aki erőforrás igényes funkciót akar tesztelni
- **Üzemeltető**, aki a rendszer komponenseit frissíteni tudja
- **Üzleti oldalú kolléga**, aki a rendszert mint szolgáltatás hirdeti/eladja

A rendszer jelen formájában nem tartalmazza a fentebb többször említett kliens komponenst, így precíz használati eseteket nehéz definiálni. Ennek ellenére a rendszert teljes funkcionalitását lehet használni, tesztelni valamilyen API tesztelő szoftverrel, mint például PostMan. Olyan kéréseket tudunk küldeni az alkalmazás felé, amelyek teljes körűen szimulálják egy kliens szoftver viselkedését.

Az alapvető elérhető funkciók közé tartozik: a munkatér létrehozás, amely egy felhasználónak dedikált eclipse munkateret hoz létre, amin belül majd további műveleteket lehet végezni; eclipse projekt importálása archivált forrásból egy létező munkatérbe; az importált projektben szereplő, műveleteket leíró fájlok alapján gamma műveletek futtatása; a gamma műveletek által generált elemek visszakérése archivált fájlként; importált eclipse projekt törlése.



3.2.1. ábra. Architektúra terv

3.2. Architektúra

Ez a fejezet részletezi a specifikált rendszer struktúráját, továbbá áttekintünk fontosabb folyamatokat.

3.2.1. Struktúra

A rendszer felépítését a 3.2.1 ábra mutatja be. A továbbiakban részletezem az ábrán látható egyes komponenseket, milyen szerepük van és hogyan kommunikálnak más komponensekkel. Az ábrán látható egy nagyobb, mindent magába foglaló *Gamma Wrapper* nevezetű modul, ez a rendszer amely megvalósításra került a félév során és amiről a dolgozat szól. A nevét a funkcionalitásáról kapta, magyarul *Gamma csomag*-nak is nevezhetnénk.

REST API Ez az alkalmazás belépési pontja, ez látható a világ számára. Önmagában egy Vert.x webszerver amely egy OpenAPI specifikációban[1] meghatározott *endpoint*-ok alapján épül fel. Endpointnak hívjuk azokat az egyedi belépési pontokat, amelyek meghívhatók az alkalmazáson kívül. A rendszer az alábbi endpointokat tartalmazza, a fontosabbak működését a következő fejezet részletesen bemutatja:

- POST /gamma/addworkspace eclipse workspace létrehozásának lehetőségét kínálja fel, minden további művelethez szükséges, a kérés nem tartalmaz további információt

[1] OpenAPI specifikáció `https://app.swaggerhub.com/apis/szlnnn/GAMMA_WRAPPER3.0/1.0.0`

- POST /gamma/addproject/*workspace*[2] eclipse projekt létrehozást teszi lehetővé a workspace paraméterből kiolvasott munkatérben, a kérés törzsében szerepelnie kell az archivált fájl, ami tartalmazza az importálni kívánt projektet
- PUT /gamma/api/*workspace*/*projectName*/*filePath* a munkatér és projekt páros alapján meghatározott Gamma műveleteket tartalmazó fájl futtatását teszi lehetővé. A *filePath* paraméter írja le, hogy a műveleteket leíró fájl hol van, ez relatív kell legyen és a *projectName* paraméterben tárolt projekt mappájában
- PUT /gamma/stopprocess/*workspace*/*projectName* a fentebb leírt folyamat leállítását teszi lehetővé, mivel egy projekt-munkatér pároson egy időben csak egy gamma művelethalmaz futhat ezért elég csak ezt a párost megadni
- PUT /gamma/getresult/*workspace*/*projectName* a gamma által generált fájlok visszakérését teszi lehetővé, ehhez szükséges, hogy a kérés törzsében utazzon, hogy milyen fájlokat/könyvtárakat szeretnénk visszakérni, az itt megadott elérések a projekthez képest relatívak kell legyenek, ha szerepel a "." elérés akkor a projekt teljes tartalmát visszaküldjük
- DELETE /gamma/deleteproject/*workspace*/*projectName* eclipse projekt törlését teszi lehetővé
- PUT /gamma/extendexpiration/*workspace*/*projectName* alapértelmezetten minden projekt automatikusan törlésre kerül 30 nappal a létrehozás után, ezzel az endpointal további 30 nappal lehet ezt az időtartamot meghosszabítani, az URL-ben utazó munkatér és projekt név párossal tudjuk eldönteni, hogy melyik projektet kívánja a felhasználó meghosszabbítani.

A komponens további feladatok is ellát, ilyenek a megfelelő HTTP válasz összeállítása, a szerverre küldött fájlok automatikus mentése vagy a más komponensek vezérlése. Egy időzítő is ide tartozik, ezzel ellenőrizzük a projektek lejárati dátumát és, ha ennek eljön az ideje akkor egy törlést fog indítani.

UI és MagicDraw Mivel a rendszerünk egy szerver oldali komponens, ezért önmagában nem tudja bárki használni. Az ábrán feltüntetett kliens rendszereknek kell a felhasználói felület szerepét betölteni, ilyen lenne egy speciális UI ami az alapvető funkcióknak egy grafikus felületet biztosít és kontextust ad az alkalmazásunknak. Továbbá, tervben van a Gamma integrációja a MagicDraw[3] modellező eszközbe, a rendszerünk ezt az integrációt megkönnyítené.

Headless Generator A Gamma önmagában nem képes eclipse munkatér és projekt létrehozásra ezért kellet egy olyan komponens, ami előkészít egy környezetet, amiben futtathatunk Gamma műveleteket, ennek a neve *Headless Generator*. Két fő funkciója van: eclipse munkatér (workspace) létrehozás és eclipse projekt importálás egy megadott munkatérbe. Mindkét funkcionalitást ugyanaz a folyamat valósítja meg annyi eltérésben, hogy ha a komponens kap egy archivált fájl elérést akkor tudni fogja, hogy ezt importálnia kell, mint eclipse projekt. Egy *headless runtime eclipse*-ként van megvalósítva és parancssorból lehet meghívni, két paramétert lehet megadni az első a *-data* amiben a workspace nevét kell megadni, ha egy nem létező workspace-t adunk meg akkor a komponens ezt létrehozza, ez egy kötelező paraméter. A második argumentum pedig a projekt archivált fájl elérése,

[2]Minden döntött betűs rész egy URL-ben utazó paraméter

[3]MagicDraw modellező eszköz `https://www.nomagic.com/products/magicdraw`

ezt fogja kicsomagolni és regisztrálni a munkatérbe, ez az argumentum opcionális. Fontos kiemelni, hogy a forrás fájl már a munkatér mappájában kell legyen.

Headless Gamma Annak ellenére, hogy csupán egy endpoint meghívása során kerül működésbe a rendszer legfontosabb eleme. A Headless Generator-hoz hasonlóan egy headless runtime eclipse-ként lett kialakítva és ugyanúgy parancssorból lehet elindítani. Három kötelező paraméterre van szüksége. (1) Létező munkatér, amely tartalmazza az átadott (2) projekt állományait. Az utolsó (3) paraméter egy a projekthez relatív elérés, ami a gamma műveleteket leíró fájlra mutat. A fájl .ggen kiterjesztésű kell legyen, és megfelelő sorrendbe kell tartalmazza a futtatni kívánt műveleteket és a futtatáshoz szükséges modell állományok hivatkozását, amelyek a projekten belül kell legyenek. A Gamma miután elvégezte a műveleteket tipikusan 3 mappába generálja le az eredményeket: src-gen,test-gen, trace. Ezek mind a projekt mappán belül találhatóak.

Az UPAAL telepítése egy alapvető követelmény a gamma megfelelő működéséhez, ennek az elérést a rendszer a környezeti változókból tudja kinyerni. Az OSGi előnyeit itt tudjuk kihasználni, mivel a headless gamma több száz *plug-in*-t igényel. A plug-in-ek egyesével frissíthetőek, elméletileg nem kéne nagy akadályt okozzon egy új release telepítése.

Process Builder A REST webszerver és a headless komponensek közé biztosítani kell egy olyan réteget, ami a webes hívások paramétereit átalakítja parancssori argumentumokká és eltud indítani egy ilyen runtime eclipse-t, ezt a szerepet a *Process Builder* tölti be. Négy funkció esetén jelenik meg a végrehajtási sorban, (1) munkatér létrehozás, (2) projekt importálás archív fájlból, (3) gamma művelet futtatás és (4) gamma műveletet futtató folyamat leállítás. Minden funkció esetén lehetőség van asszinkron működésre, viszont jelenleg csak a (3) és (4) esetén van implementálva, a szinkron működést azzal az érvel lehet védeni, hogy csak abban az esetben küldjünk választ a kliensnek ha valóban a kért funkcionalitást eltudtuk végezni. Ez a gamma művelet futtatáskor azért nem egy élhető koncepció, mert vannak olyan modellek amelyek több ezer vagy akár több tíz ezer állapotból állnak, ezeknek a feldolgozása órákba is kerülhet így az asszinkron működés egy szükséglet. A gamma művelet leállítása pedig nem igényel semmilyen megvárást, nincs olyan eset amikor ez eltud akadni. A másik két esetben (1)(2) viszont megvárjuk amíg a munkatér vagy projekt létrejön, mert nem költséges műveletek, ezt a jövőben érdemes lehet átgondolni és esetleg átalakítani.

Validator A valamennyi bejövő kérések validációs lépéseken is át kell essenek, ezt a *Validator* modul biztosítja. A REST webszerver vezérli, hogy a bejövő paraméterek közül melyek azok amelyek valamilyen validáción át kell essenek. A legfontosabb ellenőrzési lépések közé tartoznak a munkatér létezés ellenőrzése, a projekt-munkatér páros vizsgálata és az átadott projekten futó gamma folyamat ellenőrzése. Az utóbbi kiemelkedően fontos, hiszen egy futás alatt levő projekttel nem végezhetünk semmit, csak leállíthatjuk. Ahhoz, hogy a futás állapotát nyilván tudjuk tartani, be lett vezetve egy leíró json fájl amit minden projekt importálásánál létrehozunk, ez a fájl többek közt tartalmazza azt is, hogy jelenleg van-e futó folyamat a projekten. Ennek az állapotát a ProcessBuilder beállítja *inProgress*-be majd a HeadlessGamma futása végén frissíti *notInProgress*-re.

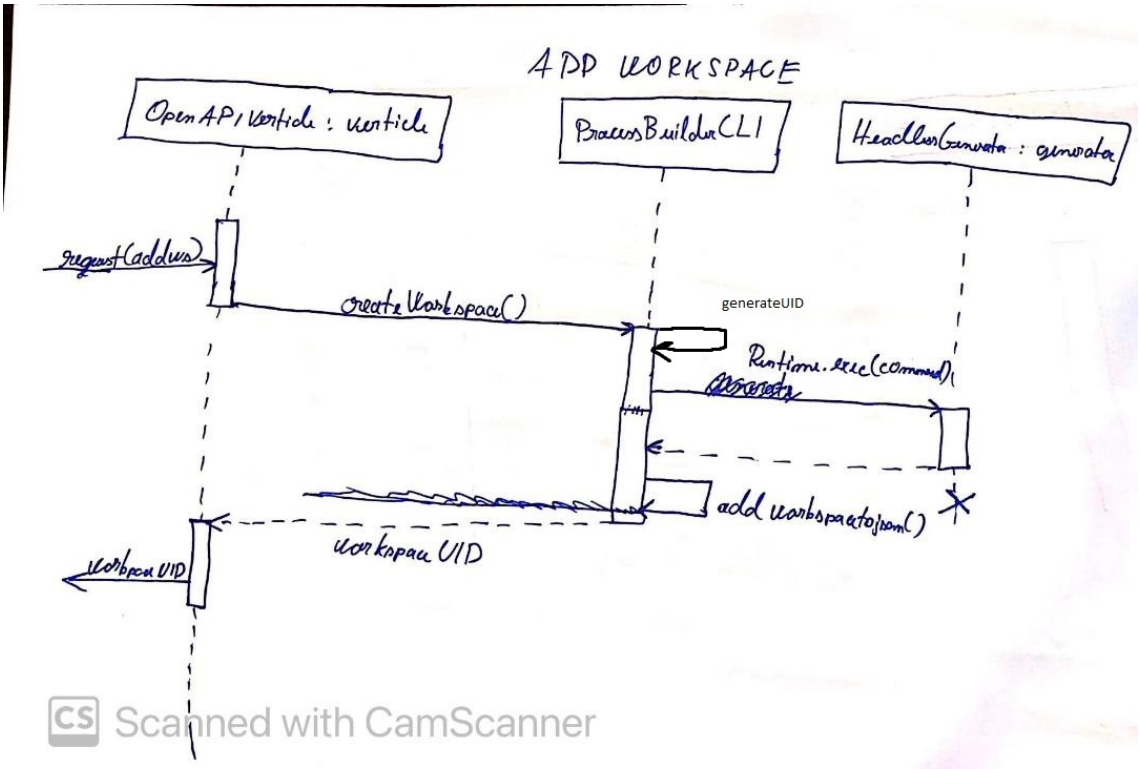
Provider A *Provider*-nek két feladat van, az első, hogy a REST webszerver által feldolgozott eredmény visszakérő endpoint-ban megadott relatív elérések alapján összecsomagoljon egy zip állományt, a másik pedig a projekt törlés. Az eredmény fájl összecsomagolásánál lehetőségünk van megadni egyedi fájlok elérését vagy akár teljes mappákat. Ide tartozik a

teljes projekt mappa is, ezt a "." eléréssel tudjuk megadni, ha ez szerepel a kérésbe akkor a többi eredmény eléréssel nem is foglalkozik a rendszer, hiszen ez mindent tartalmazni fog.

Data Az ábrán megjelenik a *Data* fogalom is, ez valójában a hoszt szerveren levő fájl kezelőt jelenti. Itt tároljuk a létrehozott munkatér és projekt párosokat és ezek teljes tartalmát, továbbá a metaadatokat tároló json fájlokat is. Egy adatbázis réteg bevezetésével a fájlban tárolt metaadatokat könnyebben tudnánk kezelni, sajnos a projektbe ez nem fért bele.

3.2.2. Folyamatok

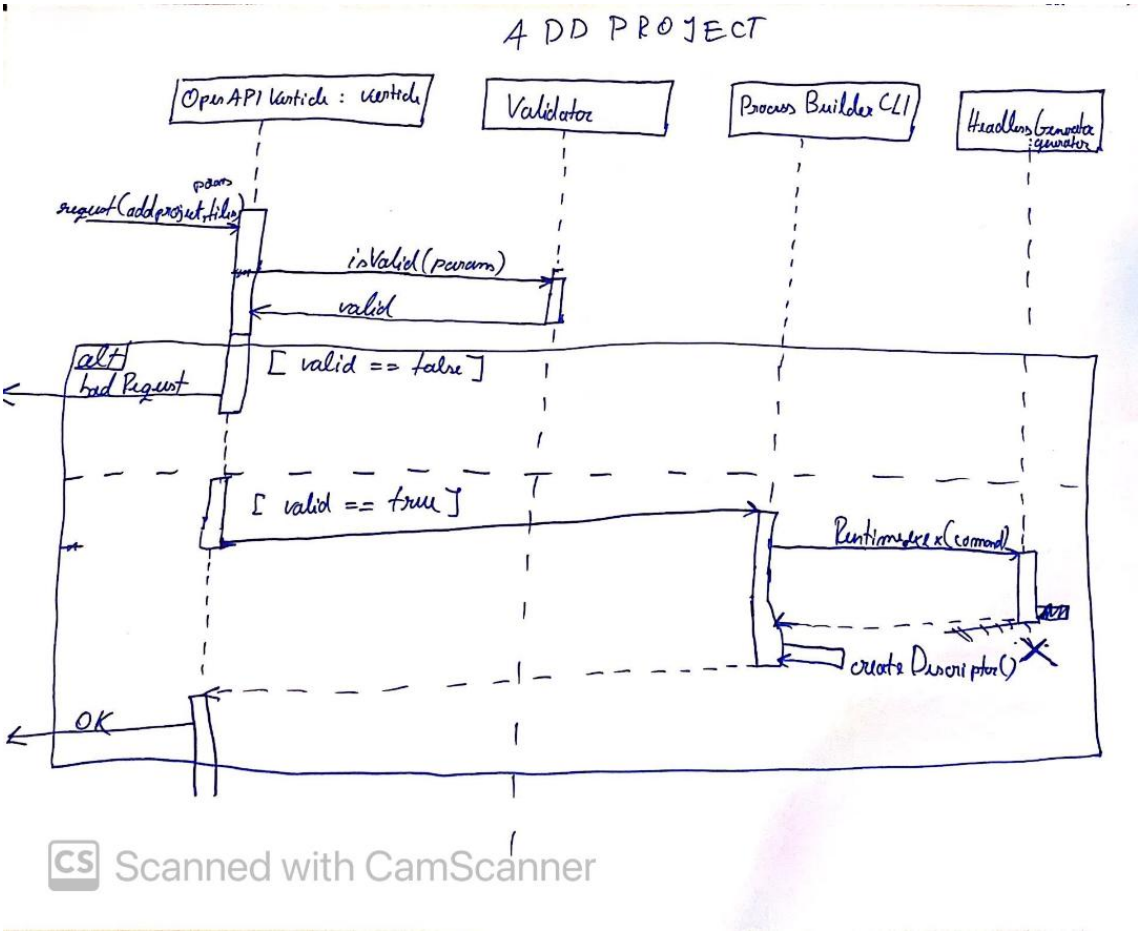
Négy alapvető funkció részletes működését fogom bemutatni ebben a fejezetben, ezek sorra: munkatér létrehozás, projekt importálás, gamma műveletek futtatása és a futtatás eredmény lekérés. A folyamat bemutatja milyen modulokon megy végig az adat, továbbá milyen feldolgozások történnek egyes végrehajtási lépéseken. Mindezt szekvencia diagramokkal ábrázoltam. Az így bemutatott funkciók lefedik a rendszer legfontosabb felhasználói eseteit.



3.2.2. ábra. Munkatér létrehozás szekvencia diagram

Munkatér létrehozás A workspace létrehozás szekvencia diagramja a 3.2.2 ábrán látható. Első fázisban bejön egy POST HTTP metódus az addworkspace endpointra, ennek a törzse üres és az URL-ben sem hordoz paramétereket. Látható, hogy az OpenAPIVerticle fogadja a kérést, ez lényegében a REST webszerverünk, mivel ez a legelső művelet, amit a kliens meg kell hívjon így nem kell semmilyen ellenőrzést végezni. A végrehajtási sor következő lépése a *ProcessBuilder*-ben van, itt a rendszer generál egy egyedi UID azonosítót, amit felhasznál a parancs kialakításhoz, amivel elindítja a a HeadlessGenerator-t.A

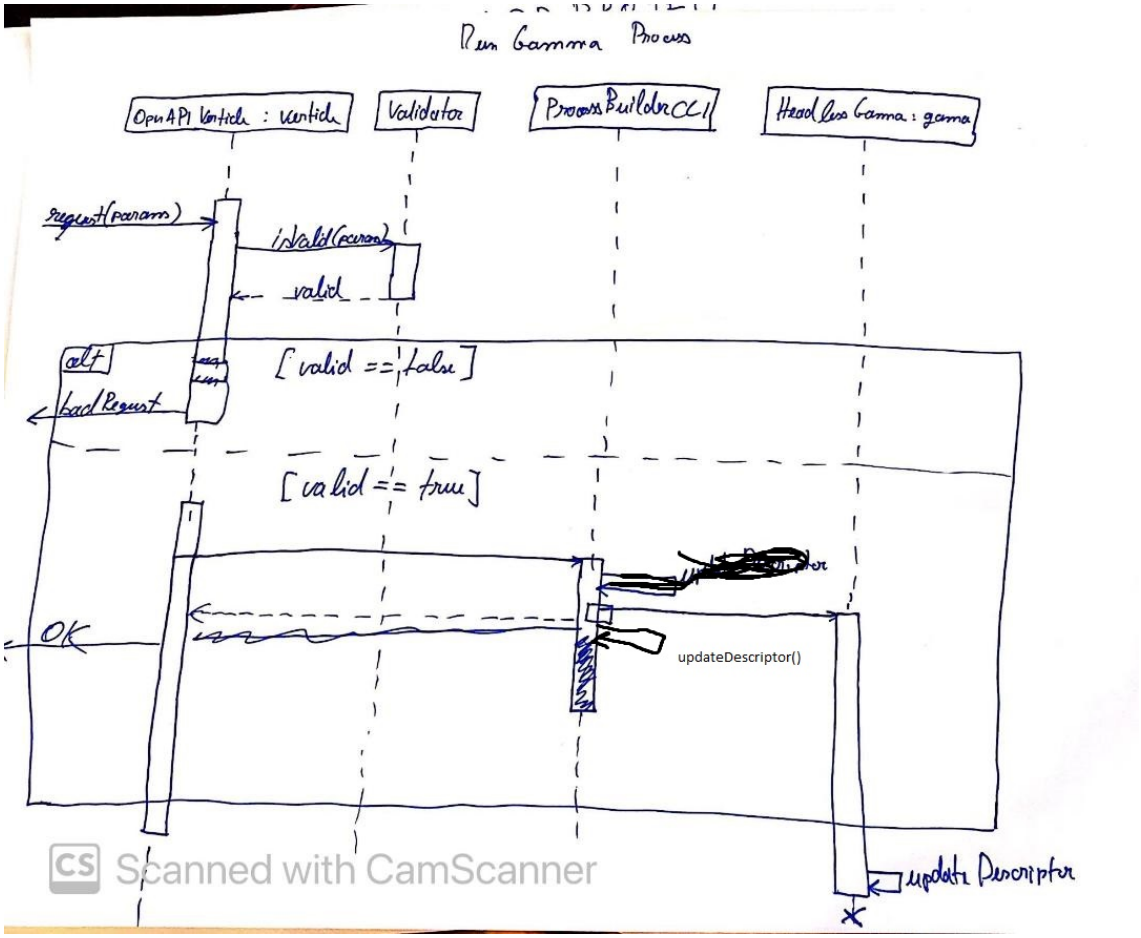
parancs (command) elkészítéséhez a ProcessBuildernek szüksége van a headless eclipse elérésére, ez jelenleg a kódba be van égetve, a továbbiakban célszerű lenne egy konfigurációs fájlba kiemelni. A generator amíg elkészíti a workspace-t a ProcessBuilder vár, ha végzett akkor visszaadja az OpenAPIVerticle-nek a generált UID-ot ami továbbítja a kliens felé. Mikor a headless eclipse elvégezte a feladatot akkor teljesen leáll, ezt jelzi az X a folyamatábrán.



3.2.3. ábra. Eclipse projekt importálás szekvencia diagram

Eclipse projekt importálás A projekt importálás funkció szekvencia diagramját a 3.2.3 ábrán lehet megtekinteni. A folyamat megint a kérés érkezésével indul, ezt minden esetben az OpenAPIVerticle dolgozza fel, ez a kérés egy POST HTTP metódus ami az addproject endpoint-ra jön be. Ebben az esetben viszont már az URL-ben utazik egy munkatér azonosító amit korábban adtunk vissza a workspace létrehozáskor. Továbbá, a törzsben két attribútum szerepel: (1) *files* ez tartalmazza a projekt archivált állományait és (2) *contact* ez egy email cím, ami a projekt tulajdonosáé, erre későbbiekben értesítések küldésénél lesz szükségünk. A munkatér létezését a Validator komponenssel elvégezzük, a kapott válasz alapján két esetet különböztetünk meg, az első, ha nem érvényes a megadott workspace azonosító, ekkor egyből szólunk a kliensnek, hogy rossz a kérés. Viszont, ha ez helyes akkor tovább adjuk a kérést a ProcessBuilder-nek ami az előző esethez hasonlóan elindítja a HeadlessGenerator-t, de kiegészíti az argumentum listát a forrás fájl nevével, mindezek előtt a nyers fájlt a munkatér mappájába másolja. A generator elvégzi az import műveletet, majd visszaadja a futási jogot a ProcessBuildernek, ami végezetül regisztrálja a munkatér-projekt párost és létrehoz egy leíró fájlt a frissen létrehozott projektben, ami

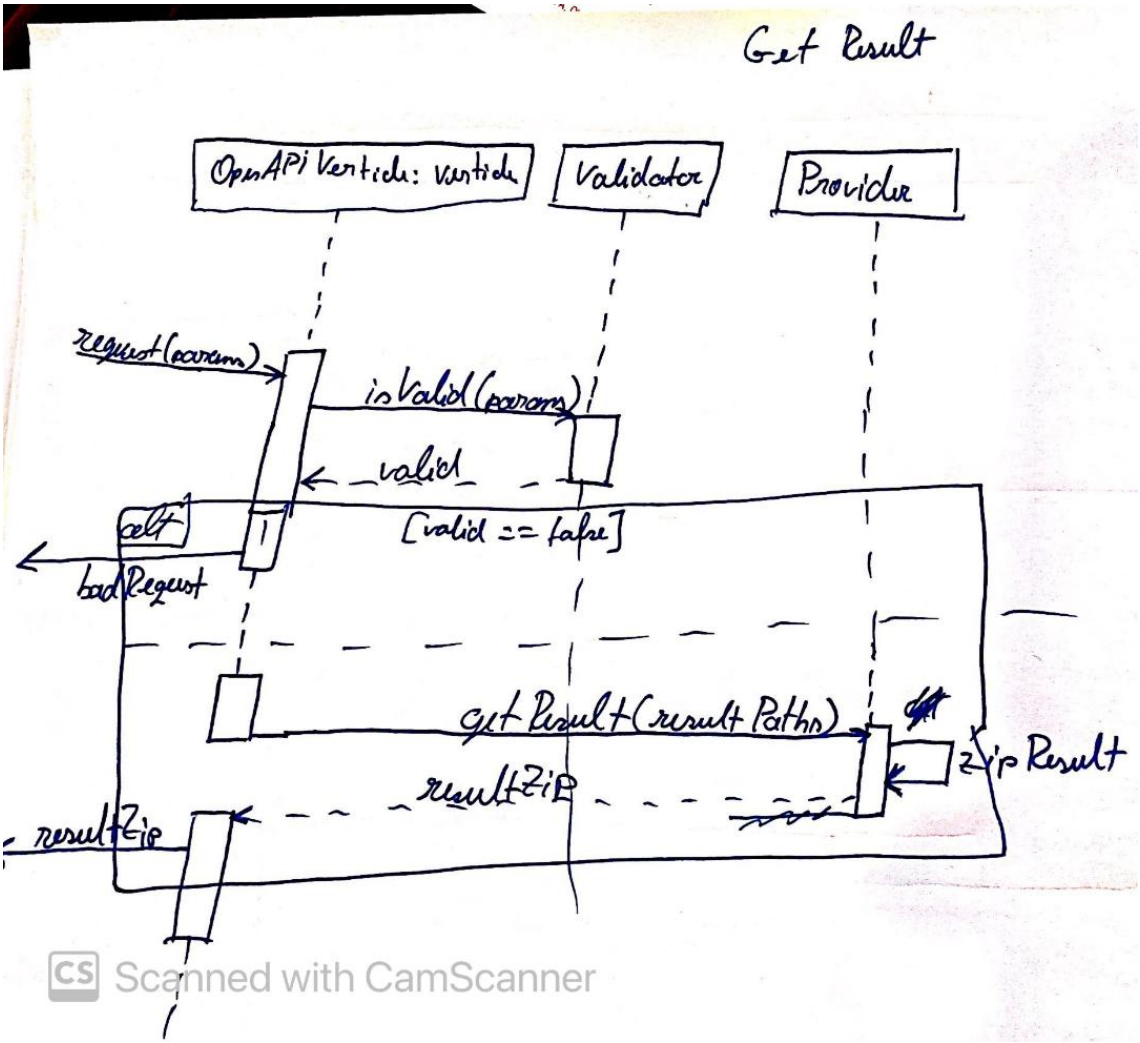
tartalmazza, hogy mi a projekt neve, ki hozta létre (email cím), mikor hozta létre, mikor jár le és beállítja, hogy jelenleg nincs futás alatt, ezek után törli az iniciális forrás fájlt, hogy tárhelyet spóroljunk. Végezetül visszaadja a futási jogot a verticle-nek aki jelez a kliensnek, hogy minden rendben létrejött.



3.2.4. ábra. Gamma művelet futtatás szekvencia diagram

Gamma műveletek futtatása A gamma műveletek futtatását bemutató szekvencia diagram a 3.2.4 ábrán tekinthető meg. Ebben az esetben egy PUT HTTP kérés érkezik az *api* endpoint-ra, amit szintén az OpenAPI réteg dolgoz fel kezdetben. A törzsben nem szerepel semmi, viszont az URL-ben utazik a munkatér azonosító, projektnév és futtatni kívánt gamma műveleteket leíró .ggen fájl elérése. Ez a hármas egyedileg meghatározza az erőforrást. A validációs fázis komplexebb ebben az esetben, hiszen ellenőriznünk kell a munkatér-projekt páros létezését és egyediségét, mert egy projekt egyszer szerepelhet egy munkatérben, továbbá azt is meg kell vizsgálni, hogy a projekt, amit ez a páros leír jelenleg használva van-e valamilyen más gamma műveletet futtató headless eclipse által. A Validator döntése alapján, megint két esetet különböztetünk meg, ha valamiért nem felel meg akkor az OpenAPIVerticle a megfelelő hibakóddal és üzenettel visszaszól a kliensnek, hogy nem érvényes a kérés. Ha minden megfelel, akkor a futást átadjuk a ProcessBuilder-nek, ami asszinkron módon elindítja a gamma műveletet futtató headless eclipset. Ezek után egyből frissíti a projektet leíró fájlt, pontosabban, rögzíti az így elindított folyamat operációs rendszer szintű azonosítóját (PID) és beállítja, hogy a projekt futás alatt van. Végezetül átadja a futást a verticle-nek, ami jelzi a kliens felé, hogy a kérés feldolgozása elkezdődött. A folyamat még nem ér véget, mivel a Gamma dolgozik a háttérben, mikor

minden művelet lefutott amit a .ggen fájlban meghatározott a felhasználó, akkor a headless eclipse frissíti a projektet leíró állományt és beállítja, hogy a projekt már nincs futás alatt. További lezáró művelet lehet az értesítés küldés a tárolt email címre, hogy a kért kérés lefutott és az eredmény mostantól lekérhető.



3.2.5. ábra. Eredmény lekérés szekvencia diagram

Eredmények lekérése A gamma által generált állományok lekérésére szolgáló funkció szekvencia diagramját a 3.2.5 ábra mutatja be. Egy PUT HTTP kérés érkezik a *getresult* endpoint-ra, ennek az URL-jében szerepelnie kell a munkatér azonosítójának és a projektnévnek amiből adatot akarunk lekérni. A kérés törzsében a *resultDirs* attribútum kell szerepeljen, ami egy relatív elérésekből álló lista. A kérés feldolgozása a szokásos módon a verticle-ben kezdődik, ahol a kérésben utazó paraméterek kicsomagolásra kerülnek. A Validator ebben az esetben ugyanazokat ellenőrzi, mint a művelet futtatás folyamat esetén. Ha van futó folyamat a projekten vagy a workspace projekt páros nem megfelelő, akkor beszédes hibaüzenettel válaszolunk a kliensnek. Abban az esetben, ha minden rendben van, akkor a Provider kapja meg a futási jogot. Itt az elérési listán iterálva összegyűjtjük a kért fájlokat és mappákat majd archiváljuk, az így archivált fájl elérést küldjük vissza a verticle-nek, ami ennek alapján visszaküldi a fájlt.

Első látszatra zavaró lehet, hogy PUT metódust használunk a kérés feldolgozásához, mikor egy GET jobb lehetne és jobban megfelelne a REST szabványnak. Ennek két oka

van, az első az, hogy annak ellenére, hogy *getresult*-nak hívjuk az endpoint-ot az erőforrás állapota módosul hiszen a projekten belül generálunk egy archivált fájlt, a másik oka pedig az, hogy a GET kérésnél nem lehet a törzsben semmit sem küldeni, ezt a REST szigorúan megköveteli, nekünk viszont a becsomagolni kívánt állományokat listában kell megadnunk és ezt az URL-ben kényelmetlen lenne kezelni.

3.3. Implementáció

3.3.1. Fejlesztés folyamata

Ebben a fejezetben a fejlesztés során felmerült nehézségeket és érédességeket mutatom be. Az alábbi információk alapvetően mély technikai szintre lemennek így szárazak lehetnek, viszont a témában dolgozóknak hasznos lehet, mert egyedi hibák és jelenségek megoldását vázolom föl.

Eclipse környezettel kapcsolatos érdekességek Az Eclipse környezetben tapasztalt fejlesztők tudják, hogy bizonyos dolgok nem triviálisak a fejlesztés során és olyan specifikus problémák merülhetnek fel, amelyekre nagyon elrejtett internetes fórumokon lehet választ kapni. Az alábbi jelenségek és problémák a *Headless Gamma* komponens fejlesztése alatt merültek föl

Az eclipse plug-in fejlesztés és termék export konfiguráció során számos függősségi problémával szembesültem. A legelső érdekesség ami felmerült, az volt, hogy az Eclipse a *product* konfigurációban meghatározott plug-in függőségek verzióját automatikusan felülírta a környezetbe telepített legfrissebb verzióval. Ezen plug-in-ek listáját az Eclipse IDE-n belül a *Target Platform*-on lehet megtekinteni és szerkeszteni. Ilyen problémás függőségek voltak a *batik.css* és *batik.util* plug-in-ek. A Target Platform szerkesztésével ezt a problémát orvosolni lehet, csupán ki kell vegyük vagy hozzá kell adjuk azt a verziót amire szükségünk van és győződjünk meg, hogy az adott plug-in-ből csak egy verziót aktív.

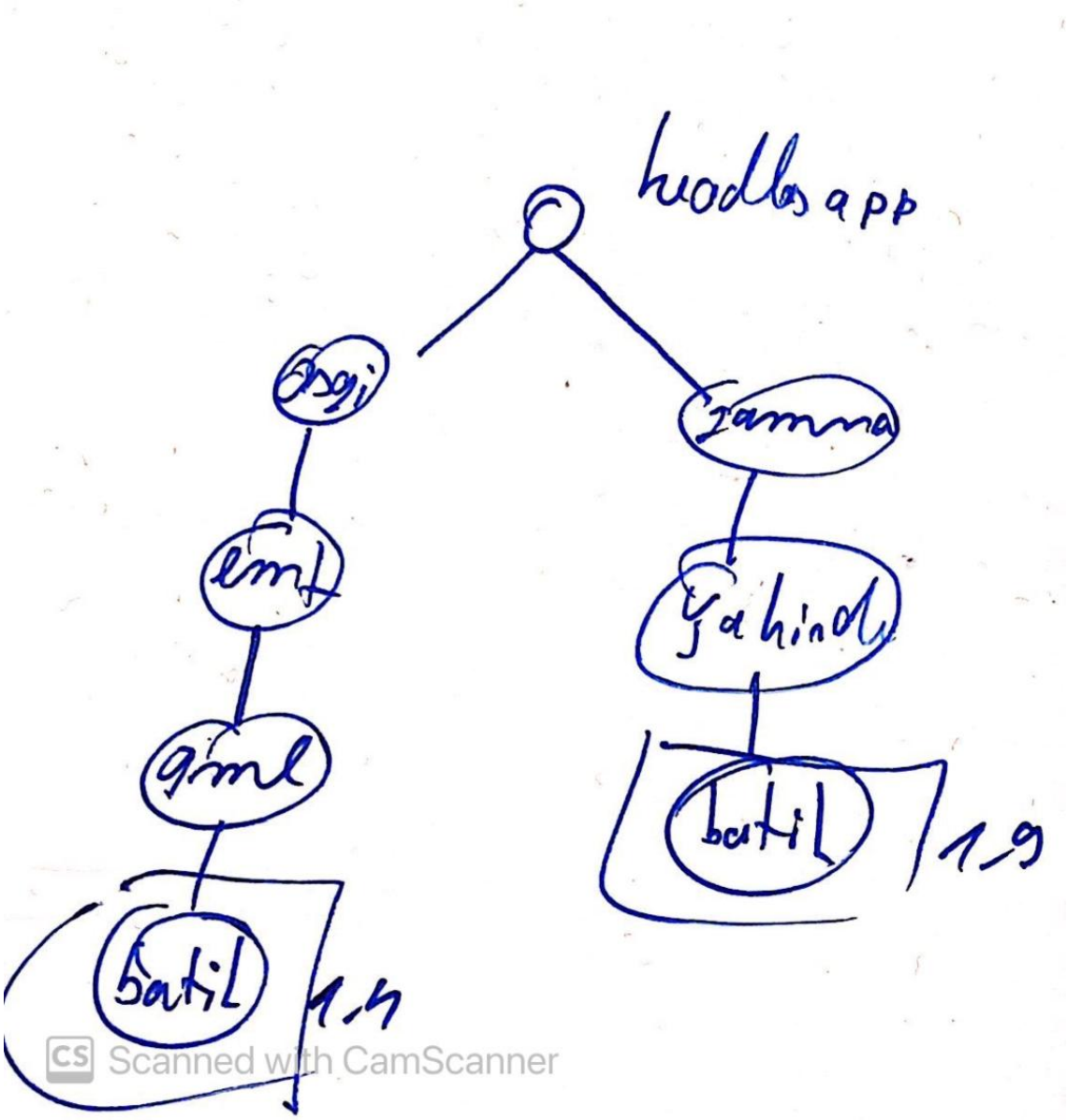
A product konfigurációban a *Content* fülön tudjuk meghatározni milyen plug-in-ekből álljon a termékünk, az IDE-ben lehetőség van arra, hogy megadjuk a fejlesztett alkalmazásunkat és rányomva az *Add Requiered* gombra az IDE automatikusan beállítja a termékünk összes függőségét. Ennek ellenére az osgi.extender olyan plug-in-eket igényel, amelyek az OSGi helyes működéséhez szükségesek de nem importálja be automatikusan. A szükséges modulok:

```
org.apache.felix.scr
org.eclipse.equinox.event
org.eclipse.compare.core
org.eclipse.fx.osgi
org.eclipse.team.core
```

Olyan jelenség is előjött, ahol a függőségi fában egy plug-in kétszer szerepelt viszont különböző verzióval, ezt az OSGi szabvány engedi de az Eclipse, az első jelenségben leírtak alapján, folyamatosan felülírta az elavultabb verziót. A probléma elképzelésében segít a 3.3.1 ábra. Erre a megoldás az, hogy a fában felfele haladva megvizsgáljuk, hogy milyen verziókat tudunk frissíteni a Target Platformon úgy, hogy a működést megtartsuk de az elavult verziójú plug-in-t frissíteni tudjuk. Alapvetően egyszerű feladatnak tűnik, viszont a rendszer közel 250 függőséggel rendelkezik.

A termék export felületen meg kell határozni, hogy az egyes Eclipse modulok milyen sorrendbe induljanak el, a mi esetünkbe az alábbi indulási szintek meghatározása volt fontos:

```
<configurations>
  <plugin id="org.apache.felix.scr" autoStart="true" startLevel="2" />
  <plugin id="org.eclipse.core.runtime" autoStart="true" startLevel="0" />
```



3.3.1. ábra. Egyszerűsített függőségi fa

```
  <plugin id="org.eclipse.equinox.common" autoStart="true" startLevel="2" />
  <plugin id="org.eclipse.equinox.event" autoStart="true" startLevel="2" />
  <plugin id="org.eclipse.equinox.simpleconfigurator" autoStart="true" startLevel="1" />
</configurations>
```

Ez a konfiguráció részlet a termék leírás XML fájlból van másolva, az IDE-ben a *Configuration* fülön lehet mindezt beállítani.

Egy termék konfiguráció futtatására két lehetőségünk van, az első, hogy exportáljuk (headless eclipse) majd parancssorból meghívjuk és átadjuk az argumentumokat vagy az IDE-ből elindítjuk. Az utóbbi tesztelés folyamán nagyon hasznos tud lenni. Ha IDE-ből futtatjuk akkor is van lehetőségünk argumentumokat átadni, erre szintén a termék konfigurációban van lehetőség, pontosabban a *Launching* fülön. Arra viszont érdemes odafigyelni, hogy ezt a tesztelés után ne hagyjuk ott, mivel az export konstans argumentumoknak értelmezi és minden headless módban indulásnál átadja önmagának. A mi esetünkbe ez hatványozottan fontos, mivel az átadott argumentumok minden esetben mások a munkatér és projektek számossága miatt.

Minden függőség lehet opcionális vagy kötelező az opcionálisakat csak abban az esetben tölti be a headless eclipse mikor éppen szüksége van rá. A rendszerünknek a *jdt* plug-in család több komponensére is szüksége van viszont ezek a függőségi fában levő szülőjükön a *xtext.ui* plug-in-en csupán opcionálisként vannak megjelölve és egy mai napig nem tisztázott ok miatt nem töltődtek be. A problémát azzal lehet orvosolni, hogy készítünk az xtext.ui plug-in-ből egy saját példányt, amin a függőségi beállításokat átírjuk opcionálisról kötelezőre. Ez sajnos nehezíti a rendszer tovább fejlesztését és a telepítési folyamat is komplexebbé válik.

Egy kulcsfontosságú probléma az volt, hogy az Xtext[4] modult eddig nem tudtuk injektálni a headless eclipse-be. Erre az alábbi kódrészlet ad megoldást:

```
  Injector injector = new StatechartLanguageStandaloneSetupGenerated()
              .createInjectorAndDoEMFRegistration();
  XtextResourceSet resourceSet = injector.getInstance(XtextResourceSet.class);
```

A fent definiált *resourceSet*-ről tudjuk majd a headless eclipse-nek argumentumban átadott fájlt elérni.

Minden Gamma nyelv interpretálóját egyedileg meg kellet hívni az alkalmazás indulási pillanataiban:

```
//Alkalmazás belépési pontja
 @Override
public Object start(final IApplicationContext context) throws Exception {
  ExpressionLanguageStandaloneSetup.doSetup();
  ActionLanguageStandaloneSetup.doSetup();
  StatechartLanguageStandaloneSetup.doSetup();
  TraceLanguageStandaloneSetup.doSetup();
  PropertyLanguageStandaloneSetup.doSetup();
  GenModelStandaloneSetup.doSetup();
  .
  .
  .
```

OpenAPI REST webszerver Olyan fejlesztői megoldásokat mutatok be, amelyek az OpenAPI és Vert.x használata által nagyon egyszerűen megvalósíthatóak voltak. Ehhez tekintsük át a 3.3.2 ábrát amely az *addproject* endpoint specifikációját tartalmazza.

A Vert.x, mint említettem támogatja az OpenAPI REST specifikáció integrációját, így kódból az alábbi módon tudjuk elérni a fájlban definiált *endpointot* (gamma-wrapper.yaml), amit majd regisztrálunk a webszerverünkbe:

```
@Override
public void start(Future<Void> future) {
        OpenAPI3RouterFactory.create(this.vertx, "gamma-wrapper.yaml", ar ->
    {
                OpenAPI3RouterFactory routerFactory = ar.result();
                routerFactory.addHandlerByOperationId("addProject", routingContext -> { ... }
    .
    .
```

A fenti kódrészletben már nagyon egyszerűen tudjuk a kérésben utazó paramétereket Java objektummá átalakítani.A példában fájlok feldolgozását is el kell végezzük, ezt a Vert.x a fenti *routingContext* objektumon már eltárolja és egy egyszerű *routingContext.fileUploads()* hívással már *File* objektumok listáját kapjuk vissza.

Bizonyos validációs feladatokat is elvégez helyettünk az OpenAPI definíció alapján a Vert.x, a 3.3.2 ábrán látható, hogy meghatározzuk a munkatér(workspace) szintaktikáját, ha ettől eltérő, más formátumú karakterlánc szerepel a kérésben akkor automatikusan küldődik egy válasz a klienshez, hogy nem megfelelő a kérése.

[4]Xtext-ről többet: `https://www.eclipse.org/Xtext/`

```yaml
/gamma/addproject/{workspace}:
  post:
    operationId: addProject
    description: Send a file that contains the eclipse project on which the gamma operations will run
    parameters:
      - in: path
        name: workspace
        required: true
        description: The workspace which contains the project
        schema:
          type: string
          format: uuid
          example: 3fa85f64-5717-4562-b3fc-2c963f66afa6
    requestBody:
      content:
        multipart/form-data:
          schema:
            type: object
            properties:
              contactEmail:
                type: string
              file:
                type: string
                format: binary
    responses:
      200:
        description: Successfully uploaded the file
      401:
        description: Did not provide a valid workspace
      403:
        description: Project already exists under this workspace, delete it and resend this request
```

3.3.2. ábra. Endpoint példa

ProcessBuilder fejlesztés Alapvetően a webszerver java 8 alapokon lett megvalósítva, majd kiderült, hogy a java 9 hasznos, új frissítéseket tartalmaz a *java.lang.ProcessBuilder* komponensén. Számunkra ez azért fontos, mert bevezetésre került a *pid* (process id) lekérése az éppen indított folyamatról. Ezt az azonosítót feltudjuk használni, hogy egy Gamma műveleteket feldolgozó folyamatot leállítsunk, ha úgy gondoljuk, hogy túl sok ideig fut. Nagyon fontos megjegyezni azt, hogy ha folyamatot indítunk a *java.lang.ProcessBuilder* segítségével és ez több ideig fut, akkor a logolási eredményét kötelezően át kell irányítsuk fájlba vagy konzolba. Ennek hiányában a folyamatunk beakad. A rendszerünk a folyamatot indító konténerre leöröklteti a logolás feladatát. A HeadlessGamma elindítására szolgáló kódrészlet:

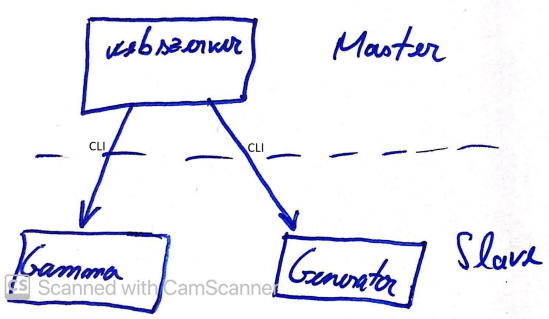
```
  ProcessBuilder pb = new ProcessBuilder(commandToBeExecuted); // ProcessBuilder inicializálás,
     commandToBeExecuted a futtatni kívánt parancs karakterláncát tartalmazza
  pb.redirectErrorStream(true);
  pb.inheritIO(); // Logolás öröklés
  long pib =  pb.start().pid(); // Folyamat azonosító lekérése
}
```

3.3.2. Rendszer szoftverkomponensei

A rendszer struktúrájából kiindulva, ezt a 3.2.1 ábrán lehet megtekinteni, a rendszer szoftver szintű felépítése három komponensből tevődik össze, ezek vizuális reprezentációját a 3.3.3 ábra mutatja be.

A Webszerver tartalmazza az összes olyan komponenst, amely a webes kérések értelmezését és átalakítását végzik, itt található az OpenAPI definíció. Önmagában egy Maven[5] alapú InteliJ IDEA-ban fejlesztett alkalmazás. A Gamma és a Generator pedig

[5]Maven-ről többet: `https://maven.apache.org/`



3.3.3. ábra. Szoftver komponensek

az eddigiekben már többször említett két headless eclipse alkalmazás. A webszerver egy master szerepet tölt be a rendszerünkbe, mivel ö koordinálja a másik kettő működését.

A szoftverkomponensek forráskódja megtekinthető:

- Webszerver `https://github.com/szlnnn/gamma-openapi-vertx/tree/dev`
- Gamma `https://github.com/szlnnn/gamma/tree/dev/plugins/headless/hu.bme.mit.gamma.api.headless`
- Generator `https://github.com/szlnnn/eclipse-headless-project-import-tool/tree/dev`

4. fejezet

Használati útmutató

Képekkel bemutatom a rendszer használatát , az eredményről képet mutatni ?

5. fejezet

Összefoglalás

miről volt szó, mi valósult meg és mit lehet tovább fejleszteni a megvalósulásnál kiemelem a követelmény ábrát és rárajzolok mi valósult meg és mi nem

Irodalomjegyzék

[1] Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar: Diplomaterv portál (2011. február 26.). http://diplomaterv.vik.bme.hu/.

[2] James C. Candy: Decimation for sigma delta modulation. 34. évf. (1986. 01) 1. sz., 72–76. p. DOI: 10.1109/TCOM.1986.1096432.

[3] Gábor Jeney: Hogyan néz ki egy igényes dokumentum? Néhány szóban az alapvető tipográfiai szabályokról, 2014. http://www.mcl.hu/~jeneyg/kinezet.pdf.

[4] Peter Kiss: Adaptive digital compensation of analog circuit imperfections for cascaded delta-sigma analog-to-digital converters, 2000. 04.

[5] Wai L. Lee – Charles G. Sodini: A topology for higher order interpolative coders. In *Proc. of the IEEE International Symposium on Circuits and Systems* (konferenciaanyag). 1987. 4-7 05., 459–462. p.

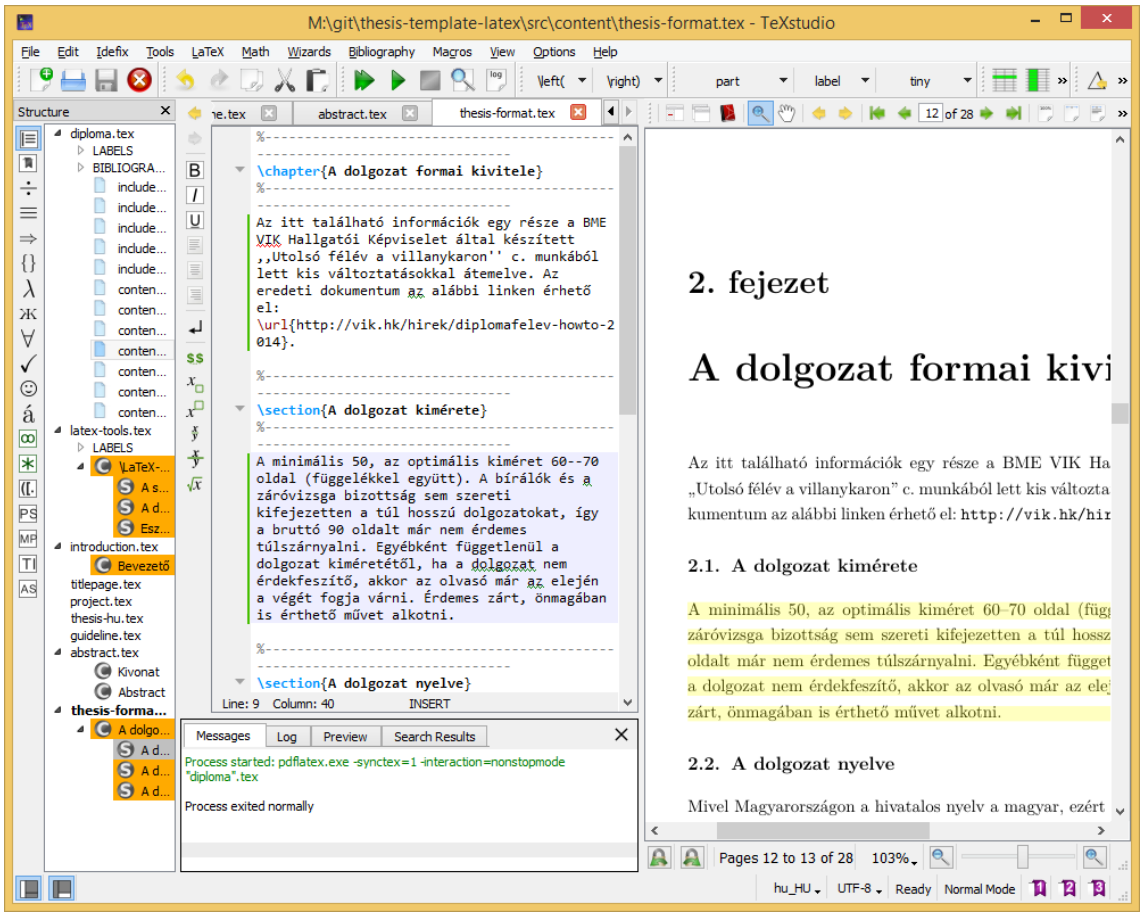
[6] Alexey Mkrtychev: Models for the logic of proofs. In Sergei Adian – Anil Nerode (szerk.): *Logical Foundations of Computer Science*. Lecture Notes in Computer Science sorozat, 1234. köt. 1997, Springer Berlin Heidelberg, 266–275. p. ISBN 978-3-540-63045-6. URL http://dx.doi.org/10.1007/3-540-63045-7_27.

[7] Richard Schreier: *The Delta-Sigma Toolbox v5.2*. Oregon State University, 2000. 01. http://www.mathworks.com/matlabcentral/fileexchange/.

[8] Ferenc Wettl – Gyula Mayer – Péter Szabó: *LaTeX kézikönyv*. 2004, Panem Könyvkiadó.

Függelék

F.1. A TeXstudio felülete



F.1.1. ábra. A TeXstudio LaTeX-szerkesztő.

F.2. Válasz az „Élet, a világmindenség, meg minden” kérdésére

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42. \tag{F.2.1}$$

A Faraday-indukciós törvényből levezetve

$$\operatorname{rot} E = -\frac{dB}{dt} \qquad \longrightarrow \qquad U_i = \oint_{\mathbf{L}} \mathbf{Edl} = -\frac{d}{dt} \int_{A} \mathbf{Bda} = 42. \tag{F.2.2}$$