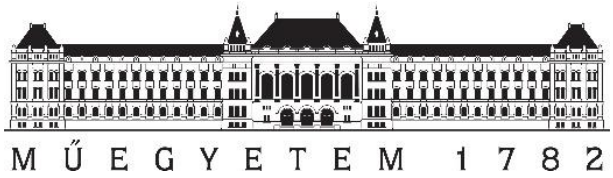


Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Sipka Bence

Modulárisan kiterjeszthető fordítás automatizálás

KONZULENS

Dr. Ekler Péter

BUDAPEST, 2017

Tartalomjegyzék

Összefoglaló

A szoftverfejlesztési folyamatok során, kis- és nagy projektek esetén is előfordulhat olyan probléma, melyre a hatékony megoldást a fordítási lépéseken való módosítás vagy optimalizálás adhatja. Ilyen esetben olyan fordítói rendszerre van szüksége a fejlesztőnek, mely ezt könnyen konfigurálható módon teszi lehetővé, míg mellette ugyanazokat a szolgáltatásokat is nyújtja, amiket a jelenleg piacon fellelhető elterjedt fordítói eszközök is. A dolgozat egy általam készített, erre a problémára nyújtott modulárisan kiterjeszthető fordító rendszer tervezését, felépítését, megvalósítását és használatát mutatja be.

A dolgozatban bemutatott megoldásom fő tulajdonsága, hogy a fordítás során elvégzendő lépéseket, a jelenleg elterjedt függőség alapú leírásoktól eltérően, szekvenciálisan van mód megadni. Ez egyrészt a programozó számára könnyebb áttekinthetőséget biztosít, mivel minden folyamatnak meghatározott lefutási sorrendje van, másrészt különböző rendszerekkel ellentétben, nem határoz meg a fejlesztőnek egy kötelezően követendő projektstruktúrát.

A kiterjeszthetőség biztosításával lehetőség van a fejlesztő által létrehozott komplex lépések definiálására, mely során tetszőleges feladatok elvégezhetőek fordítási időben. A rendszer ehhez igazodva egy olyan lazán csatolt szerkezet alapján működik, melyben dinamikusan van lehetőség különböző modulok betöltésére. Ennek megoldására a környezet a Java nyelv és virtuális gép lehetőséget használja ki, mely kiválóan támogatja kódok futásidejű betöltését.

A fordítási lépések megadásához a rendszer számára egy különálló nyelvet definiáltam, melynek segítségével leírhatóak, és konfigurálhatóak az elvégzendő feladatok. A nyelv által leírható adatszerkezetek, és a példányosított feladatokat elvégző modulok bemenetei között a rendszer automatikus konverziót biztosít, mellyel egy gyengén tipizált környezetet ad.

A fejlesztők támogatásának érdekében a rendszer rendelkezik *Eclipse* integrált fejlesztői környezethez tartozó kiterjesztéssel, mely támogatja az általam kidolgozott nyelv alapján történő fordítást. A fordítás után a kiterjesztés a rendelkezésre álló információk alapján úgy módosítja a lefordított projekt tulajdonságait, hogy a fejlesztőnek lehetősége legyen a beépített funkciók használatára (pl.: *Content Assist*).

Jelen TDK dolgozat összehasonlítja az általam kidolgozott fordítás támogató rendszert a piacon lévő kész megoldásokkal, és taglalja a tervezés során felmerülő kérdéseket. Részletesen bemutatja a rendszer tervezését, felépítését, használatát, és ismerteti a moduláris kiterjeszthetőség módszereit. Zárásként mérési eredmények bizonyítják az elkészült rendszer gyorsaságát és skálázhatóságát.

Abstract

During the course of software development, such problems can surface, that the efficient solution is to modify or optimize the compilation steps in order to produce the result. For either small or big projects, in such cases the developer requires a build system which allows easy configuration of the build pipeline, while providing the same features as the state of the art build systems on the market. This thesis presents the structure, development and usage of such modularly extensible build system.

Apart from the current trends, the main attribute of the system is that the build steps can be specified sequentially. Due to this nature, it is easier to overview the build process of the developed software, because every part of the compilation has a definite run order, unlike the dependency based description, which could force the developer into a fixed project structure, or the build steps could be hard to trace on the first look.

With the promise of extensibility, the developer has the possibility of creating complicated build steps, which can complete the specified tasks during compilation. In order for this to work the system operates in a loosely attached way, where it is possible to load different modules dynamically. To achieve this, the build system uses the features of the Java language and its virtual machine to support loading executable code during runtime.

The system defines its own scripting language, where it is possible to describe and configure the build steps appropriately. Data structures defined in the script are automatically converted to the inputs of the corresponding modules, which results in a weakly typed system.

With the support of the developers in mind, the system offers a plugin for the *Eclipse* integrated development environment, which support script based build execution. After the compilation, the plugin processes available information, and modifies the *Eclipse* project structure, in order to make available such features to the developers, that makes software development easier. (E.g. *Content Assist*).

This thesis compares the developed build system with the ones currently available solutions on the market, and outlines the emerging implementation questions. It provides details of the structure for the system, describes the methods of creating modular extensions, and provides examples for the system usage. The presented measurements will prove the performance and scalability of the working solution.

1 Bevezetés

A dolgozat témáját egy olyan szoftverfejlesztési probléma felmerülése határozta meg, melynek megoldása nem triviális. Egy több platformon (Android, iOS, UWP) futó alkalmazás elkészítéséhez fordítási időben kellett generálni olyan fájlokat, melyek a cél operációs rendszerre való fordítás bemenetét képezik. A generálást úgy kellet elvégezni, hogy az különböző rendszereken is futtatható legyen, mivel az elkészült alkalmazásokat nem volt lehetőség egy operációs rendszer alatt fordítani.

A problémából adódóan első megoldásként egy olyan monolitikus Java alapú eszköz került kifejlesztésre, mely a generálást megfelelően teljesítette. A program meghatározott feladatokat végzett el, azonban újabb igények felmerülése esetén ennek bővítése egyre nehezebbé vált a szorosan kapcsolt felépítése miatt. További fejlesztés során a különböző részek külön lettek választva, és második megoldásként egy olyan lazán csatolt rendszer jött létre, melyben a feladatok egymástól függetlenek, és azok futtatását egy központi kód vezényli.

Míg az újabb generálási lépések hozzáadása így könnyebb lett, a rendszer még mindig nem nyújtott önállóan használható fejlesztési megoldást. A generálásnak ugyanúgy a platform fordítói előtt kellett lefutnia, ezzel egyrészt meggátolva az inkrementális fejlesztést, másrészt a forráskódot minden exportálás során frissíteni kellett a megfelelő integrált fejlesztői környezetekben.

A fenti problémából adódóan a dolgozat egy olyan rendszert mutat be, mely lehetőséget ad egy-egy alkalmazás fordításának teljes levezénylésére, illetve nyújtja azokat a funkciókat, melyeket a programozó a napi munkája során elvár. Az inkrementális támogatás segítségével egy forrásfájl módosításával nincs szükség az egész projekt újrafordítására, illetve lehetőség van egy számítógépről más rendszereken való fordítás vezénylésére, mellyel sok idő takarítható meg.

A fordítási folyamat lépései szkriptfájlok segítségével írhatóak le, melyek megadják, hogy a fordítás során milyen bemenetek, és konfigurációk alapján készüljön el az eredmény. Míg a főbb fordítói rendszerek a leírást szabály alapon, deklaratívan követelik meg, az elkészült rendszer ezt szekvenciálisan teszi meg, így egy átláthatóbb, és könnyebben bővíthető leírásra ad lehetőséget.

2 Irodalomkutatás

A feladat melyet el kellett végezni a felhasználás módjából adódott, így a fejlesztés meghatározott követelmények alapján történt. Későbbiekben ez egy általános fordítást támogató rendszerré fejlődött, azonban mielőtt egy már létező megoldás kerül újra implementálásra házon belül, meg kellett vizsgálni, hogy milyen már elérhető megoldások léteznek, melyek kielégítik a felmerülő problémát.

Fő követelményként merült fel, hogy a fordítás több operációs rendszeren is fusson, és általános célú legyen, melyből adódóan különböző Java alapú automatizálási rendszerek kerülhettek felhasználásra (*Apache Ant, Apache Maven, Gradle*), illetve más *cross-platform* megoldások, melyek megadják a bővíthetőség lehetőségét (*Make* alapú, illetve egyéb piaci megoldások).

A fenti követelmény teljesítésére nem találtam olyan megoldást, mely *plug & play* módon kiválthatná a különböző platformokra való fordítást, és megadja az egyszerű bővíthetőség lehetőségét. Ennek eredményeként egy parancssorból futtatható Java program készült, mely a megfelelő *IDE*-ben (*Integrated Development Environment)* hozzáadható volt a fordítási lépésekhez, így annak használata segítette a napi fejlesztést.

> Jellemzően abból kifolyólag nem találtam megoldást, hogy a megfelelő általános célú automatizáló szoftver nem rendelkezett a megfelelő kiterjesztéssel, mely elvégzi a fordítás vezénylését az összes célplatformra. Ennek hiányában, nincs olyan rendszer mely helyettesíti a fejlesztőkörnyezetekben történő fordítást a kívánt operációs rendszerekre.

A későbbiekben elkészült általános célú eszköz gyorsabban végezte el a fordítási feladatokat, mint a vizsgált megoldások. A rendszerhez készült nyelv segítségével tovább csökkenthető a fejlesztők fordítási fájlok karbantartásával töltött ideje, mely megelőző tanulmányok alapján az idejük 12%-át öleli fel [1].

2.1 Általános célú automatizáló szoftverek

A dolgozat témájaként egy olyan rendszer készült el, mellyel a fordítás tetszőlegesen konfigurálható, így ebből következően a piacon található megoldásokkal szembeni összehasonlítás nem hagyható el.

Az automatizáló szoftverek egyik fő tulajdonságuk alapján, hogy milyen módon írhatóak le bennük a fordítási lépések, két kategóriába sorolhatók:

A függőség alapú leírás során általában deklaratívan határozhatóak meg az elvégzendő feladatok, ami szerint a program egy gráfot készít, és ez alapján vezényli a fordítást. Ez a módszer jó megoldást nyújt az inkrementális fejlesztés támogatásához, azonban esetlegesen

körülményesen írható le, és a rendszer mélyreható ismeretét igényli. (pl.: *Apache Ant, Apache Maven, Gradle*)

A szekvenciális leírás során az intuíciónak megfelelően kerülnek felsorolásra az elvégzendő feladatok, melyek egymás utáni sorrendben lesznek végrehajtva. A fordító rendszer ilyenkor gyakran nem végez inkrementális vizsgálatot, hiszen előfordulhat, hogy a lépések ki és bemenetei előre nem meghatározottak a koordinátor (fordítás irányító) számára. Az ilyen rendszerek bővítése általában egyszerű, azonban a fordítás gyakran sokkal több időt vehet igénybe. (pl.: *Shell* szkriptek)

A fordítók további meghatározó tulajdonságai, hogy milyen mértékben teszik lehetővé a külső függőségek hozzáadását a szoftverhez, mennyire támogatják a fejlesztőkörnyezetek a használatukat, illetve milyen módon bővíthetőek.

2.1.1 Apache Ant

Az *Apache Ant* [2] egy 2000-ben kiadott, és mai napig fejlesztett Java alapú automatizáló szoftver, melynek fő felhasználási területe Java alapú alkalmazások készítésében mutatkozik, azonban van lehetőség más nyelvek használatára is (pl.: C/C++). Az *Ant* lehetőséget ad bármilyen feladatok elvégzésére, melyek leírhatóak célok (*target*) és feladatok (*task*) összességeként [3].

A feladatok megadására *XML* (*Extensible Markup Language*) formátum használatával van lehetőség, melyben deklaratív módon lehet leírni a lépéseket. A függőségek meghatározásával támogatja az inkrementális fordítást, és rendelkezik fejlesztőkörnyezetekhez hozzáadható kiterjesztésekkel (pl.: *Eclipse*[1], *IntelliJ*[2], *NetBeans*[3]). Eredetileg nem alkalmazott függőségkezelést, azonban később kiterjesztésként hozzá lett adva az *Apache Ivy*[4] kezelő támogatása.

Az *Ant* fő előnye, hogy erős kontrollt ad a futó folyamatok felett, azonban a fordítást leíró *XML* a projekt méretének növekedésével naggyá, és nehezen kezelhetővé válhat.

2.1.2 Apache Maven

Míg az *Ant* direkt felügyeletet adott a lefutó folyamatok felett, addig az *Apache Maven* [4] egy olyan konvenciókra épülő megoldást nyújt, melyben a fordítás előre meghatározott lépések alapján zajlik. Kezdetektől fogva támogatást nyújt a külső függőségek kezelésére, melyet

[1] Eclipse - https://www.eclipse.org/
[2] IntelliJ IDEA - https://www.jetbrains.com/idea/
[3] NetBeans - https://netbeans.org/
[4] Apache Ivy - http://ant.apache.org/ivy/

szorosan beépít a fordítási folyamatba. Az *Ant*-hoz hasonlóan *XML*-ben van lehetőség a lépések függőség alapú leírására, mely magával hordozza az abból adódó hátrányokat is.

A *Maven*-hez tartozó kiterjesztések, fejlesztőkörnyezet támogatás, és az azt használó közösség jelenléte megkönnyítik a fejlesztést, azonban az egyedi feladatok hozzáadása nehézkessé válhat a projekt méretének növekedésével.

2.1.3 Gradle

A *Gradle* [5] az előzőektől eltérően saját nyelvet használ a folyamatok leírására, és fő fordítójává vált az *Android*[5] operációs rendszerre való fejlesztésnek. A nyelv használatával szintén deklaratívan lehet megadni a függőségeket, azonban van lehetőség szekvenciális részletek leírására is a *Groovy* nyelv segítségével [6]. Jelentős fejlesztői környezet támogatással rendelkezik (pl.: *Eclipse, IntelliJ, NetBeans*), támogatja az *Ivy* és *Maven* függőségkezelőket, és lehetőséget ad kiterjesztések készítésére.

Az fordítást lépésekre bontja fel, melyeket csak akkor végez el, ha azok bemenetei megváltoztak, így jó inkrementális támogatást nyújt. Érdemes megemlíteni, hogy a Java nyelv inkrementális fordítását kiválóan támogatja a 3.4-es verziótól kezdődően [7], melyet *bytecode* analízissel, és *ABI* (*Application Binary Interface*) változások keresésével old meg.

Az eszköz hosszú betöltési idővel rendelkezik, azonban erre a *Gradle Daemon* nyújt megoldást, mely egy háttérben futó folyamat, ami elvégzi az adatok gyorsítótárazását és végrehajtja a hozzá irányuló kéréseket, ezzel kiiktatva a többszöri kezdeti inicializálást.

Ezek alapján a *Gradle* egy sokoldalú eszköz, mely különféle projektek fordítását is el tudja végezni, azonban egy technológiával ismeretlen programozónak elsőre nehéz lehet a feladatok átlátása és a komplexebb testreszabás.

2.1.4 *Shell* szkriptek

Egyszerűbb esetekben a fordítási folyamatot elég más eszközök egymás utáni végrehajtásával leírni, melyre az operációs rendszerekbe beépített parancsvégrehajtók segítségével van lehetőség. Ez a megoldás könnyen bővíthető, és átlátható, azonban sok olyan funkciót nem biztosít, melyet a modern szoftverfejlesztés elvárhat.

Korlátozottan van lehetőség inkrementális támogatásra, így a programozónak gyakran végig kell várnia az egész fordítási folyamatot, mely már közepes méretű projektek esetén is hosszú időbe telhet. Függőségkezelésre általában nincs lehetőség, azonban megfelelő implementáció esetén egyes alfolyamatok elvégezhetik azt. Fejlesztői környezet támogatás minimális, a

[5] Android - https://www.android.com/

programozónak manuálisan kell beállítania a projekt tulajdonságait, mivel azok a fordítási folyamat végén nem állnak rendelkezésre.

Továbbgondolt változata ennek a *Make* [8], mely hozzáadja a célok fogalmát, így deklaratív függőségeket lehet meghatározni fájlok között. Ennek hátránya, hogy a programozónak explicit kell ismernie a ki és bemeneti fájlokat a megfelelő konfiguráláshoz. A fordítás párhuzamosítását a programozónak manuálisan kell elvégezve, ezzel is több feladatot róva rá. A *Make* fájlok leírása idővel bonyolulttá válhat, mely jelentősen megnehezíti nagy projektek esetén a további bővítést [9].

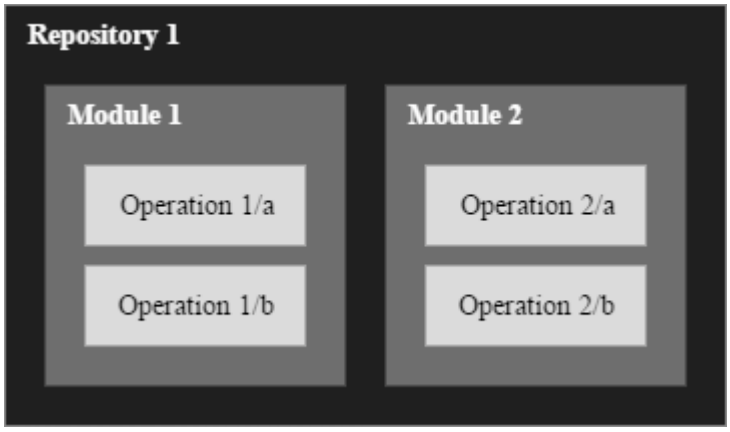
2.2 Konklúzió

A vizsgált és piacon elterjedt megoldások alapján a programozónak a függőség alapú és inkrementális, illetve a szekvenciális, egyszerűbben áttekinthető, de hosszabb fordítási idejű megoldások közül kell választania.

Ebből adódóan felmerül a lehetőség egy olyan rendszer fejlesztésére, mely támogatja az egyszerű szekvenciális leírást, miközben ugyanúgy nyújtja a függőség alapú megoldások előnyeit. A dolgozat egy ilyen automatizáló eszköz felépítését és fejlesztési kérdéseit mutatja be, végül pedig gyakran előforduló fordítási példákon végzett fordítási mérésekkel igazolja működését.

3 Áttekintés

A rendszer készítésének céljából adódóan, a felépítését a kiterjeszthetőség lehetőségének támogatása határozza meg. Ez alapján három alapvető alkotórészre bontható fel a program, melyek biztosítják a modularitást, és feladatuk, hogy bezáró egységet alkossanak a beljebb lévő részek számára.



1. ábra: Felépítés

3.1 *Repository*

A legmagasabb szintű egységet a *Repository* képezi, melynek feladata, hogy egységbe zárja a benne található modulokat, és igény esetén példányosítsa őket. A rendszer egy interfész implementálásán kívül nem határoz meg semmilyen követelményt egy-egy *Repository*-val szemben, így azok tetszőleges módon végezhetik a befoglalt entitások azonosítását és létrehozását. Futási időben mindegyik egy-egy azonosítóval rendelkezik, melyeknek globálisan egyedinek kell lenniük.

A *Repository*-k rendszerbe való betöltésére JAR (*Java ARchive*) fájlokon keresztül van lehetőség, mely lehet akár lokális gépen, akár interneten keresztül is elérhető (ebben az esetben automatikusan letöltődik).

3.2 Module

A modulok feladata, hogy egységet képezzenek az általuk reprezentált funkciók felett, egységbe foglalják az általuk definiált operációkat, és a kapott paraméterek alapján létrehozzák a végrehajtandó operációk által elvárt környezetet. A megfelelő erőforrások lefoglalását és felszabadítását a két fő életciklus függvényben végezhetik el (*open, close*).

Minden modul egyedi azonosítóval rendelkezik, melyek alapján meg lehet őket határozni a fordítást leíró szkriptből. Egy-egy azonosítónak csak a tartalmazó *Repository*-n belül szükséges egyedinek kell lennie, azonban érdemes globálisan is erre törekedni. A szkriptből való

hivatkozás során lehetőség van az azonosítókhoz ’minősítőket’ hozzáadni, mellyel megvalósítható a megfelelő verziókezelés, és egyéb statikus tartalomválasztás.

3.3 Operation

A fordítás során az operációk végzik el a programozó által leírt feladatokat, melyek példányosításáért szintén a befoglaló *Repository* felelős. Az operációk jelentik a legkisebb egységet a rendszerben, így ők a modulokkal ellentétben nem rendelkeznek életciklus függvényekkel, mivel futásuk után egyből megsemmisülnek.

Az operációk egyszerű azonosítóval rendelkeznek, melyek az őket befoglaló modulokon belül egyediek.

3.4 Futtatókörnyezet

A futtatókörnyezet feladata, hogy beolvassa a programozó által definiált szkriptet, leképezze azt fordítási lépésekre, majd végrehajtsa. A környezet konfigurálása az indításkor hajtódik végre, és ebben futnak a definiált feladatok. A program biztosítja a hozzáférést az operációs rendszer védett erőforrásaihoz (fájl olvasás, írás, futtatás), formázza a ki- és bemenetet, vezényli a Java osztályok betöltését, illetve absztrakciós réteget nyújt a fájlrendszer felett.

A rendszer felépítését parancssori hívás esetén minden alkalommal fel kell építeni, azonban *IDE* kiterjesztés vagy háttérben futó *Daemon* használatával, ezt nem szükséges mindig elvégezni, a betöltött környezet újra felhasználható.

3.5 Fordítószkript

A végrehajtandó műveletek egy szkript segítségével írhatók le, melyben szekvenciális módon a modulok és operációk végzik el a feladatokat. A konfiguráció könnyítésének érdekében lehetőség van változók definiálására, melyek akár egyszerű, akár komplex struktúrát is alkothatnak. Ezek között a változók között automatikus konverziók vannak definiálva, így könnyen átalakítható egy-egy szkriptbeli változó a Java oldalon definiált típusokká.

A szkriptben lehetőség van kontrollszerkezetek használatára is, melyekkel a feladatok ki- és bemeneteinek feldolgozása, illetve konfigurálása egyszerűbben megoldható.

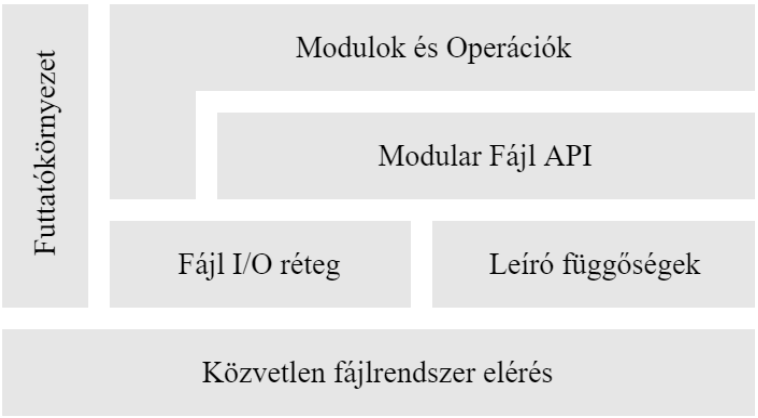
A szkriptek legfelső egysége a cél (*target*), mely összefogja az előzőleg említett utasításokat. Ezek segítségével van lehetőség másik szkriptekben található célokra való hivatkozásra, így különböző fordítási lépések szemantikailag elkülöníthetőek.

3.6 Absztrakt fájlrendszer

Két okból kifolyólag is érdemes egy réteget bevezetni a fájlrendszer közvetlen elérése és a felhasználói kód közé. Egyrészt a fordítás során a generált fájlokat nem minden esetben kell a

fájlrendszerre közvetlenül kiírni, így felesleges *Input-Output* műveleteket lehet megspórolni. Másrészt az inkrementális fordítás támogatásának érdekében ez a réteg biztosítja azt, hogy csak akkor kerülnek a változások végrehajtásra, ha a megfelelő bemeneti adatok megváltoztak.

Ennek érdekében a modulok és operációk egy olyan rétegen keresztül látják a fájlokat, melyben azokat egy leíróval kell ellátni, ami egyértelműen definiálja a fájlban található adatokat. Az újonnan generált fájlok csak akkor lesznek kiírva a háttértárra, amennyiben az új leíró az előzőleg perzisztálthoz képest megváltozott. Ezzel a módszerrel könnyen lehet meghatározni fájlok közötti függőségeket, így csökkentve a fordítási időt.



2. ábra: Fájl hozzáférés

A 2. ábra bemutatja, hogy a modulok és operációk elsősorban az absztrakt *API*-t (*Application Programming Interface*) használhatják, így biztosítva az inkrementális alrendszer működését, azonban esetekben van lehetőség direkt hozzáférésre is a fájlokhoz. A Fájl I/O réteg biztosítja, hogy a fájlok a hálózaton keresztül is elérhetőek legyenek, illetve a tartalomleírók (Leíró függőségek) segítségével végzik a perzisztálást.

Az absztrakt fájlrendszer ezen kívül lehetőséget nyújt olyan fájlok mappaként való reprezentálására, mely valójában nem azok (pl.: *ZIP* archívumok), így egyszerű hozzáférést ad becsomagolt fájlokhoz is.

Fontos művelete a rétegnek a ’kicsomagolás’, mely során egy-egy fájl tartalma a lokális gép rendszerén kerül elhelyezésre, így lehetőség van azokat más folyamatok bemeneteként felhasználni.

> Erre fontos felhasználási példa lehet a C++ fordítás során a fájlok elérhetővé tétele. Amennyiben a fordítandó fájl a hálózaton található, archívumban van becsomagolva, vagy még nincs kiírva a háttértárra, abban az esetben a művelet elhelyezi azt a fordítási könyvtárban, azt külső folyamatok számára elérhetővé téve.

3.7 Daemon

Egy-egy fordítás végrehajtásához sok esetben több különböző *Repository*-t, és modult kell betölteni, melyek jelentősen megnövelhetik az inicializálás idejét. Mivel a futtatókörnyezet és a futtatandó feladatok egymástól elkülöníthetőek, ennek megoldására lehetőség van egy már létrehozott környezet többszöri felhasználására.

A *Daemon* feladata, hogy egy háttérben futó eszközként életben tartja a fordításhoz szükséges alapvető szerkezeteket, és gyorsítótárként funkcionálva lehetőséget nyújt az inicializációs idő csökkentésére. Egy-egy futó *Daemon*-t hálózaton keresztül is el lehet érni, ezzel megadva az alkalmat arra, hogy a fordítást más gépen végezhesse a fejlesztő.

4 Fordítószkript

A rendszer tervezésekor meg kellett határozni, hogy milyen nyelven lehessen a programozó által leírni a fordítási folyamatokat. Ennek a következő követelményeket kellett biztosítania:

- Szekvenciális nyelv,
- Illeszkedik a rendszer felépítéséhez,
- A lépések egyszerűen és intuitívan konfigurálhatók,
- Az utasítások könnyen leképezhetők futási lépésekké.

Mivel a fenti követelményeket kielégítő nyelvet nem találtam annak meghatározásakor, illetve nem szívesen hoztam volna be újabb függőségeket a projektbe, ezért a szkript nyelve egy újonnan definiált szakterület-specifikus nyelv (*Domain Specific Language* [10]) lett. Egy új nyelv bevezetése mellett szól az is, hogy egy feladatra érdemesebb egy arra megfelelően tervezett megoldást találni és speciális absztrakciós réteget szervezni rá, minthogy egy létező megoldást alakítsunk át a problémára, mellyel új nem kívánt hibákat hozhatunk a rendszerbe [11]. Míg egy új nyelv tervezése nagyobb kezdeti implementációs költséggel rendelkezik, addig hosszú távon könnyebben bővíthető, így jobban lehet hozzá felhasználásspecifikus funkciókat adni [12].

> Habár egy új nyelv a fordítási lépések alapvető leíró nyelve, a rendszer lehetőséget nyújt tetszőleges nyelvek használatára, amennyiben a fejlesztő betölti a rendszerbe a megfelelő feldolgozó osztályt, illetve a szkriptfájlok elején azt egy ’#’ karakterrel kezdődő sorral jelzi.

A fejezet a nyelv fő funkcióit, és a konfigurálás alapvető lehetőségeit mutatja be. A példák során a kifejezések pontosvessző karakterrel vannak lezárva, azonban ez a felhasználás során elhagyható.

4.1 Változók és adattípusok

Az adatok megfelelő kezeléséhez lehetőség van változók definiálására, melyek globálisan írhatók és olvashatók. A rendszerben lévő változókat a nevük alapján lehet elérni, a *PHP* nyelvhez hasonlóan a ’$’ jel használatával:

```
$Variable = ”Test”; // értékadás
$Variable = $Other; // lekérdezés és értékadás
```

1. példa: Változók

A ’$’ jel egy kifejezés derefereálását jelenti, mely kiértékeli a ’$’ előtt álló *bármilyen* kifejezést, majd annak megfelelő nevű változó értékével tér vissza. Ebből adódóan, többszörös derefereálást is meg lehet fogalmazni a szkriptben:

```
$RealVariable = ”Value”; // RealVariable értéke ”Value”
$Name = ”RealVariable”; // Name értéke legyen ”RealVariable”

$$Name = ”Modified”; // $Name nevű változó értéke legyen ”Modified”
//megyezik a következő kifejezésekkel:
$”RealVariable” = ”Modified”;
$($Name) = ”Modified”;
```

2. példa: Többszörös dereferálás

A nyelv háromféle adattípust definiál, mely lehet egyszerű, vagy összetett (lista és szótár). A modulok konfigurációja ezeknek az értéktípusoknak a felhasználásával lehetséges.

Az egyszerű adattípusok közé tartoznak a sztringek és számok, melyeknek nem rendelkeznek komplex belső struktúrával:

```
$Simple = Test; // $Var értéke ”Test”
$String = ”Szöveg”; // $String értéke ”Szöveg”
$Number = ”123”; // $Number értéke ”123”
$Escaped = Comma\,Separated; // $Escaped értéke ”Comma,Separated”
$QuoteEsc = ”He said: \”...\”” // $QuoteEsc értéke ”He said: ”...””
```

3. példa: Konstansok

Összetett adatszerkezet egyszerűbb fajtája a lista:

```
$IntList = [1, 2, 3];
$MixedList = [4, Aladár, $IntList];
```

4. példa: Listák

A listák elemeit szögletes zárójelet között kell felsorolni, vesszővel elválasztva. A listák elemei bármilyen kifejezések lehetnek, melyek a lista létrehozásának pillanatában kerülnek kiértékelésre. A listák Java oldalon *List<Object>* példánnyá képződnek le.

Fontos összetett adatszerkezet a szótár, mely a listához hasonlóan bármilyen kifejezések értékét tárolhatják:

```
$SimpleMap = { key: value, abc: xyz };
$MixedMap = { item: $SimpleMap, value: [1, 2, 3] };
```

5. példa: Szótárak

Ez Java oldalon *Map<Object, Object>* példánnyá képződik le, azonban ezeknek a típusoknak a konverzióját automatikusa kezeli a rendszer. (Lásd:5.2 Adatkezelés)

Az összetett adatok mezőinek az eléréséhez a megfelelő objektumon való szögletes zárójellel határolt kifejezéseket használhatjuk (*index* operátor):

```
$Map = { key: value, abc: xyz };
$List = [1, 2, 3];
$JavaObject = ...; // Tetszőleges Java Object, modul által beállítva
$Value = $Map[key]; // $Value = value
$Integer = List[2]; // 0-tól indexelve $Integer = 3
$Field = $JavaObject[field] // $Field = ’JavaObject field mezője’
```

6. példa: Mező elérés

A mező elérés intuitívan működik a szkriptben definiált objektumokon, illetve Java oldalról beállított értékek esetén azoknak a megfelelő nevű mezői kerülnek lekérdezésre.

4.2 Kontroll szerkezetek

A legfelsőbb szintű szerkezet, ami egy definiálható az a cél (*target*), mely meghatároz egy-egy fordítási részfeladatot:

```
build: {
     ...
     // test cél meghívása, ebben a fájlban
     run test;
     // exportandroid cél meghívása platform.build fájlban
     run exportandroid@platform.build;
}
test: {
     ...
}
```

7. példa: Cél definiálása és hívása

A rendszerben két végrehajtási mód használatára van lehetőség, mely lehet szekvenciális, vagy párhuzamos. Szekvenciális végrehajtás esetén egy parancs csak akkor kerül futtatásra, ha az azt megelőző befejezte a műveleteit, míg párhuzamos futás esetén több szál egyszerre végzi el a műveleteit, majd a végén a szálakat bevárva halad tovább a folyamat. Ez fontos szerepet játszik a fordítás során, mivel a mai számítógépek rendszerint többmagos processzorral rendelkeznek, és azok kihasználása jelentősen lerövidítheti a fordítási időt.

Fontos, hogy bizonyos kódrészletek feltételesen futtathatók legyenek, így a nyelv *if-else* szerkezetek leírására is ad lehetőséget:

```
if $Condition: {
     // szekvenciális törzs
     ...
} else: [ // else elhagyható
     // párhuzamosan végrehajtott törzs
     ...
]
```

8. példa: *If*-else, párhuzamos és szekvenciális ágak

Az elágazás során a megadott feltétel kiértékelése után lesz kiválasztva a megfelelő lefutási ág. Amennyiben a kifejezés *null* értéket vesz fel, az *else* ág fog lefutni.

További lehetőségei a programozónak a ciklusok használata, mely során tetszőleges adathalmazon futtathat iteratív kódot:

```
foreach $value in [1, 2, 3]: {
     // 1, 2, 3 listán való végigiterálás value ciklusváltozóval
     ...
}
```

9. példa: Iterálás listán

```
$Map = { key: value, abc: xyz };
foreach $key, $value in $Map: {
     // Map-on való végigiterálás kulcs és érték ciklusváltozókkal
     ...
}
```

10. példa: Iterálás szótáron

A *foreach* szerkezet segítségével bármilyen szótáron, listán, Java tömbön, vagy Java *Iterable* interfészt megvalósító objektumon van lehetőség utasításokat végezni.

A más nyelvekből megszokott módon, feltételes ciklus is megadható a szkriptben:

```
while $Condition: {
     // ciklus törzs
     ...
}
```

11. példa: Feltételes ciklus

4.3 Modul és operáció hívás

A fordítás során a fő feladatokat a modulok és azok operációi végzi, melyet a programozó a következőképpen írhat le:

```
test.module(Parameter = $Value): {
     ...
     operation(Param1 = 1, Param2 = $List);
     ...
}
```

12. példa: Modul és operáció hívása

A paraméterek a moduloknak példányosításuk alkalmával, név szerint adhatók át. Amennyiben egy-egy paramétert nem adunk meg, abban az esetben a rendszer megpróbálja a paraméter nevének megfelelő globális változót értékül adni neki, ha sikerül.

A létrehozott modulok az életciklusuknak megfelelően, az utánuk definiált kontrollszerkezet lefutása után bezárulnak.

A paraméterlistában megadhatók kimeneti változók is, melyek akkor kerülnek kiértékelésre, amennyiben az utasítás megfelelően lefutott:

```
test.module: {
     operation(
          Param1 = $Input1,
          OutParam1 => $OutValue,
          OutParam2 +> $List
     );
}
```

13. példa: Kimeneti paraméterek

Kimeneti paraméterek a ’>’ karakterrel jelezhetőek, mely során a megfelelő művelet lesz végrehajtva a kifejezés eredményével. A példában az *OutParam1* értékül lesz adva az *$OutValue* változónak, illetve az *OutParam2* hozzá lesz fűzve a *$List* változóhoz.

A modulok példányosításakor, a névütközések elkerülése végett, lehetőség van *Repository* azonosítók, illetve operációk esetén *Module* azonosítók megadására:

```
test.module@repositoryid: {
     ...
     operation@test.module;
}
```

14. példa: Példányosítás azonosítók megadásával

Egy-egy szoftver életciklusa során gyakran többféle kiadás (*release*) készül, mely során szükséges lehet az előző verziókkal való kompatibilitás megőrzése. Annak érdekében, hogy a modulok azonosítóit kiadástól függően ne kelljen változtatni, lehetőség van a modul azonosítók minősítőkkel való ellátására:

```
test.module-v1.0-lhu: {
     // minősítők: [v1.0, lhu]
     ...
}
```

15. példa: Modul minősítők

Ilyenkor a példányosító rendszer megvizsgálhatja a minősítőket, és az alapján más modulokat hozhat létre. Ez a feladat a *Repository* implementációkra van bízva.

5 Belső felépítés

A rendszer belső felépítésére jellemző a lazán csatolt szerkezet, mely biztosítja azt, hogy futási időben könnyebben konfigurálhatóak egyes alrendszerek. Az alábbi fejezet ennek a részleteit mutatja be, kitérve az inkrementális fordítás támogatására, a szkripttel való kapcsolat létrehozására, bővíthetőség módjára, és adatvédelmi megoldásokra.

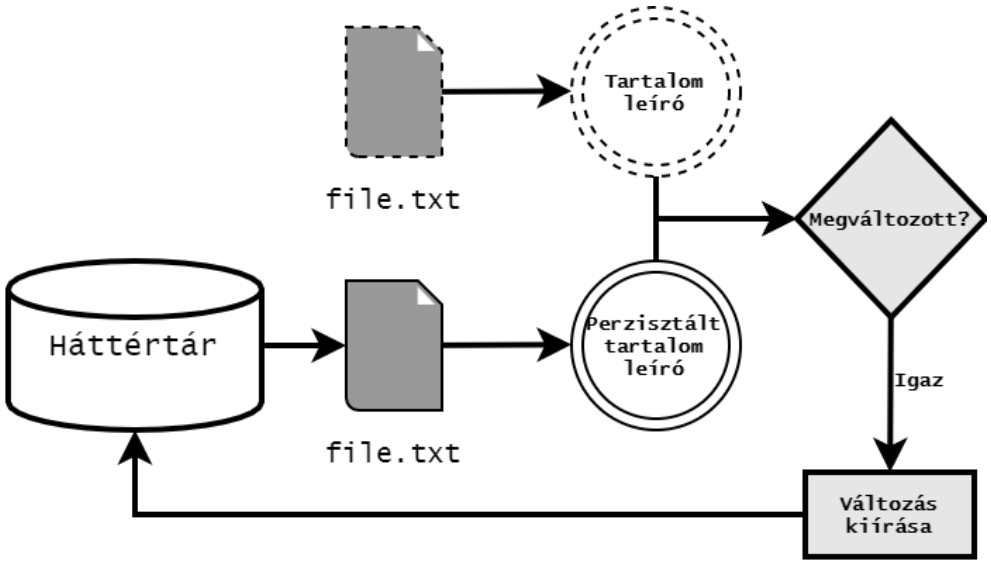
5.1 Inkrementális támogatás

A szoftverfejlesztési folyamatok során az inkrementális fordítás jelentése, hogy egy-egy fordítási lépés során nem az összes bemeneten hajtjuk végre a számítási feladatokat, hanem csak a megváltozottakon. A változások detektálása után felhasználhatjuk az előző fordítási folyamat kimenetelét, és ebben az esetben kevesebb idő alatt el tudjuk készíteni az eredményt. Ezzel a megoldással jelentősen lerövidíthetőek a kis és nagy projektek fordítására szánt idők egyaránt.

Az inkrementális fejlesztést támogató szoftverek feladata, hogy a lépések között felépítsen egy függőségi gráfot, melyhez ismernie kell a lépések közvetlen ki- és bemeneteit. A piacon elterjedt fordítók jelentős része így működik, és ebből adódóan egy deklaratív módszerrel lehet bennük leírni a fordítási feladatokat. Míg ez a módszer teljesen megállja a helyét, nem zárja ki egy olyan implementáció lehetőségét, mely nem szabály alapon közelíti meg a problémát.

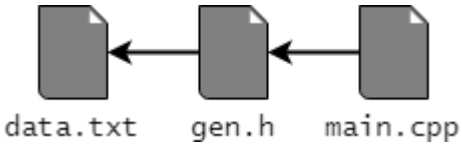
A megoldásomban két réteg kerül megkülönböztetésre:

1. Fájlrendszerre kiírt tartalom,
2. Fájlrendszer memóriában tárolt reprezentációja.



3. ábra: Inkrementális perzisztálás

A 3. ábra alapján, minden háttértáron eltárolt fájlhoz elmentünk egy tartalom leíró adatot, mely egyedien definiálja, hogy azon az útvonalon milyen tartalom található. Ennek eredményéül, csak akkor kell egy adatot perzisztálni, amennyiben a jelenleg háttértáron található adattól szemantikailag különbözik. Ezzel a módszerrel létrehozhatóak olyan adatfüggőségek, mellyel megspórolhatóak hosszú időt igénylő műveletek. A működés szemléltetését a következő példában vizsgálhatjuk meg:



4. ábra: Függőség példa

Három fájllal rendelkezünk, melyből a *data.txt* egy tetszőleges felhasználói fájl, a *main.cpp* pedig egy lefordítandó C++ forrásfájl. A *gen.h* egy olyan *header* fájl, melyet egy fordítási lépés során generálunk a *data.txt* állomány alapján. Ebben az esetben a generálás során létrehozunk egy olyan adatfüggőséget, mely a *data.txt* adatleírójára mutat. A *gen.h* háttértárra írásakor összehasonlítjuk a már kiírt adatleíróval a *gen.h* adatleíróját, és változás esetén a *gen.h* tartalmát perzisztáljuk. Észrevehetjük, hogy ilyenkor csak akkor fogjuk a *gen.h*-t kiírni a lemezre, ha a *data.txt* fájl tartalma megváltozott, vagy a lemezen még nem létezik a *gen.h* állomány. Ennek eredményéül, a várakozásunknak megfelelően, csak akkor fog a generálás (és a hozzá tartozó számítás) végrehajtódni, ha a bemeneti adat(ok) megváltoztak.

A *main.cpp* esetében a függőséget a *gen.h* állomány adatára határozzuk meg, így az tranzitívan függ a *data.txt* leírójától. Ebben az esetben a változások tovább fognak propagálódni a függőségi gráf mentén, és a hivatkozó adatok újra lesznek számítva.

A példához kapcsolódóan érdemes megemlíteni, hogy a C/C++ fordítók működéséből adódóan, egy *header* fájlra mutató függőség csak akkor határozható meg, ha a forrásfájlt már lefordítottuk. Ebből adódóan ezekre a forrásfájlokra úgy kell implementálni a két adatleíró közötti változás vizsgálatát, hogy amennyiben még nem állnak rendelkezésre *header* függőségek, abban az esetben mindenképpen le kell fordítani az adott fájlt, majd ezeket a fordítás után hozzáadni a leíróhoz. Ezt az eljárást teljes mértékben támogatja az általam adott megoldás.

Az előbbiek alapján belátható, hogy a fordítási folyamatok fő feladata ezeknek a függőségeknek a létrehozása, majd később ezek alapján a megfelelő szinkronizáció a háttértárral. A megoldásom eredményeként egy olyan inkrementális rendszer készült el, mely könnyen használható függőségek leírására, miközben az elvárt eredményt és teljesítményt nyújtja a mindennapi használat során. (Lásd: 9 Teljesítménymérés)

5.1.1 Szerializáció

A rendszer megköveteli, hogy az adatleírók implementálják a Java *Serializable* interfészt, így sorosítva őket a háttértárra, azonban a működés lazán csatoltságából kifolyólag felmerülő probléma, hogy beolvasás során a betöltött objektumokhoz tartozó osztályok esetlegesen még nincsenek betöltve a Java virtuális gépbe.

Ennek megoldására a rendszer elvárja a *Repsository* implementációktól, hogy ne hozzanak létre saját *ClassLoader* példányokat, hanem azt a futtatókörnyezeten keresztül tegyék meg. Amennyiben ezt a szabályt betartják, a környezet képes lesz arra, hogy az osztályok mellé elmentsen egy *ClassLoader* leíró objektumot, ami alapján újbóli betöltés alkalmával azok automatikusan példányosíthatók.

5.2 Adatkezelés

A működés során gyakran előfordul olyan eset, hogy egy adatot más formában szeretnénk elemezni, mely magával vonja egy olyan konverziós rendszer iránti igényt, ami az alapvető típusok közötti átalakítást elvégzi. Ilyen esetek előfordulhatnak például szkriptben definiált adatok Java oldalon elemezhető típussá konvertálása esetén, vagy azonos modulok különböző verziói által létrehozott objektumok elemzésénél.

A rendszer a konvertálások során a Java *API* által nyújtott reflexiós lehetőségeket használja ki. Alapvető átalakításokat végez primitív, tömb és *String* típusok esetén, illetve kezeli a kollekciókat, mely során olyan helyettesítő osztályokat példányosít, melyek a függvényeik hívása során végzi el az elemek konvertálását. (Ehhez szükséges a kollekciók sablon paramétereinek futási idejű tárolása is.) A Java 8-ban megjelent *Supplier* interfész példányait automatikusan kicsomagolja, és amennyiben *Supplier* példányra van szüksége a felhasználónak, azokat is létrehozza, így lehetőség van egyes számítások késleltetésére (*lazy* kiértékelésére) is. A rendszer konverziós mechanizmusa bővíthető az osztályok által, amennyiben statikusan implementálnak egy megfelelő típusú paraméterrel rendelkező *valueOf* metódust.

5.2.1 Szkript adatok

A fordítószkriptben leírt értékek Java oldalon elérhetővé tétele nem egyértelmű, mivel a szkriptben leírt összetett típusoknak nem szükséges, hogy legyen Java oldalú megfelelője. Ennek érdekében a rendszer egy olyan megoldást ad, mellyel az összetett típusokat, a kért típusú reprezentációvá alakítja át. Ehhez a felhasználónak a típusait Java *interface*-ként kell definiálnia, így megteremtheti annak a lehetőségét, hogy futási időben *Proxy* osztályok által példányosíthatóak legyenek, illetve a különböző verziók között is átjárást biztosít. Tekintsük a következő példát:

```
// Szkriptben definiált objektum
$Map = { Items: [ apple, pear ], Name: Alice };
```

```
public interface MyType {
     public String getName();

     public void setName(String Name);

     public default Collection<String> getItems() {
          return new ArrayList<>();
     }

     public void setItems(Collection<String> Items);
}
```

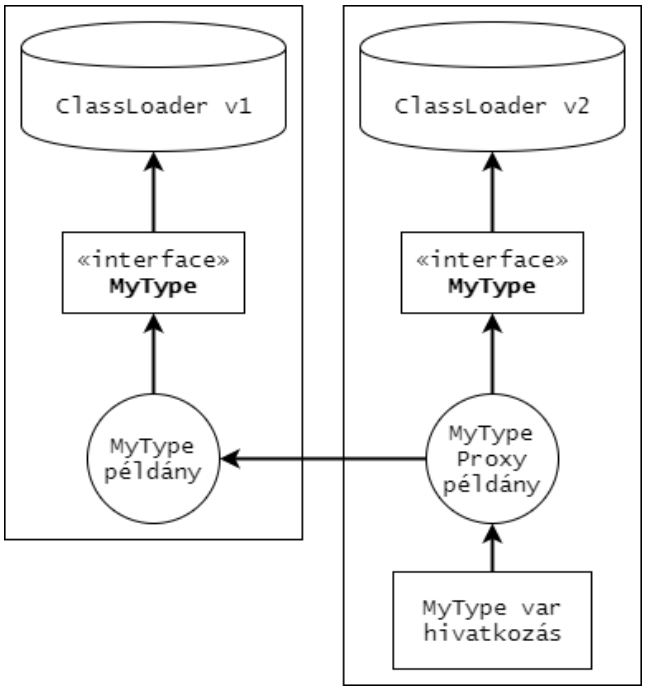
16. példa: Szkript adat konvertálás

A példa szkriptben definiált objektumot Java oldalon *MyType* típusú interfészként szeretnénk elérni, mely esetben a rendszer automatikusan becsomagolja a leírt *Map* objektumot egy *Proxy* osztályba, melynek a metódushívásait úgy továbbítja, hogy azok a szótár elemeit kérdezzék le.

A *MyType.getName()* metódus *"Alice"* értékkel fog visszatérni, míg a *MyType.getItems()* az *"apple"* és *"pear"* elemeket fogja tartalmazni. A rendszer ezt úgy végzi el, hogy a *Proxy* példányon hívott metódusok nevét, paramétereit, és visszatérési értékét elemzi, majd megfelelő formátum esetén elvégzi a műveletet a befoglalt *Map* példányon. Lehetőség van *default* implementációk megadására, mely során könnyebben kezelhetőek az adatok, amennyiben a felhasználó nem adta meg őket szkript oldalon. (pl.: nincs szükség redundáns *null* ellenőrzésekre)

5.2.2 Verzió kezelés

Amennyiben egy osztályból több verzió van betöltve és az egyik *ClassLoader*-hez tartozó osztály példányát akarunk egy másik verzióban található változónak értékül adni, *ClassCastException*-t fog dobni a Java környezet, mivel valójában két különböző osztály között akarunk kapcsolatot teremteni. Erre a problémára a rendszer megoldást kínál, mely során az előzőekhez hasonlóan *Proxy* osztályt hoz létre a kívánt *ClassLoader*-ben, és a hívásokat a másik *ClassLoader* általi osztály példányának továbbítja.



5. ábra: *ClassLoader* verzió konvertálás

Fontos megjegyezni, hogy a hívások továbbítása során nem egyenesen kerül meghívásra az eredeti példányon a metódus, hanem annak paraméterei és visszatérési értéke is konvertálásra kerül a megfelelő *ClassLoader* osztályaira.

5.2.3 Konklúzió

Az adatok kezelésének módszere alapján levonható a következtetés, hogy a felhasználó által implementált *Module* és *Operation* implementációknak a konfigurációs adatokat nem Java osztályokként érdemes kezelniük, hanem interfészek által definiált tulajdonságokként. Az interfészek implementációját a futtatókörnyezet automatikusan biztosítja, ezzel magas fokú flexibilitást és robosztusságot adva a rendszernek.

5.3 Bővítés

A megoldásom fő tulajdonsága, hogy támogatja különböző, felhasználó által létrehozott kódbázisok futási idejű betöltését. Ehhez a kapcsolatot a programozó egy *Repository* implementációval teremtheti meg, melyet a fordító rendszerbe *JAR* fájlok segítségével tölthet be.

A *Repository*-k feladatából adódóan, a következő metódusokat köteles a programozó megvalósítani:

Module createModule(ModuleIdentifier moduleid)

Létrehozza a paraméterként átadott azonosítóval rendelkező modult. Az azonosító tartalmazza az esetleges felhasználó által adott minősítőket.

ModuleOperation createModuleOperation(Module module, String operation)

Példányosítja a paraméterként átadott modulhoz tartozó, adott névvel ellátott operációt, amennyiben létezik.

ClassLoader getClassLoader(String identifier)

A függőségek megfelelő beolvasásához, a paraméterként kapott tetszőleges azonosító alapján visszaadja a hozzá tartozó *ClassLoader*-t.

A tervezés során törekedtem a lehető legkevesebb feladat megkövetelésére a *Repository* implementációktól, ezzel megadva a lehetőséget arra, hogy különféle megoldások is megvalósíthatóak legyenek. Ezekre például szolgálhatnak a következők, a teljesség igénye nélkül:

- Lokális gépen található modul betöltő implementáció,
- Privát hálózaton keresztüli modulok betöltését támogató megvalósítás,
- Online adatbázisból, közösségi modulok és operációk betöltésének támogatása.

A rendszer beépített modul használatával lehetőséget ad a *Repository*-k *JAR* fájlba való exportálására, mely során a megfelelő metaadatokkal ellátja a készült állományt. Mivel a *Repository*-knak azonosítóval kell rendelkezniük, azt a kódban egyszerű módon az osztály *@RepositoryURI* annotációval való ellátásával lehet megadni, mely a fordítás során annotáció feldolgozás segítségével beolvasásra kerül.

5.3.1 Modul és operáció implementáció

A rendszerben a modulok és operációk felelnek a feladatok elvégzéséért, melyek implementációját a programozónak kell biztosítania. A rendszer a megvalósítandó osztályokon kívül nem határoz meg megkötést arra nézve, hogy milyen feladatokat, és milyen módon kell elvégezniük. A rendszer azt sem határozza meg, és módszert sem ad arra, hogy hogyan lehet az elkészített modulokat exportálni az őket befoglaló *Repository*-k számára. Ebből következik, hogy az exportálás mechanizmusát teljes mértékben a *Repository* implementációknak kell biztosítania.

Ahhoz, hogy a modulok és operációk megfelelően paraméterezhetőek legyenek, a futtatókörnyezet automatikusan értékül adja a rendszerben található változókat a példányosított entitások megfelelően annotált mezőinek. Miután a modul, vagy operáció életciklusa véget ért, azok kimeneti változóit is analizálja a környezet, és változóként elérhetővé teszi őket további futás esetén. Ezt a következő példa szemlélteti:

```
// Hívó szkript
$Variable = test;
mymodule.example(Input = param): {
     ...
}
```

```
public class MyModule extends Module {
     @ModularData
     private String Variable;
     @ModularParameter
     private String Input;

     @ModularOutput
     private String Result;

     @Override public void open() { ... }
     @Override public void close() { ... }
}
```

17. példa: Modul implementáció

A feladatot végző példányokban (modul és operációban egyaránt) három annotáció segítségével határozhatunk meg automatikus ki- és bemeneti változókat:

- *ModularData*
  - Ezeknek a változóknak csak olyan adatok lesznek értékül adva, melyek nem a paraméterlistán, azonban a globális változók között szerepelnek.
  - Akkor érdemes ezt használni, amennyiben a feladat szemantikailag megköveteli, hogy a felhasználó a feladat lefutása után is felhasználja a változót.
- *ModularParameter*
  - Ezek a változók elsődlegesen a paraméterlistán átadott értékeket veszik fel, illetve ha a felhasználó nem adta meg őket, a globális változók között keres név szerint.
- *ModularOutput*
  - Kimeneti változó, melyet a rendszer az életciklus végén kezel az annotáció paraméterezésének megfelelően.

> Észrevehető, hogy a rendszerben definiált változók nem követik a Java nyelvben megszokott *camelCase* formázási irányvonalat. Ezt az indokolja, hogy amennyiben a futtatókörnyezetben található változókat *PascalCase* alapján nevezzük el, abban az esetben könnyen látható módon elkülöníthetők a Java oldalon definiált, és szkriptben is létező változók.

Mivel az operációknak csak a rájuk bízott feladatot kell elvégezniük, csak egyetlen absztrakt metódust kell megvalósítaniuk:

```
public class MyOperation extends ModuleOperation {
     // Input és Output változók Module-hoz hasonlóan

     @Override public void run() { ... }
}
```

18. példa: Operáció megvalósítás

Az operációk implementációjában a *Module*-hoz hasonlóan lehet a változókat deklarálni, a futtatókörnyezet ugyanúgy kezeli őket. A változók mezőknek való értékül adása során a rendszer az adatkezelés fejezetben bemutatott konverziókat alkalmazza. (Lásd: 5.2 Adatkezelés)

5.3.2 Együttműködés külső folyamatokkal

Egy fordítói rendszer esetében fontos, hogy lehetőséget adjon más eszközök meghívására, így már elkészült megoldásokat nem kell újra implementálni. Erre a feladatra a megoldásom nem nyújt közvetlen megoldást, azonban van módszer ilyen feladatok elvégzésére is. Amennyiben egyszerű külső folyamat meghívását szeretné elvégezni a fejlesztő, arra lehetőséget ad egy olyan modul, mely parancssori paraméterek alapján végrehajtja a folyamatot. Az inkrementális támogatáshoz a programozónak specifikálnia kell a be- és kimeneti fájlokat, így csak szükség esetén lesz végrehajtva a parancs. Ezzel a módszerrel megállapítható, hogy parancssori paraméterek segítségével bármely olyan eszköz integrálható a rendszerembe, mely közvetlen alfolyamatként példányosítható (pl.: *Gradle, Maven,* stb…).

Mivel a rendszerem a Java virtuális gépet használja, így lehetőség van olyan modulok készítésére, melyek más JVM alapú folyamatok betöltését elvégzik, és az általuk publikált *API* alapján meghívják azt. Ez a módszer körültekintőbb implementációt igényel, azonban így könnyebben konfigurálható szkript oldalról egy-egy külső eszköz. (Így működnek az általam implementált Android *dex* és *apksigner* feladatokért felelős operációk.)

5.4 Adatvédelem

Habár a rendszert használó programozó saját felelősségére futtatja azt, szükség lehet olyan funkcióra, mely biztosítja azt, hogy a fordítás során nem kerülnek olvasásra, írásra, és futtatásra olyan állományok, melyet a fejlesztő nem engedélyezett. Ennek érdekében a megoldásom olyan *SecurityManager* megvalósítást alkalmaz a futás során, mely az elérni kívánt állományokat a hívó kód forrása alapján szabályozza. Ehhez egyrészt az osztályokat beöltő *ClassLoader*-eknek kell megfelelő engedélyeket megadni, másrészt univerzális engedélyeket kell definiálni, melyek a futtató környezet tulajdonságai alapján dinamikusan vizsgálják az engedélyeket. Ebben a rendszerben háromféle kódot különböztetünk meg:

1. Rendszer osztályok, melyek a fordítói rendszerhez tartoznak,

2. *Repository* osztályok, melyek a betöltött modul és operáció feloldáshoz kötődnek,
3. *Module* és *Operation* implementációk, melyek a feladatokat végzik.

A számukra meghatározott engedélyeket a következő táblázat szemlélteti:

	Rendszer osztályok	*Repository* osztályok	*Module* és *Operation* osztályok
java.io.temp	*read, write, delete*		
Working directory	*read*	-	*read*
Storage directory	*read, write, delete*	Alkönyvtár*: read, write, delete*	
Build directory	*read, write, delete*	-	Alkönyvtár *read, write, delete*
modular.jar *JAR* fájl	*read - Whitelist*		
java.home *java.ext.dirs*	*read - Whitelist*		

1. táblázat: Hozzáférés szabályozás

A *java.io.temp, java.home, java.ext.dirs System property-k* által meghatározott könyvtárakhoz való hozzáférésnek az engedélyezése szükséges bizonyos Java *API*-k zavartalan működéséhez.

A *Working directory* az a könyvtár, mely indítási paraméterként van átadva a futtatandó feladatnak, és a relatív útvonalak ezzel szemben lesznek feloldva. A rendszer osztályoknak azért van *read* hozzáférésük, mivel a folyamat vezényléséhez a fordítói szkriptekhez muszáj kérdés nélkül hozzáférniük. A *Repository* osztályok nem kapnak ehhez hozzáférést, mivel azoknak a futási könyvtár ismerete nélkül kell végezniük a feladataikat. Ebbe a könyvtárba senki nem kap írási és törlési engedélyt, mivel szándékosan senki nem írhatja felül a programozó által használt, projekthez tartozó fájlokat.

A *Storage directory* olyan könyvtár, mely egy inicializáláskor meghatározott elkülönített helyen található, és a jelenleg futó fordítási folyamattól független. Erre akkor van szükség, ha egy-egy *Repository, Module* vagy *Operation* fordítási folyamatokhoz és projektekhez nem kapcsolódó adatot szeretne eltárolni. Minden *Repository*-nak külön alkönyvtára van a szeparáció megvalósításáért.

A *Build directory* könyvtár az, melybe a fordítási folyamat során a modulok és operációk a fájljaikat elhelyezhetik. A *Build directory*-n belül minden modul és operáció a hozzájuk tartozó *Repository* azonosítójától függően elkülönítve tárolhatja a fájljait. Tipikus *Repository* implementáció esetén további szeparáció van megvalósítva modulok között is.

A fordítási rendszer osztályait tartalmazó *JAR* archívumhoz mindenkinek olvasási hozzáférése van, mivel meghatározó felhasználási eset a Java fordítás során ennek a *classpath*-hoz való hozzáadása.

A Java *API* további megfelelő működéséért a *java.home* és *java.ext.dirs* értékek által meghatározott könyvtárak a fehérlistával kerültek engedélyezésre.

Fontos megemlíteni, hogy semmilyen osztály nem rendelkezik *execute* (futtatási) joggal, mely azt jelenti, hogy minden olyan művelethez, mely más folyamatot szeretne indítani a felhasználó gépén, először engedélyt kell kérni.

Ezt a biztonsági rendszert lehetőség van kikapcsolni, ami esetén minden kérés engedélyezésre kerül. (Ezt akkor érdemes megtenni, ha például automatizált fordítást és teszteket szeretnénk végrehajtani a programozó felügyelete nélkül, és tőle függetlenül.)

6 Inkrementális Java fordítás

A Java nyelven fejlesztett szoftverek esetén is fontos, hogy a fordítási idő minél rövidebb legyen kis módosítások esetén. A JDK-hoz tartozó *javac* fordító nincs arra felkészítve, hogy ezt támogassa, így azt csak a hozzá tartozó *API*-n keresztül lehet inkrementálissá tenni.

A *javac* fordító egy lépésben végzi el a fordítást, és a fájlok közötti függőségek egyértelmű meghatározására nincs lehetőség további analízis nélkül. Ahhoz, hogy megtehessük, hogy a fordítás során csak a megváltozott forrásokat fordítjuk le, szükség van tudni, hogy mely osztályok milyen adatokat használnak más forrásfájlhoz tartozó osztályokból. Ezt az analízist tovább nehezíti, hogy a *javac* a fordítás során a konstans értékekre nem tartja meg a hivatkozást, hanem azokat behelyettesíti.

Meglévő megoldások ezt úgy közelítik meg, hogy elemzik a fordítás során létrejött osztályok *bytecode*-ját, és az alapján hozzák létre a függőségeket (pl.: *Gradle* [7]). Más megoldások saját fordítói mechanizmust implementálnak, így fordítás közben hozzák létre a függőségeket (pl.: Eclipse[6]). Míg az utóbbi jó megoldást nyújt, addig sokkal nehezebb implementációt kíván, a *bytecode* alapú megoldás pedig a konstans behelyettesítés miatt nem mindig tudja meghatározni a pontos függőségeket.

A két fenti megoldás helyett lehetőség van egy harmadikra, mely azon alapul, hogy a *javac* lehetőséget nyújt a fordítás során a forrásfájlok szintaxis alapú elemzésére, mely alapján meghatározhatóak a függőségek. Ennek előnye, hogy nem szükséges saját Java fordítót implementálni, illetve az elkészült *bytecode* elemzésére sincs szükség, miközben a konstans behelyettesítési hibák nem jelentkeznek.

A megoldásomban a Java forrásfájlok inkrementális fordítása a következő lépésekre bontható:

1. Megváltozott fájlok meghatározása,
    - Amennyiben a fejlesztő módosította a forrásokat, vagy más módon törlődtek az előző fordítás kimeneti fájljai, a megfelelő forrásokat újra kell fordítani.
2. A Java osztályokban bekövetkezett változások felderítése (*ABI* változás),
    - Amennyiben olyan változás történt egy osztályban, amire másik osztály hivatkozik, abban az esetben a hivatkozó forrásokat is újra kell fordítani, különben a fejlesztő nem értesülne a hibáról. Ilyen esemény pl.: metódus törlése, konstans módosítása, osztály törlése.
3. Az összes érintett fájl fordítása.

[6] Eclipse JDT Core Component - https://www.eclipse.org/jdt/core/

A megoldás egyik pozitívuma, hogy csak publikus *javac API*-t használ, és a forrásanalízis miatt biztosan korrekt eredményt ad. Hátrányként elmondható, hogy a változásdetektálás módszere miatt, az első fázisban beolvasott, és absztrakt szintaxis fává alakított állományokat újra be kell olvasni a valódi fordítás alkalmával is.

6.1 *ABI* változás felderítése

A *javac* fordító 3 lépésre tagolja a fordítást, melyek a következők: *parse, analyze, generate* [13].

Ezeket a fázisokat manuálisan végiglépve lehetőség van csak a beolvasás meghívására, ami szükséges a megfelelő működéshez, mivel a változások felderítése fázisban nem szabad elvégezni a fájlok fordítását. Ha elvégeznénk, az elkészült *bytecode* hibás lehet, tekintsük a következő példát:

```
public class A {
     public static final int CONSTA = C.CONSTC;
}
```

```
public class B {
     public static final int CONSTB = 13;
}
```

```
public class C {
     public static final int CONSTC = B.CONSTB;
}
```

19. példa: Konstans behelyettesítés *ABI* változás

A példában 3 osztályunk van, 3 különböző fájlban, a megfelelő konstans hivatkozásokkal. Ebben az esetben az első teljes fordítás után mindhárom elkészült *class* fájl a 13-as értéket fogja tartalmazni.

Módosítsuk az A és B osztályokhoz tartozó fájlokat, írjuk át *CONSTB* értékét 10-re! Ebben az esetben a változások felderítésénél az A és B fájlok kerülnek beolvasásra, melyben észrevesszük, hogy a B konstans értéke megváltozott. Ha ilyenkor a C lefordítása előtt/nélkül generálnánk le A és B osztályhoz tartozó kódot, az A osztály *class* fájljában még mindig 13-as értékkel szerepelne a konstans, hiszen annak az értékét a C osztály *class* fájljából olvasná ki, ahol még mindig 13 maradt. Ebből adódóan, a valódi fordítást az összes megváltozott, és összes hivatkozó fájllal együtt kell elvégezni, különben hibás kimenetet kaphatunk.

Az *ABI* változások meghatározása után, ciklikusan kell keresni azokat a forrásfájlokat, melyekre egy-egy változás kihat, és ezek alapján újrafordítani azokat.

> A ciklikus keresést indokolja a fenti példa abban az esetben, ha csak a B fájl konstansa kerül módosításra. Ebben az esetben a C-re közvetlenül kihat, viszont az A-ra csak C-n

> keresztül másodlagosan. Ha csak egy lépésen keresztül vizsgálnánk a függőségeket, hibás eredményt kapnánk.

6.2 Fordítás

Utolsó lépéseként a megváltozott fájlok a *javac API*-n keresztül lefordításra kerülnek, majd azok deklarált metódusai, változói, és egyéb osztály leíró adatai lementésre kerülnek, hogy következő alkalommal összehasonlíthatóak legyenek a változásokkal.

Az osztályok adatainak felderítése után amennyiben a felhasználó kéri, a natív C/C++ *header* forrásfájlok is legenerálásra kerülnek, így két feladatot lehet egyszerre elvégezni. Érdemes megemlíteni, hogy a megoldásom a JDK-val mellékelt *javah* programnál bizonyos szempontból jobb *header* fájlokat generál. (Lásd: 12 Függelék)

7 Daemon

Egy modern fordítói rendszernek érdemes szolgáltatást nyújtania egy olyan háttérben futó folyamatra, mely felé továbbítani lehet a fordítási kéréseket. Ennek eredményeként sokszor csökkenteni lehet a fordításra szánt időt az inicializálási idővel, mely a projekt méretétől függően összemérhető lehet a teljes ráfordítási idővel. A háttérben futó szolgáltatás (*daemon*) ezek mellett a projektek struktúrájáról gyorsítótárazhat egyéb adatokat, illetve más funkciókat is adhat a programozónak.

A megoldásomban egy ilyen rendszert is implementáltam, mely a gyorsítótárazás mellett lehetőséget ad a fordítás más gépeken való végrehajtására is. A szolgáltatásnak a következő feladatokat kell teljesítenie:

1. Hálózati és lokális gépen keresztüli hozzáférés biztosítása,
2. Fordítási feladatok fogadása és futtatása,
3. Fájlrendszerekhez való hozzáférés kezelése,
4. Projekt specifikus adatok gyorsítótárazása.

A *daemon*-hoz való csatlakozás terminálon keresztül történhet, mely segítségével alapvető navigációt lehet végezni a fájlrendszerben, illetve konfigurálni a fordítási környezet paramétereit. A kapcsolat lokális gépen, illetve hálózaton keresztül épülhet fel, így lehetőség van egy számítógép segítségével máshol fordítási folyamatok indítására. A programozónak lehetősége van a *daemon*-t futtató számítógép, illetve a saját rendszerén található fájlok használatára is. A *daemon* eltárolja egyes az fordításokhoz tartozó gyorsítótárazható adatokat, melyeket automatikusan lebont a hozzájuk tartozó terminál kapcsolat megszűnése esetén.

7.1 Hálózati hozzáférés

A hálózaton keresztüli elérés fontos része a *daemon* szolgáltatásainak, mivel ez biztosítja a távol fordítás lehetőségét.

> Példának felhozható egy olyan projekt fordítása, mely során szeretnénk azt elérhetővé tenni macOS, Windows, és Linux operációs rendszerekre. A különböző operációs rendszerekhez tartozó fejlesztői eszközök gyakran nem elérhetőek más platformokon, így szükség van több számítógépen való fordítás elvégzésére. Erre tökéletes megoldást nyújt a hálózati *daemon*-on keresztüli vezénylés.

A terminál és célszámítógép bináris *TCP* csatornán kommunikál, mely a megbízhatósága miatt teljes mértékben megfelel a követelményeknek. (Lokális géppel lévő kapcsolat esetén is ezt használja a rendszer, mivel jó kommunikációs csatornát nyújt kép folyamat (*process*) között.)

Fontos része a *daemon*-nak, hogy a fordítási folyamat minden esetben a *daemon*-t futtató gépen fog lefutni. Ilyenkor a felhasználó gépén található fájlokat elérhetővé kell tenni a távoli gépnek, mely új funkciók bevezetését igénylik a rendszerbe.

Az előbbi problémából adódóan a megoldásomban a közvetlen fájlrendszer elérését egy absztrakciós réteg mögé rejtettem, mely megengedi azt, hogy a megfelelő implementáció esetén az érintett fájlok akár távoli gépen legyenek. Emellett arra is lehetőség nyílik, hogy futtatás során tetszőleges könyvtárakat csatoljunk fel meghajtóként a rendszerbe, így sokkal testre szabhatóbb a fájlokhoz való hozzáférés, melyet a következő példa szemléltet:

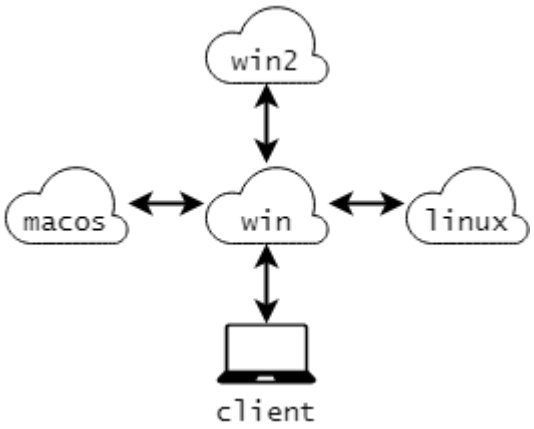
```
Lokális fájlrendszer gyökér meghajtói:
C:\
D:\

Projekt könyvtár:
C:\Users\Sipka\Projects\Example

Felcsatolt meghajtók:
D:\       ->   D:\
WD:\      ->   C:\Users\User\Projects\Example
```

20. példa: Meghajtók

A példában látható, hogy a futtatás során csak a D: és WD: nevű meghajtók lesznek láthatók kódból, így például a rendszer *API*-ján keresztül nem fér hozzá a felhasználó (nem is látja) a C: meghajtót. A meghajtók nevei több betűből is állhatnak, illetve az UNIX rendszereken használt ’/’ nevet is felvehetik. Ez a megoldás nem gátolja meg a modulokat a fájlok más Java *API*-val történő elérésében, azonban annak szabályozására be lehet állítani *SecurityManager*-t (Lásd: 5.4 Adatvédelem). A meghajtók más számítógépen lévő könyvtárra is mutathatnak.

Távoli gépen való futtatás esetén előfordulhat, hogy egy fájlt többször szeretnénk megnyitni, mely fölöslegesen duplikált adatforgalmat jelenthet. Ennek érdekében a távoli gépen lévő fájlokat a rendszer gyorsítótárban helyezi el, és automatikusan frissíti azokat, ha megváltoznak.

Fontos tulajdonsága a hálózaton keresztüli fordításnak, hogy a folyamat során nem csak egy számítógép fájljaira hivatkozhatunk, hanem több eszköznek az állományait használhatjuk. Ez alapján a *daemon* csillag gráf szerkezetben építi fel a kapcsolatait, és a futtatás során dinamikusan kéri el a fájlokat. (Csak azok a fájlok kerülnek átküldésre, melyeket a felhasználó valóban be kíván olvasni, tehát a hozzájuk tartozó *InputStream* megnyitásakor automatikusan.)



6. ábra: *Daemon* hálózat

A 6. ábra alapján felépített elrendezésben a *client* azonosítóval rendelkező gépről csatlakozunk a *win* gépen futó *daemon*-hoz, mely a *macos, win2, linux* gépekkel kapcsolatban áll. A terminál segítségével lehetőségünk van az összes elérhető fájlrendszerben való navigálásra, és fordítási folyamat indítása esetén is hivatkozhatunk azokon a gépeken található állományokra. Ez jó felhasználási módja lehet olyan fejlesztési eseteknek, ahol több embernek kell együtt dolgoznia, azonban úgy szeretnék tesztelni a programot, hogy ne kelljen közben fájlok másolásával foglalkozni. (Erre a problémára alapvető forráskezelő rendszerek megoldást nyújtanak, azonban így lehetőség van közös tesztelésre, és nincs szükség esetleg hibás források feltöltésére.)

> Érdemes megjegyezni, hogy a *daemon* példányoknak alapvető útvonalkeresési funkciót kell megvalósítaniuk, mivel amennyiben a *client* gépről próbáljuk a *linux* gépen található fájlokat kezelni, a *win*-en található *daemon*-nak továbbítania kell a forgalmat a megfelelő irányba. Ez könnyen megvalósítható, és mivel csak konfigurálás alatt történik ilyen, nem ró jelentős terhet a rendszerre.
>
> Észre lehet venni, hogy bár útvonalkeresést végeznek a *daemon*-ok, ezt csak egy lépés (*hop*) erejéig kell megtenniük, mivel nem hozható létre olyan fordítási konfiguráció, ahol egy fájlt fordítási időben kettő vagy több lépésen keresztül kell elérni. Amennyiben ilyen történik, a távolabbi gépet közvetlenül kell csatlakoztatni a fordítást végző *daemon*-hoz.

7.2 Gyorsítótárazás

A gyorsítótárazás feladata, hogy csökkentse a rendszer indítására fordított időt, így gyorsabban el lehet végezni egy-egy fordítást. A *daemon* kétféle adatot tart életben futása során:

1. Kliensek kapcsolódásától független adatok
    - Ezek olyan objektumok, melyek egy-egy fordítási folyamattól függetlenek, így azokat a kliensek szétkapcsolása után is életben lehet tartani.

- Ebbe a kategóriába sorolhatóak a betöltött *Repository*-k, modulok, és operációk.

2. Kliens specifikus adatok
    - Ezek olyan objektumokat határoznak meg, melyek projekt specifikus értékeket tartalmaznak. Kliens szétkapcsolása után ezeket fel lehet szabadítani.
    - Ide sorolhatóak az egy-egy projekthez tartozó függőségi gráfok, melyek beolvasása és kiírása a projekt növekedésével egyre több ideig tarthatnak.

A projekt specifikus függőségi gráfok egy-egy fordítás végeztével aszinkron módon íródnak ki a háttértárra, ezzel is közelebb hozva a fordítás eredményéhez való hozzáférés legkorábbi időpontját.

A gyorsítótárazás alá sorolható a távoli gépeken tárolt fájlok adatainak lokális tárolása is, azonban ezek a lehetséges nagy méretük miatt mindig a háttértáron lesznek elhelyezve. Ezek az eltárolt adatok a *daemon* futása után is megmaradnak.

8 Eclipse kiterjesztés

Az *Eclipse* fejlesztőkörnyezet a *The Eclipse Foundation*[7] által fejlesztett nyílt forráskódú szoftver, mely Java környezetben fut. Fő tulajdonsága, hogy kiterjeszthető, ez által sok programnyelvet támogat, illetve számos fejlesztői kiterjesztéssel (*pluginnal*) rendelkezik.

Az általam készített kiterjesztéssel egy alapvető integrációját valósítottam meg a rendszeremnek, mely a következő funkciókkal segíti a fejlesztőt:

- A fordítás során létrejövő információ alapján konfigurálja a projekteket,
  - Ennek segítségével az *IDE*-ben használható lesz a forrásfájlokban az automatikus kiegészítés (*Content Assist*).
  - Ezek között a konfigurációk között tetszőlegesen váltani lehet, mely a többcélú fejlesztés esetén hasznos lehet a programozó számára.
- Szkriptfájlokban található fordítási célok meghívása,
  - A kiterjesztés értelmezi a fordítószkripteket, és kiolvassa a meghívható célokat. Ezek alapján egy legördülő menüben elérhetővé teszi futtatás céljából.
- Fordítási hibák és figyelmeztetések esetén jelzők (*markerek*) elhelyezése a forrásfájlokban,
  - A futtatókörnyezet egy meghatározott kimeneti formátum alapján írja ki az ilyen jellegű hibákat a kimenetre, melyek automatikus értelmezésével lehet segíteni a programozót.
- Fordítási adatok és objektumok gyorsítótárazása.
  - *Daemon*-hoz hasonló módon, a fordítási idő csökkentése. (Lásd: 7 *Daemon*)

A kiterjesztéshez tartozó példák a Függelékben láthatóak.

[7] About the Eclipse Foundation - http://www.eclipse.org/org/

9 Teljesítménymérés

A rendszer eredményei az alapján verifikálhatóak, hogy a mindennapi használat során mennyire lehet a segítségével probléma és megakadás nélkül fejleszteni. Az eredményeket ebben a fejezetben teljesítmény mérések segítségével, és használati példákkal támasztom alá.

A mérések során olyan példákkal hasonlítom össze az általam készült megoldás fordítási idejét, melyek a mindennapi szoftverfejlesztés során előfordulhatnak. A példák olyan fordítási lépések levezénylésén alapulnak, melyek mind teljes fordítást, mind inkrementális változtatások utáni fordítás idejét méri.

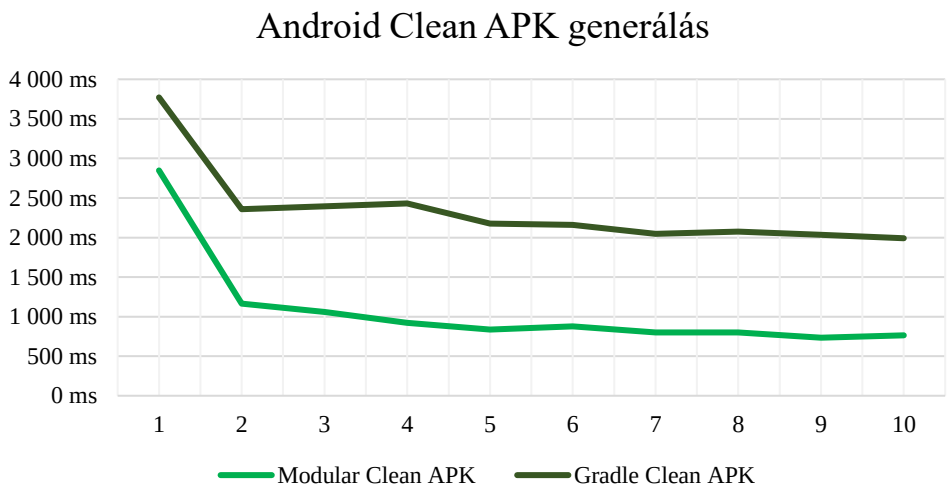
A megoldás saját felhasználás alapján, Java, C++, és Android alapú fordítási lépések sebességét hasonlítottam össze a hivatalosan adott fejlesztőeszközök fordítási idejével. A rendszeremben a fordítási feladatokért a saját magam által írt modulok és operációk felelősek.

A mérések során minden alkalommal a *daemon* háttérszolgáltatással lett futtatva a fordítás, a megoldásom az adatokon a *Modular* jelzővel vannak ellátva.

9.1 Android fordítás

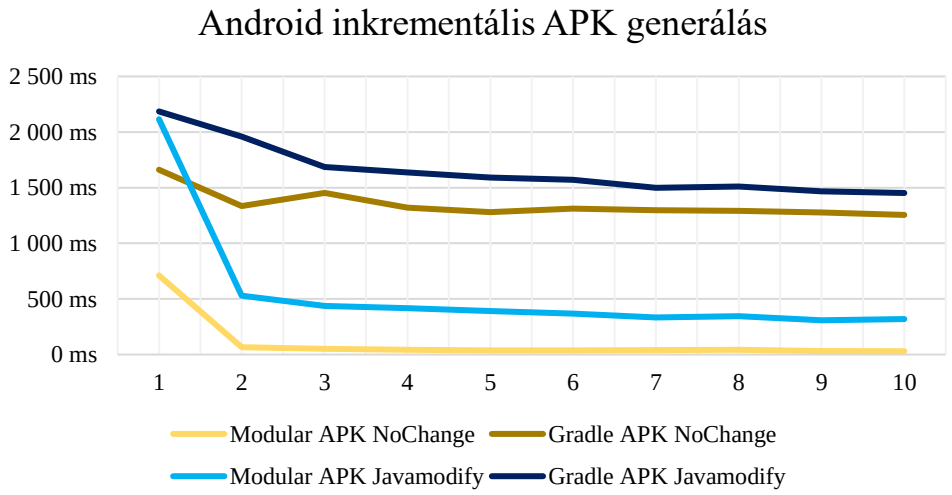
Ezen teszt olyan egyszerű Android *APK* (*Application Package File*) készítésének idejét méri, mely több Android fejlesztés során használt technológiát megkövetel. A folyamat során Java és *AIDL* (*Android Interface Definition Language*) forrásokat kell fordítani, az elkészül *class* fájlokat Android kompatibilis *dex* formátummá átalakítani, Android erőforrásokat kell konvertálni, *APK*-t csomagolni, majd digitálisan aláírni.

Az Android fordítás időeredménye a *Gradle* általi futás után kiírt időtartam. Az első tesztfordítás előtt a *Gradle Daemon* el lett indítva. A *Gradle* az *--offline* kapcsolóval lett futtatva, így meggátolva a fordítási idő hálózati kapcsolatokból adódó szükségtelen növekedését. Érdemes megemlíteni, hogy a *Gradle* a *Daemon* elindításakor projektspecifikus konfigurálást végez, ami az első fordítás gyorsabb eredményét hozhatja magával.



7. ábra: *APK* generálás előzmény nélkül

Az előzmények nélküli fordítás esetén a kezdeti inicializálás időhozamát leszámítva az én megoldásom (*Modular*) a végére kb. 40%-al gyorsabban végezte el a rá rótt feladatokat.



8. ábra: Android inkrementális fordítás

A *JavaModify* tesztek esetén egy Java forrásfájl került módosításra minden teszt futtatása előtt, míg a *NoChange* azonosítóval rendelkező teszteknél bementi fájlok módosítása nélkül kellett az eredményt generálni. A megoldásom (*Modular*) mindkét esetben gyorsabban készítette el a kimeneti APK fájlt. Mivel a Java fájlok fordításához tartozó tesztek kb 270 ms-al tovább tartottak a változás nélkülihez képest, ezért megállapítható, hogy a *Gradle* fordító nagyobb kezdeti költséggel rendelkezik, azonban hosszabb távon ez a projekt méretének növekedésével elhanyagolhatóvá válhat. A *NoChange* tesztek esetén az előző kimeneti *APK* tartalma egyik teszt esetén sem módosult.

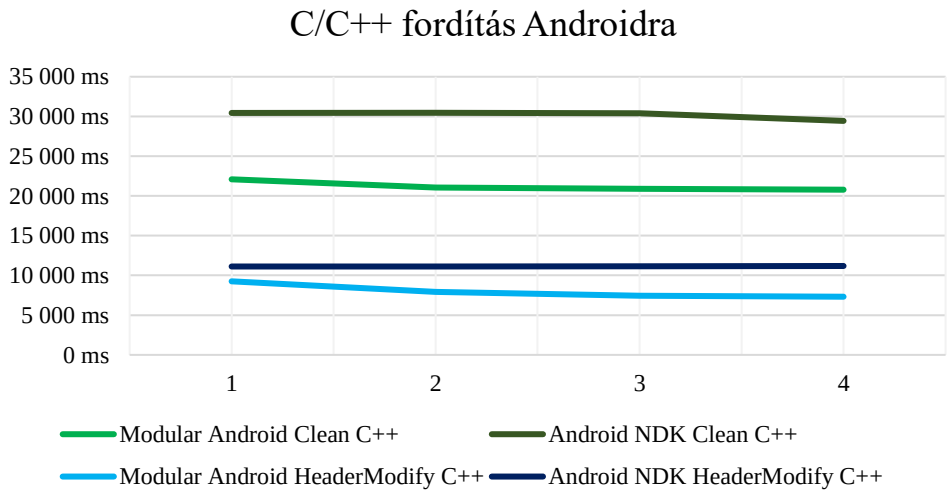
Ezt a tesztet először olyan Android modul implementációval is elvégeztem, ahol a *dex* formátummá való átalakítást nem a Java folyamaton belül, hanem új külső folyamat létrehozásával végeztem el. Ebben az esetben az én megoldásom alulmúlta a *Gradle* által nyújtottat, mivel ez a hívás több mint 1 másodperccel hosszabb ideig tartott. Ezután átalakítottam a modult, és a *dex* átalakítást Java folyamaton belül hívtam meg, mely esetben az eredmény jelentősen gyorsult. (Az *dex* átalakítást az Android Studio is folyamaton belül végzi el.)

9.2 C++ fordítás

A C++ nyelv támogatásával két operációs rendszerre végeztem el fordítási méréseket. A saját megoldásomat hasonlítottam össze az Android *NDK* (*Native Development Kit*) és a Visual Studio fordítási idejével. Ebben az esetben külső folyamatok végzik a fordítás nagy részét, és ez megmutatja, hogy az általam adott rendszer mennyire képes az inkrementális függőségek gyors kezelésére.

A tesztek során egy 628 forrásfájlból és kb. 210 000 sor kódból álló projekt fordítása a feladat. A teszteket 4 alkalommal végeztem el, mivel az eredmény már ennyi mérés alapján is állandósult.

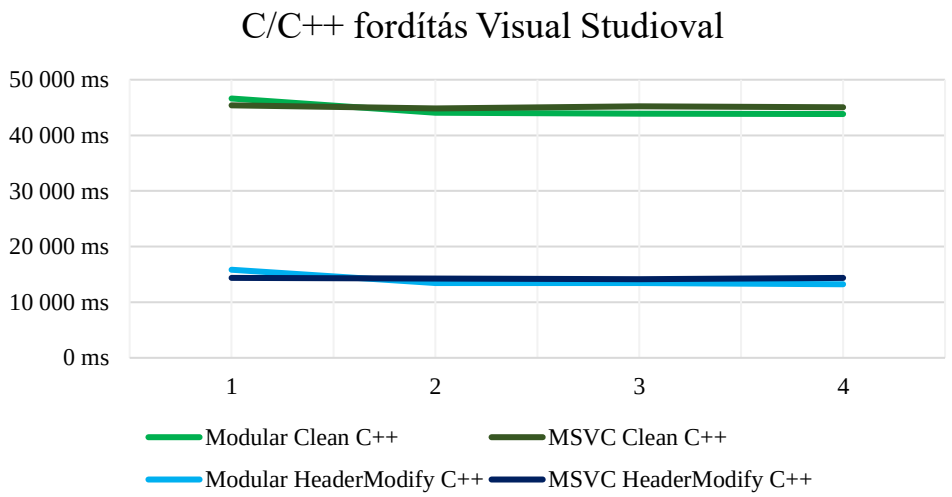
A *Clean* tesztek előzmény nélküli fordítási időt mutatnak, a *HeaderModify* azonosítóval rendelkezők pedig egyetlen C++ *header* fájl módosításából adódó további újrafordítások (18 forrásfájl) miatti feladatok futási idejét adja.



9. ábra: C/C++ fordítás Androidra

Az eredmények alapján az én megoldáson (*Modular*) az Android *NDK*-hoz képest mindkét esetben kb. 33%-al gyorsabb fordítási időt adott. Érdemes emellett megemlíteni, hogy amennyiben az *NDK*-val való fordítás során engedélyezzük a *LOCAL_SHORT_COMMANDS*

használatát, az jelentősen lelassítja annak futását. (Ennek a változónak a beállításával az *NDK* kezeli a túl hosszú parancssorok általi hibákat Windows operációs rendszer alatt.)



10. ábra: C/C++ projekt Visual Studio-val

Teszteset	Átlag
Modular Clean C++	44 608 ms
MSVC Clean C++	45 117 ms
Modular HeaderModify C++	13 988 ms
MSVC HeaderModify C++	14 282 ms

2. táblázat: C/C++ Visual Studio teszt átlagok

A Visual Studio-val összehasonlítva, mindkét tesztesetben az elemzett fordítórendszerek közel azonos eredményt hoztak, mely alapján megállapítható, hogy a rendszerem (*Modular*) legalább olyan gyorsan végzi el az ekvivalens feladatokat, mint a Microsoft fejlesztőkörnyezete. Fontos megjegyezni, hogy a rendszerem úgy produkálja az azonos fordítási időt, hogy közben több funkciót támogat, és kevésbé optimalizált eszköz.

A Visual Studio szorosan csatoltan hívja a fordítási lépéseket, mely azt eredményezi, hogy a fordítás elején létrehoz megfelelő számú (ebben az esetben 4) fordító folyamatot (*process*), majd azokkal kommunikálva végzi el a fordítást. Ezzel megspórolja a folyamat létrehozásával és inicializálásával járó költséget, mely 628 fájl esetén 624 folyamattal kevesebbet jelent a rendszeremhez képest.

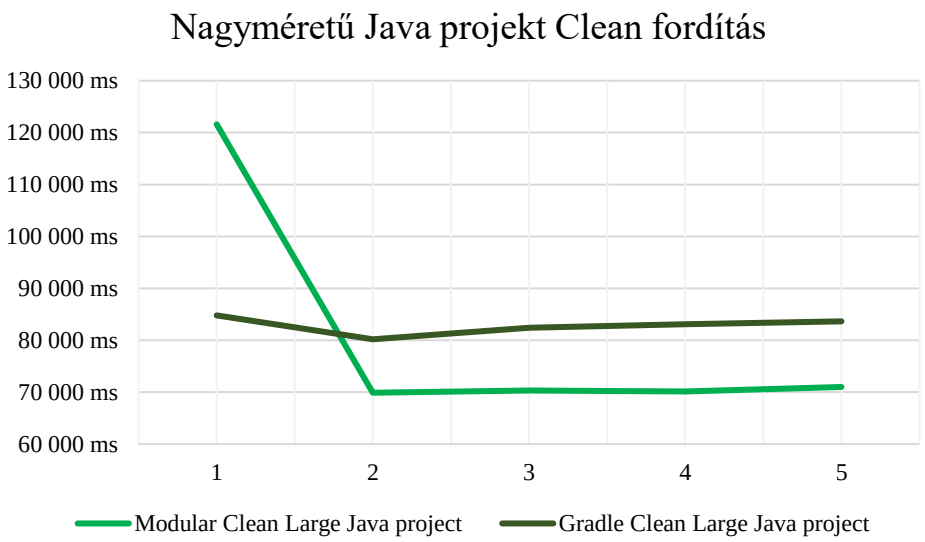
Érdemes megemlíteni, hogy amennyiben különböző forrásfájlokra különböző fordítási paramétereket határozunk meg, abban az esetben a Visual Studio egy folyamat használatával fogja elvégezni a fordítást, így a többprocesszoros rendszereket egyáltalán nem használja ki.

Az én megoldásom ugyanúgy támogatja az ilyen típusú konfigurálást. (Ilyen konfigurációra az Android *NDK* hagyományos használata esetén nincs lehetőség.)

Fontos leírni, ha amennyiben rendszer *header* fájlban hajtunk végre módosítást (pl.: *stdio.h*), abban az esetben a Visual Studio nem érzékeli a változást, és a megfelelő projektben lévő forrásfájlokat nem fordítja újra, mellyel hibás működést produkál. A Visual Studio-ban a teljesítményt tovább fokozza, hogy a *header* fájlokra mutató függőségeket szorosan csatolt módon, folyamatok közötti kommunikáció révén oldja meg, míg én a megoldásomban a */showincludes* kapcsoló használatával határozom ezeket meg. A Microsoft jelenleg nem publikált *API*-t a fordítójának közvetlen konfigurálására.

9.3 Java fordítás

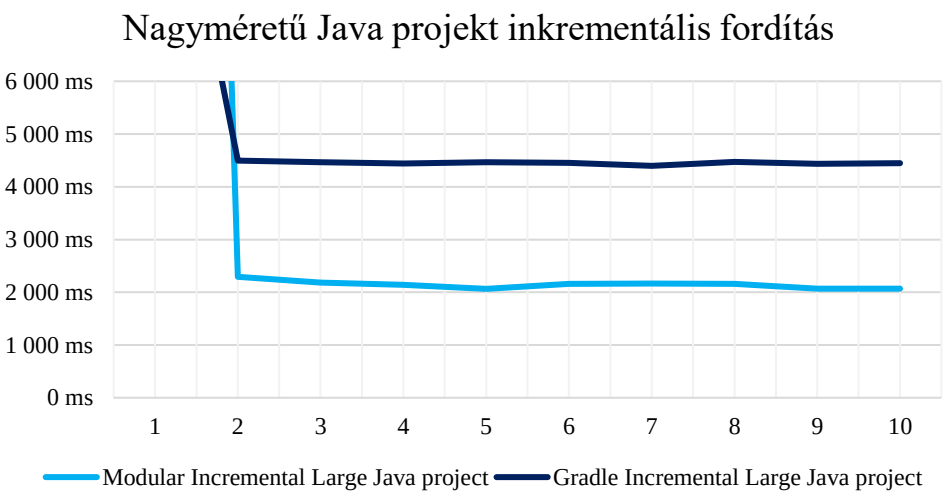
Egy fordítói rendszernek fontos tulajdonsága, hogy a nagyméretű projekteket hogyan kezeli. Ennek a vizsgálatára egy 50 000 fájlból álló Java projekt teljes és inkrementális fordítási idejét hasonlítottam össze a *Gradle* rendszerével. A mérések a *Gradle* inkrementális teszteléséhez [7] használt, publikusan elérhető projektjével zajlottak.[8]



11. ábra: Nagyméretű Java projekt Clean fordítási idők

A mérés alapján látható, hogy a kezdeti inicializálást leszámítva az én rendszerem (*Modular*) kb 10 másodperccel (kb. 12%) gyorsabban végezte el a fordítást. A kezdeti hosszabb inicializálási azzal magyarázható, hogy a rendszerem a *Daemon* indításakor nem végez el projekt specifikus konfigurálást, ellenben a *Gradle*-vel, mely a fordítás megkezdése előtt már betölti a megfelelő adatokat.

[8] Single, large project - https://github.com/gradle/performance-comparisons



12. ábra: Nagyméretű Java projekt inkrementális fordítási idők

Az inkrementális mérések során a fordítás előtt egy Java forrásfájl lett módosítva, majd a fordítók feladata a megfelelő függőségek megállapítása, és csak a változott forrásfájlok újrafordítása. Ebben a kezdeti inicializáció jelentősen meghaladja a hosszú távú fordítási időt, így megfelelő szemléltetés érdekében a grafikonon ez nem látható (*Modular*: 51 353 ms, *Gradle* 12 377 ms). Az inicializációs idők különbsége az előző méréssel megegyezően az előkészítési többlettel magyarázható.

A mérések eredményeként megállapítható, hogy a rendszerem mind teljes, mind inkrementális fordítás esetén jól használható nagyméretű projektek esetén is.

10 Összefoglalás

Dolgozatomban egy olyan fordítói rendszer felépítését mutattam be, mely a jelenleg elérhető eszközöktől eltérően közelíti meg a problémát, és a feladatok szekvenciális leírásán alapul. Egyik fő tulajdonsága a bővíthetőség támogatása, mellyel lehetőség van tetszőleges feladatok elvégzésére fordítási időben.

Az elkészült rendszerhez készült *Eclipse* kiterjesztéssel akadálytalanul lehet ellátni a napi fejlesztési feladatokat, és a fordítási lépések átláthatóan konfigurálhatóak a leírónyelv segítségével. A bemutatott mérési eredmények bizonyítékot adnak a rendszer működésére, mely a fordítási feladatokat a dedikált fordítóknál gyorsabban képes levezényelni. A mérések a skálázhatóság megfelelő támogatását is alátámasztják, így kimondható, hogy ipari környezetben is megállja a helyét mind funkcionalitás, mind teljesítmény tekintetében.

10.1 Továbbfejlesztési lehetőségek

A rendszer fő feladata a kiterjeszthetőség támogatása volt, így a teljesítménybeli fejlesztések kevesebb figyelmet kaptak az implementáció során. Ennek ellenére a mérések alapján megállapítható, hogy a rendszer jól megállja a helyét, azonban a későbbiekben további optimalizációkat érdemes megvalósítani (pl.: további többszálúsítás, egyéb gyorsítótárazás).

A fejlesztések támogatásának érdekében további *IDE* kiterjesztések készítése is fontos feladat, mellyel az *Eclipse* mellett más környezetek is zökkenőmentesen használhatóak lesznek. (pl.: *IntelliJ, NetBeans, Visual Studio*). A kiterjesztések megvalósíthatóságáról mindhárom fejlesztőkörnyezet esetén végeztem vizsgálatokat, melyek alapján azok is megfelelően támogatják a kiterjesztés lehetőségét (erre azok széles választéka is bizonyítékként szolgál).

A kifejlesztett rendszer a Java 8-as verziója által nyújtott szolgáltatásokat használja, és ezek meglétére alapoz. A Java 9 megjelenésével érdemes annak új megoldásait is bevezetni, illetve kompatibilissé tenni a bevezetett modul alrendszerrel [14].

Habár *Repository*-k fejlesztése nem tartozik az alaprendszerhez, érdemes egy olyan elkészítése, mely lehetőséget ad modulok interneten keresztüli betöltésére és publikálására. Ez által a programozónak kevesebb manuális konfigurációt kell végeznie, illetve *open-source* implementáció esetén jó közösségi támogatást nyújthat a fejlesztők számára.

11 Irodalomjegyzék

[1] G. Kumfert és T. Epperly, „Software in the DOE: The Hidden Overhead of"The Build"," Lawrence Livermore National Lab., CA (US), 2002.

[2] S. Bailliez, „Apache Ant User Manual," The Apache Software Foundation, 2003.

[3] S. McIntosh, B. Adams és A. E. Hassan, „The evolution of Java build systems," *Empirical Software Engineering,* pp. 578-608., 2012.

[4] T. O'Brien, „Maven: By example. An introduction to Apache Maven," 2010.

[5] H. Dockter és A. Murdoch, „Gradle User Guide," 2015.

[6] G. Team, „The groovy programming language," 2014.

[7] „Incremental Compilation, the Java Library Plugin, and other performance features in Gradle 3.4," [Online]. Available: https://blog.gradle.org/incremental-compiler-avoidance. [Hozzáférés dátuma: 15. október 2017.].

[8] R. M. Stallman, R. McGrath és P. D. Smith, „GNU make manual," Free Software Foundation, 2014..

[9] D. H. Martin és J. R. Cordy, „On the maintenance complexity of makefiles," *WETSoM '16 Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics,* pp. 50-56., 14 május 2016.

[10] A. Van Deursen, P. Klint és J. Visser, „Domain-specific languages: An annotated bibliography," *ACM Sigplan Notices,* pp. 26-36, június 2000.

[11] P. Hudak, „Building domain-specific embedded languages," *ACM Computing Surveys (CSUR),* december 1996.

[12] P. Hudak, „Modular domain specific languages and tools," in *Software Reuse, 1998,* Proceedings. Fifth International Conference, 1998.

[13] „JavacTask (Compiler Tree API)," [Online]. Available: https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/util/JavacTask.html. [Hozzáférés dátuma: 15. október 2017.].

[14] J. Juneau, Java 9 Recipes, Berkeley, CA: Apress, 2017.

12 Függelék

12.1 Java C *header* fájl generátor osztály

```java
package pack;

import java.lang.annotation.Native;

public class NativeClass {
	/**
	 * Add two numbers and return with the result.
	 *
	 * @param a
	 *            Left operand.
	 * @param b
	 *            Right operand.
	 * @return The operands mathematically added to each other.
	 */
	private native int add(int a, int b);

	private native int add(long a, long b);

	private native Thread method(Runnable r, NativeClass c);

	@Native
	public static final int CONSTANT = 123;
}
```

12.2 Generált C *header* fájl *javah* által

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class pack_NativeClass */

#ifndef _Included_pack_NativeClass
#define _Included_pack_NativeClass
#ifdef __cplusplus
extern "C" {
#endif
#undef pack_NativeClass_CONSTANT
#define pack_NativeClass_CONSTANT 123L
/*
 * Class:     pack_NativeClass
 * Method:    add
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_pack_NativeClass_add__II
  (JNIEnv *, jobject, jint, jint);

/*
 * Class:     pack_NativeClass
 * Method:    add
 * Signature: (JJ)I
 */
JNIEXPORT jint JNICALL Java_pack_NativeClass_add__JJ
  (JNIEnv *, jobject, jlong, jlong);
```

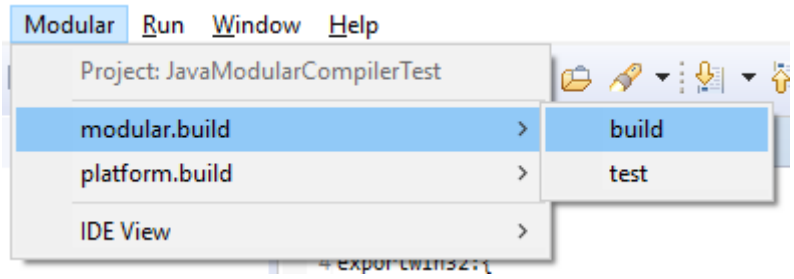
```
/*
 * Class:     pack_NativeClass
 * Method:    method
 * Signature: (Ljava/lang/Runnable;Lpack/NativeClass;)Ljava/lang/Thread;
 */
JNIEXPORT jobject JNICALL Java_pack_NativeClass_method
  (JNIEnv *, jobject, jobject, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

12.3 Generált C *header* fájl a rendszer által

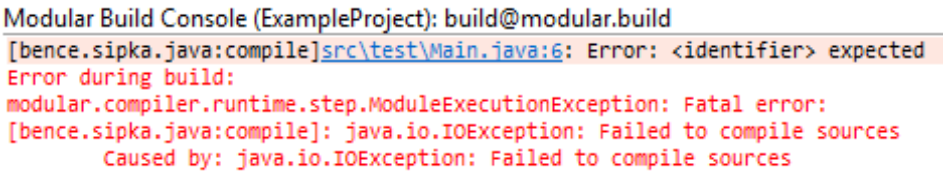
```
/* Compiler generated file */
/* Header for class pack.NativeClass */

#ifndef JAVA_NATIVE_pack_NativeClass_H_
#define JAVA_NATIVE_pack_NativeClass_H_
#include <jni.h>
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/**
 *  Add two numbers and return with the result.
 *
 *  @param a Left operand.
 *  @param b Right operand.
 *  @return The operands mathematically added to each other.
 *
 * Method: pack.NativeClass.add
 * Arguments:
 *            int a,
 *            int b
 * Return type: int
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_pack_NativeClass_add__II(JNIEnv* env, jobject obj,
jint a, jint b);

/**
 * Method: pack.NativeClass.add
 * Arguments:
 *            long a,
 *            long b
 * Return type: int
 * Signature: (LL)I
 */
JNIEXPORT jint JNICALL Java_pack_NativeClass_add__LL(JNIEnv* env, jobject obj,
jlong a, jlong b);

/**
 * Method: pack.NativeClass.method
 * Arguments:
 *            java.lang.Runnable r,
 *            pack.NativeClass c
 * Return type: java.lang.Thread
```

```
 * Signature: (Ljava/lang/Runnable;Lpack/NativeClass;)Ljava/lang/Thread;
 */
JNIEXPORT jobject JNICALL Java_pack_NativeClass_method(JNIEnv* env, jobject
obj, jobject r, jobject c);

/**
 * Type:     int
 * Constant: pack.NativeClass.CONSTANT
 * Value:    123
 */
#define Java_const_pack_NativeClass_CONSTANT ((jint) 123)


#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif /* JAVA_NATIVE_pack_NativeClass_H_ */
```

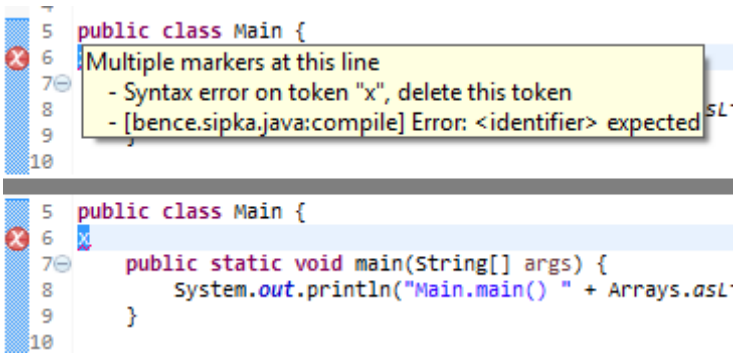
12.4 Eclipse kiterjesztés



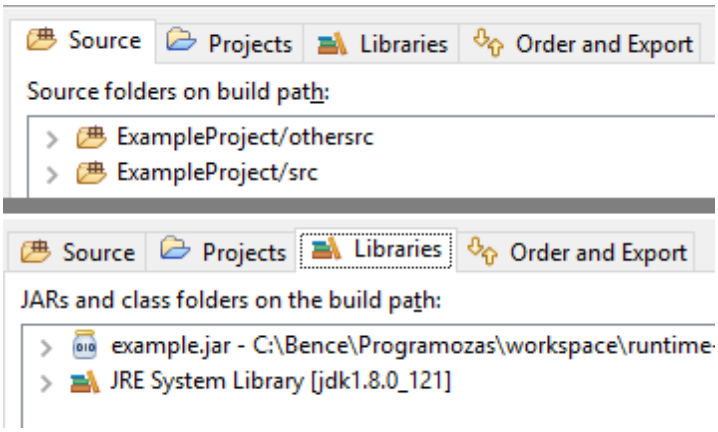
13. ábra: Projekt specifikus legördülő menü



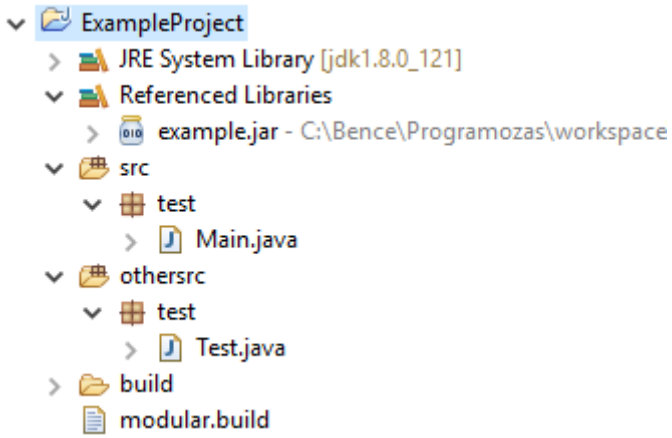
14. ábra: Fordítás kimenet hibával



15. ábra: Hibajelző markerek



16. ábra: Automatikusan konfigurált Java projekt



17. ábra: Példa projekt felépítése Eclipse-ben