OULUN AMMATTIKORKEAKOULU

Jani Sivonen

CMAKE-TIEDOSTOJEN ANALYSOINTI

CMAKE-TIEDOSTOJEN ANALYSOINTI

Jani Sivonen
Opinnäytetyö
Kevät 2023
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, laite- ja tuotesuunnittelun suuntautumisvaihtoehto

Tekijä: Jani Sivonen
Opinnäytetyön nimi: CMake-tiedostojen analysointi
Työn ohjaaja: Jukka Jauhiainen
Työn valmistumislukukausi ja -vuosi: Kevät 2023 Sivumäärä: 24

Toimeksiantaja valmistaa tuotteita, joiden ohjelmointiin käytetään C++-ohjelmointikieltä. Toimeksiantaja halusi tutkia kiertäviä riippuvuuksia ohjelmistoista. Jotta voitaisiin tutkia kiertäviä riippuvuuksia, on tiedettävä tutkittavan ohjelman liittyvät lähdekooditiedostot. Opinnäytetyön tarkoituksena oli luoda työkalu, joka lukee ohjelmistoprojektiin liittyvät CMakeLists.txt-tiedostot ja tulostaa näistä käyttäjälle tuotteeseen liittyvien lähdekooditiedostojen nimet hakemistopolkuineen. Opinnäytetyön tietoperustaan sisältyy kirjallisuutta ja internetissä julkaistuja artikkeleita. Tietoperusta koostuu suurimmalta osin englanninkielisistä julkaisuista.

Työkalu analysoi CMakeLists.txt-tiedostot käyttäjän syöttämästä hakemistopolusta. Käyttäjä määrittelee parametrit työkalun luomaan definitions.txt-tiedostoon. Parametrit säätelevät, minkä ohjelmiston käännösversion käyttäjä haluaa työkalun analysoivan. Se kykenee analysoimaan CMaken ehtolausekkeet ja siirtymään analysoimaan määriteltyjä alihakemistoja. Työkalu tuottaa käyttäjälle listauksen hakemistokohtaisesti tuotteeseen sisältyneet cpp-tiedostot.

Työkalua luodessa suurimpana haasteena oli CMake:n ehtolausekkeiden läpikäyminen ohjelmallisesti. Työssä huomattiin, että ohjelmointivirheiden löytäminen sisäkkäisten objektien sisältä tuotti haasteita.

Työkalun tehokkuus saataisiin parhaiten hyötykäytettyä muuntamalla työkalu kirjastoksi ja liittämällä se osaksi laajempaa staattista analyysia tekevää ohjelmistoa. Työkalun tuottaman datan muuntaminen tunnettuun formaattiin toisi mahdollisuuksia liittää data osaksi graafista käyttöliittymää.

Asiasanat: automaatiojärjestelmät, C++, olio-ohjelmointi, prototyypit

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Device and Product Design

Author: Jani Sivonen
Title of thesis: Analysis of CMake files
Supervisor: Jukka Jauhiainen
Term and year when the thesis was submitted: Spring 2023
Number of pages: 24

A global wireless network company required a comprehensive tool for analyzing circular dependencies within C++ code. In order to initiate the analysis of circular dependencies, an additional tool was necessary to extract source files from CMakeLists.txt. The objective of the thesis was to develop a tool capable of parsing CMake files and extracting source files from CMakeLists.txt files.

Upon compilation, the tool can be utilized by the user, who must specify the target path where the top-level directory is located. The tool's parameters can be customized by the user through the definitions.txt file. This file is automatically generated by the tool during the initial run. The program employs the definitions.txt file to identify the appropriate C++ product files referenced in the CMakeLists.txt files. The tool possesses the capability to analyze all specified subdirectories that are included in the product. Once the analysis is complete, the tool outputs the names of the subdirectories along with the files utilized by CMake from each respective directory.

Class definitions are typically encapsulated within hpp files, which can also include additional files. By utilizing the developed tool, it becomes possible to analyze the dependencies of cpp files, explore object definitions, and consequently obtain more accurate information regarding dependencies. This approach facilitates the identification of nested classes, as well as all global object and variable dependencies.

Keywords: programming languages, prototyping, automation systems

SISÄLLYS

1 JOHDANTO

Toimeksiantajana opinnäytetyölle on kansainvälinen langattomaan verkkoliikenteeseen keskittynyt yritys. Yrityksen toiminta on maailmanlaajuista. Yrityksellä on Euroopassa 80 toimipistettä ja niissä työskentelee 37 700 henkilöä. Toimeksiantaja valmistaa tuotteita, joiden ohjelmointiin käytetään C++-ohjelmointikieltä. Toimeksiantaja halusi tutkia kiertäviä riippuvuuksia ohjelmistoista. Tähän tarkoitukseen toimeksiantaja tarvitsee työkalun, jonka avulla haetaan ohjelmistoprojektiin liittyvät lähdekooditiedostot.

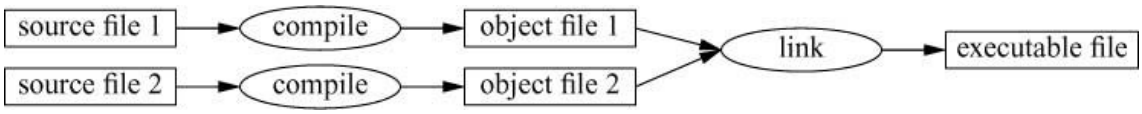
Opinnäytetyön tarkoituksena oli luoda työkalu, joka lukee ohjelmistoprojektiin liittyvät CMakeLists.txt-tiedostot ja tulostaa näistä käyttäjälle tuotteeseen liittyvien lähdekooditiedostojen nimet hakemistopolkuineen. Tuotantokoodin automaattiseen kääntämiseen käytetään CMakea (1). Tavoitteena on tallentaa CMaken konfiguraatiotiedostoista cpp-päätteiset tiedostonimet hakemistopolkuineen. Jotta työkalu voi valita oikean tuotteen konfiguraatiotiedostoista, on käyttäjän määriteltävä sille sopivat parametrit. Viimeisenä työkalun halutaan tulostavan terminaaliin lähdekooditiedostot hakemisto polkuineen. Opinnäytetyö toteutetaan C++-ohjelmointikielellä ja käännöksen automatisointiin käytetään CMakea. (2.)

2 C++-OHJELMOINTIKIELI JA OLEELLISET TEKNIIKAT

2.1 C++-kielen tausta

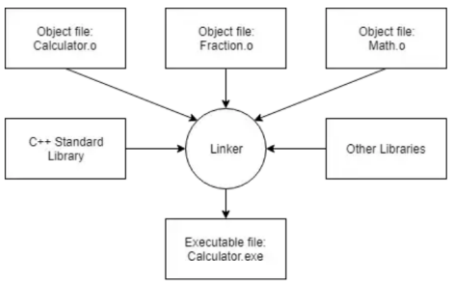
Tanskalainen Bjarne Stroustrup loi C++-kielen vuonna 1979 työskennellessään AT&T Bell Laboratories-yhtiössä. Hän kirjoitti kielen ensimmäiset määritelmät ja loi ensimmäisen implementoinnin kieleen. Kielen kehityksen alkuvaiheessa sitä nimitettiin C with Classes. Vuonna 1983 C++ nimi otettiin käyttöön Rick Mascittin ehdotuksesta. Vuonna 1985 kieli vapautettiin käytettäväksi AT&T Bell Laboratories ulkopuolelle. (3; 4.)

C++-kieli on käännettävä kieli. Kääntäjä on ohjelmisto, joka muuntaa lähdekooditiedostot tekstimuodosta tietokoneen ymmärtämään binäärimuotoon (5). Jotta ohjelman käyttö onnistuu kohdelaitteella, on lähdekooditiedostot prosessoitava kääntäjän avulla. Kuvassa 1 kuvataan miten kääntäjä tuottaa objektitiedostoja, jotka yhdistetään linkitysohjelman avulla käynnistettäväksi ohjelmaksi. (4.)



KUVA 1. Useiden lähdekoodi tiedostojen kääntäminen käynnistettäväksi ohjelmaksi (4)

Linkitysohjelma (engl. linker) on ohjelmisto, joka tarvitaan käännösvaiheen jälkeen (Kuva 2). Linkitysohjelma toimii kolmessa vaiheessa. Ensimmäiseksi linkitysohjelma yhdistää kaikki luodut objektitiedostot käynnistettäväksi ohjelmaksi. Toisessa vaiheessa linkitysohjelma yhdistää ohjelman valmiisiin ohjelmistokirjastoihin, jos näin on lähdekoodissa määritelty. Kolmannessa eli viimeisessä vaiheessa linkitysohjelma tarkistaa, että riippuvuudet objektien välillä on määritelty oikein. (6.)



KUVA 2. Linkitysohjelman toiminta (5)

2.2 Riippuvuudet C++-kielessä

Riippuvuus (engl. dependencies) muodostuu C++-kielessä, kun objektitiedosto käyttää toisen objektitiedoston toiminnallisuutta suorittaakseen tietyn tehtävän. Riippuvuus on aina yksisuuntaista. (7.) C++-kielessä riippuvuuksien hallinta on haastavaa, koska niistä ei ole standardeja eikä pakettien hallintaohjelmistoa kuten useissa muissa kielissä. C++-kieleen kehitetään useiden eri toimijoiden toimesta riippuvuuksien hallintaohjelmistoja. (8.)

Riippuvuus voi olla kiertävää riippuvuutta (engl. circular dependency). Kiertävä riippuvuus muodostuu tilanteessa, jossa TiedostoA on riippuvainen TiedostonB:n toiminnallisuudesta ja TiedostoB vastaavasti TiedostoA:n toiminnallisuudesta, mikä selviää alapuolella olevasta (Kuva 3). (9.)

KUVA 3. Kiertävä riippuvuus TiedostoA:n ja TiedostoB:n välillä

Yleisesti ohjelmiston kehityksessä kiertävän riippuvuuden muodostumista ei pidetä hyvänä asiana, vaikka niitä muodostuu usein (10). Yleisimpiä käyttötapauksia, joissa kiertäviä riippuvuussuhteita muodostuu:

- Luokan määrittelyssä virheellisen vastuun toteutuminen eli luokan jäsenet ovat väärin määritelty, jolloin jäsenet voivat viitata toisiinsa.
- Itseviittauksen välittäminen eli menetelmäkutsu luokasta toiseen sisältää usein tiedonsiirtoa. Jos sen sijaan, että nimenomaisesti välitettäisiin vain vaaditut tiedot, luokka välittää oman viittauksensa toisen luokan menetelmään, syntyy tästä kiertävä riippuvuus.
- Takaisinkutsutoiminnon käyttöönotto eli kiertävä riippuvuus syntyy tarpeettomasti kahden luokan välillä takaisinkutsutoimintoa toteutettaessa. Tämä johtuu siitä, että kokemattomat kehittäjät eivät ehkä tunne hyviä ratkaisuja, kuten suunnittelumalleja.

- Vaikeasti visualisoitavat epäsuorat riippuvuudet eli monimutkaisissa ohjelmistojärjestelmissä suunnittelijoiden on haasteellista visualisoida luokkien välisiä riippuvuussuhteita. Tämän seurauksena suunnittelijat päätyvät vahingossa luomaan kiertäviä riippuvuuksia luokkien välille. (11.)

2.3 Rekursio ohjelmointikielessä

Rekursion toiminta perustuu siihen, että aliohjelma tai funktio kutsuu itseään uudestaan tiettyjen ehtojen perusteella (12). Kuvassa 4 on yksinkertainen esimerkki rekursiivisesta funktiosta ja sen käyttämisestä.

```
1  #include <iostream>
2
3  void recursio(int n)
4  {
5      std::cout<<n<<std::endl;
6      n++;
7      if (n<10) {
8          recursio(n);
9      }
10 }
11
12 int main()
13 {
14     int n = 0;
15     recursio(n);
16     return 0;
17
18 }
```

KUVA 4. Yksinkertainen rekursiivinen funktio ja kutsu

Kuvassa 4 ohjelman main-funktiosta kutsutaan kerran recursio-funktiota, joka suorittaa itsensä kutsumista, kunnes n-muuttujalle määritelty ehto ei täyty. (13.)

3 OHJELMISTON KÄÄNTÄMISEN AUTOMATISOINTI

Ohjelmiston automaattiseen kääntämiseen on useita eri ohjelmistoja: Make, Autotools, SCons, Ninja, Premake, CMake (14). Automatisointia käytetään, jotta riippuvuussuhteiden tarkistus ja käännöksen toistettavuus on mahdollista. Eri käännösversiot ohjelmasta määritellään konfiguraatiotiedostojen avulla. Automaattiset käännösohjelmistot säästävät aikaa ja resursseja varsinkin laajoissa projekteissa. (15.)

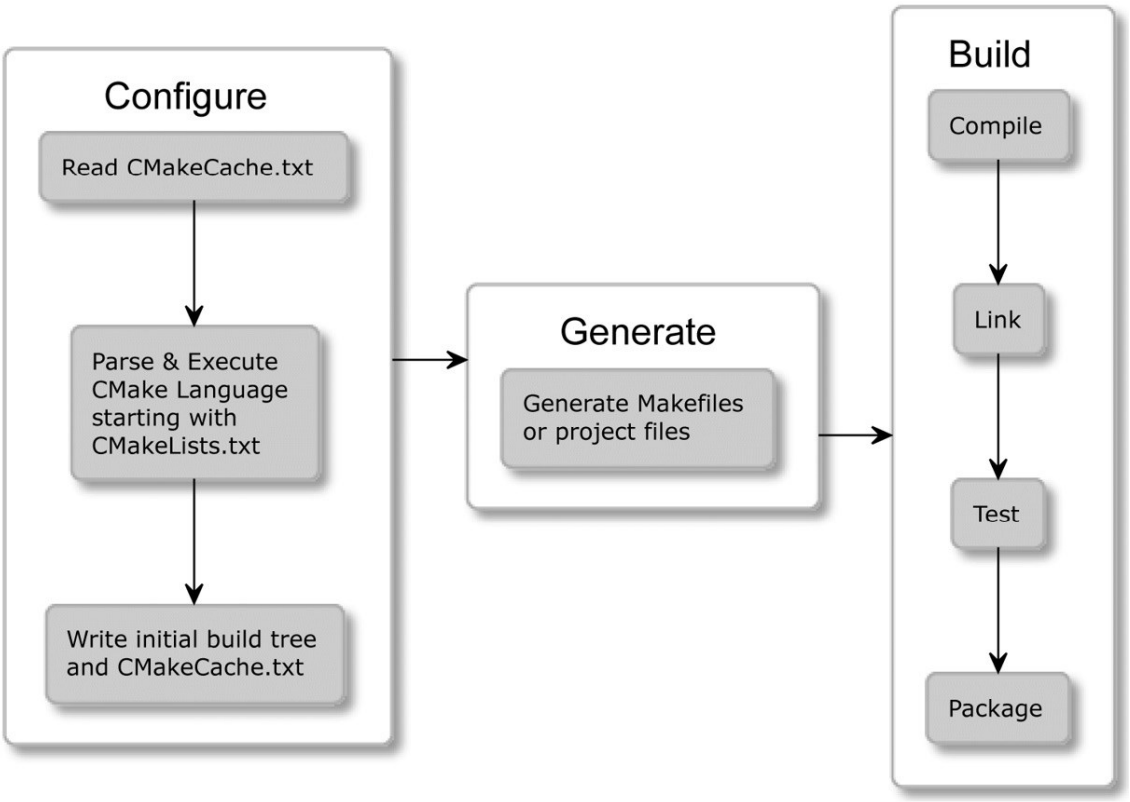
3.1 CMaken toiminta

Bill Hoffman kehitti CMakea vuonna 2000 työskennellessään Kitwarella. Vaikutteita CMaken kehitykseen, Hoffman otti Pcmaker- ja Unix configure-työkaluista (16). CMake on generaattori, joka luo tarvittavat tiedostot projektin kääntämiseksi. Generaattori on käyttöjärjestelmä riippumaton, joten sen avulla voidaan kääntää ohjelmistoja eri käyttöjärjestelmille ja järjestelmäarkkitehtuureille. Kehittäjä määrittelee projektin rakenteen vaadittavalla tavalla. (17.)

CMake kykenee esimerkiksi:

- kääntämään ajettavia ohjelmia ja kirjastoja
- hallitsemaan riippuvuuksia
- testaamaan
- asentamaan
- paketoimaan
- luomaan dokumentaatiota.

CMake ei sisällä sisäänrakennettua kääntäjää. CMake käyttää käyttöjärjestelmään asennettuja työkaluja käännösprosessin suorittamiseksi. CMaken kokonaisprosessi jakautuu kolmeen alueeseen (Kuva 5): konfigurointiin, generointiin ja kääntämiseen. (14.)



KUVA 5. CMaken kokonaisprosessin vaiheet (14)

Konfigurointivaiheessa luetaan projektin yksityiskohdat hakemistorakenteesta ja luodaan hakemisto käännettävälle ohjelmistolle. Aluksi CMake luo tyhjän prototyyppiversion puurakenteen sekä tallentaa käyttöjärjestelmän tiedot. Tallennettavat tiedot ovat käytettävissä oleva kääntäjä, linkittäjäohjelmisto ja käyttöjärjestelmänarkkitehtuuri. Tämän jälkeen CMake suorittaa CMakeLists.txt konfiguraatiotiedostot ja luo CMakeCache.txt-tiedoston. Suorituksen tuloksena saatu data ilmaisee CMakelle projektin rakenteen, sen kohteet ja riippuvuussuhteet. (14.)

Generointivaiheessa CMake luo käännösjärjestelmän siihen ympäristöön, missä CMakea käytetään. Tämän vaiheen aikana CMake muokkaa vielä konfiguraatiotietoja, jotka edellisessä vaiheessa on luotu. (14.)

Seuraavassa vaiheessa ohjelma suorittaa määritellyt käännösvaiheet, jotka luovat määritellyn ohjelman. Tässä vaiheessa pystytään myös käynnistämään staattisia ja dynaamisia analysointityökaluja, raportointityökaluja ja muita vastaavia toimenpiteitä. (14.)

3.2 CMaken lähdekooditiedostojen ja kirjastojen lisääminen

CMake-ohjelmointikielessä määritellään käännettävä ohjelma kolmella eri komennolla: add_executable, add_library ja add_custom_target. Komento add_executable vastaanottaa ensimmäisenä parametrina käännettävän ohjelman nimen. Sen jälkeen seuraavat parametrit ovat lähdekooditiedostojen nimiä hakemistopolkuineen. (14.)

CMakessa pystytään luomaan sisäisiä kirjastoja käyttämällä komentoa add_library. Siihen määritellään parametreina kirjaston nimi, tyyppi ja lähdekooditiedostot hakemistopolkuineen. Kirjasto linkitetään käännettävään ohjelmaan käyttäen komentoa target_link_libraries, johon annetaan parametrina käännettävän ohjelman nimi, näkyvyys ja kaikkien linkitettävien kirjastojen nimet. (14.)

Projektin lähdekooditiedostojen jakautuessa alihakemistoihin (Kuva 6) pystytään alihakemisto liittämään projektiin komennolla add_subdirectory. Siihen määritellään parametrina alihakemiston nimi (Kuva 7). Alihakemistoon luodaan uusi CMakeLists.txt-tiedosto (Kuva 8), jotta käännöksen lähdekooditiedostot ovat käytettävissä. (14.)

```
├── CMakeLists.txt
├── cars
│   ├── CMakeLists.txt
│   ├── car.cpp
│   └── car.h
└── main.cpp
```

KUVA 6. Hakemistorakenne (14)

```
cmake_minimum_required(VERSION 3.20.0)
project(Rental CXX)
add_executable(Rental main.cpp)
add_subdirectory(cars)
target_link_libraries(Rental PRIVATE cars)
```

KUVA 7. Päätason CMakeLists.txt (14)

```
add_library(cars OBJECT
    car.cpp
#   car_maintenance.cpp
)
target_include_directories(cars PUBLIC .)
```

KUVA 8. Alihakemiston CMakeLists.txt (14)

3.3 CMaken muuttujat

CMakessa on kolme erityyppistä muuttujaa: normaali-, välimuisti- ja ympäristömuuttujat. Jokaisella muuttujalla on oma näkyvyytensä koodissa. Muuttujia voidaan muokata komennoilla set ja unset (Kuva 9) ja niihin pystytään vaikuttamaan string- ja list-komennoilla. (14.)

```
set(MyString1 "Text1")
set([[My String2]] "Text2")
set("My String 3" "Text3")
```

KUVA 9. Muuttujien asetus CMakeLists.txt tiedostossa (14)

Käyttäjät pystyvät asettamaan välimuistimuuttujia suoraan terminaalista (Kuva 10). Tyypin määritteleminen ei ole pakollista, kun terminaalikomennolla määritellään muuttujia. Jos ne määritellään, tyypit ovat BOOL, FILEPATH, PATH, STRING tai INTERNAL. (14.)

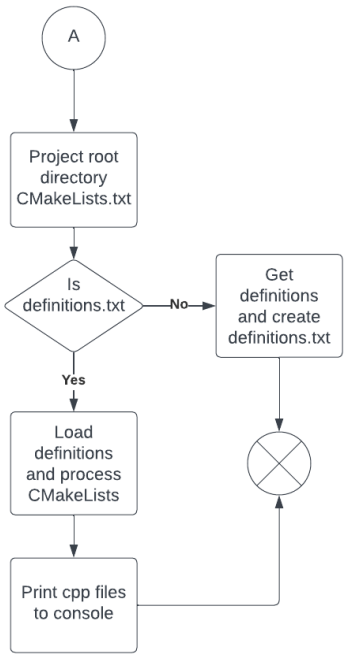
```
cmake -D <var>[:<type>]=<value> <path-to-source>
```

KUVA 10. Välimuistimuuttujan määrittely terminaalista (14)

4 ANALYSOINTITYÖKALUN KUVAUS

4.1 Analysointityökalun toiminta

Jotta projektissa käytettävät lähdekooditiedostot tiedettäisiin, täytyy työkalun analysoida CMakeLists.txt-tiedostot. Kun analysointityökalu käynnistetään terminaalista, täytyy siihen syöttää parametrina analysoitavan projektin hakemistopolku. Työkalu tarkistaa, onko ympäristömuuttujat asetettu definitions.txt-tiedostoon. Jos työkalu ei löydä definitions.txt-tiedostoa, se aloittaa analysoimaan päätason CMakeLists.txt-tiedostoa. Tämän jälkeen se luo definitions.txt-tiedoston ja tallentaa kaikki CMakeLists.txt-tiedostossa määritellyt ympäristömuuttujat definitions.txt-tiedostoon. Jos työkalu generoi definitions.txt-tiedoston, käyttäjän pitää määritellä, mitä ympäristömuuttujia käytetään analysointivaiheessa. Jos työkalu havaitsee definitions.txt-tiedoston, tallennetaan ympäristömuuttujat sen sisäiseen muistiin ja aloitetaan projektin analysointi. Analysoinnin päättyessä työkalu tulostaa terminaaliin muuttujien määrittelyt ja hakemiston, missä muuttujat ovat määritelty. Analysointityökalun toimintaperiaate on havainnollistettu alla (Kuva 11.).



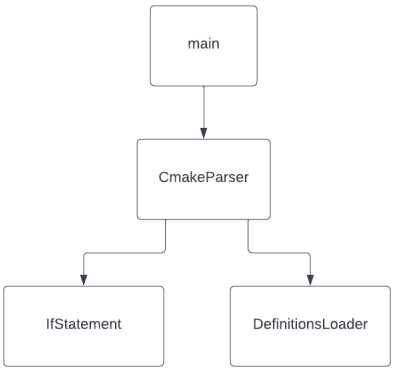
KUVA 11. Analysointityökalun toimintaperiaate

Analysointityökaluun sisällytettiin toiminnallisuus, joka kykenee analysoimaan CMake-ohjelmointikielellä kirjoitettuja ehtolausekkeita. Ehtolausekkeista tallennetaan analysointityökalun muistiin muuttujat ja alihakemisto määrittelyt. Lausekkeen ehdon täytyy olla tosi, jotta se tallentuu analysointityökaluun. Siihen määriteltiin kolme avainsanaa. Niiden avulla tallennetaan muuttujia tai alihakemistoja työkalun muistiin. Määritellyt avainsanat ovat set, SET ja add_subdirectory. Avainsanat SET ja set lisäävät muuttujia ja avainsana add_subdirectory lisää uuden alihakemiston.

Työkalu analysoi päätason CMakeLists.txt-tiedoston ja siirtyy analysoimaan siinä määriteltyjä alihakemistoja. Ympäristömuuttujat toimivat ainoastaan kaikissa CMakeLists.txt-tiedostoissa analysoinnin aikana. Yksittäisessä CMakeLists.txt-tiedostossa voidaan kuitenkin määritellä kyseisen tiedoston analysointiin käytettäviä muuttujia. Kun kaikki CMakeLists.txt-tiedostot on käsitelty, tulostaa työkalu tulokset terminaaliin.

4.2 Analysointityökalun rakenne

Analysointityökalu kirjoitettiin C++-kielellä käyttäen olio-ohjelmointitekniikoita. Työkalu koostuu useista tiedostoista, joihin kuuluvat lähdekooditiedostot cpp ja otsikkotiedostot hpp. Analysointityökalun luokkarakenne on nähtävissä Kuvassa 11. Funktio ”main” luo luokan CmakeParser ja se sisältää IfStatement- ja DefinitionsLoader- luokan.

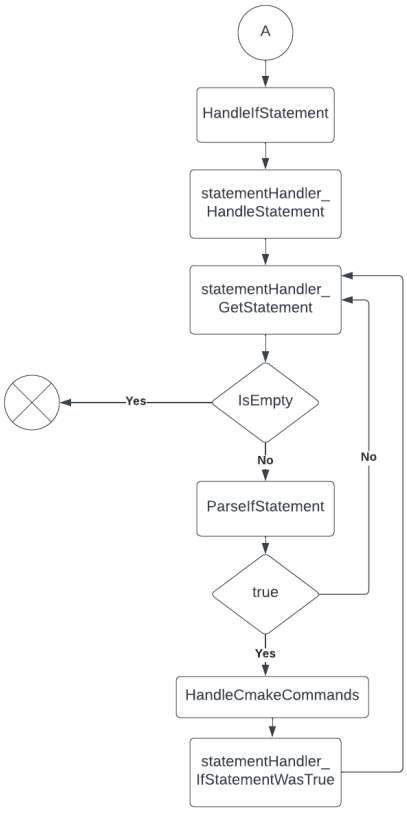


KUVA 11. Analysointityökalun luokkarakenne

Analysointityökalun CmakeParser-luokkaan sisällettiin toiminnallisuus, jolla pystytään palauttamaan siihen tallennetut alihakemistot ja muuttujat. Toiminnallisuus mahdollistaa sen, että palaute-

tut alihakemistot kyetään analysoimaan luomalla uusi CmakeParser-luokka, johon välitetään parametrina alihakemiston hakemistopolku. Metodia toistetaan, kunnes työkalu on analysoinut kaikki alihakemistot.

CmakeParser-luokan metodi lukee CMakeLists.txt-tiedoston rivi kerrallaan. Jos metodi havaitsee ehtolauseen, se yhdistää usean rivin ehtolauseen yhdeksi riviksi. Muunnettu rivi välitetään CmakeParser-luokan HandleIfStatement-metodille käsiteltäväksi. Tässä luodaan IfStatement-luokka, jolle rivi välitetään käyttämällä HandleStatement-metodia. Se pilkkoo ehtolausekkeen ja tarvittaessa luo uuden IfStatement-luokan omaan muistiin, jos ehtolausekkeessa on sisäkkäisiä ehtolausekkeita. ParseIfStatement-metodin toimintaperiaate on kuvassa 12.



KUVA 12. HandleIfStatement-metodin toimintaperiaate CmakeParser-luokassa

IfStatement-luokan GetStatement-metodia kutsutaan, kunnes luokka palauttaa tyhjän rivin. Jos ehtolauseen paluuarvo IfStatement-luokalle on epätosi, palauttaa tämä yhden osan ehtolauseesta. Ehtolausekkeiden ehdot tarkistetaan CmakeParser-luokassa. CmakeParser-luokan metodi selvittää, onko kyseessä if-, elseif- tai else-lauseke. Tämän perusteella lauseke ohjataan oikealle käsittelijälle.

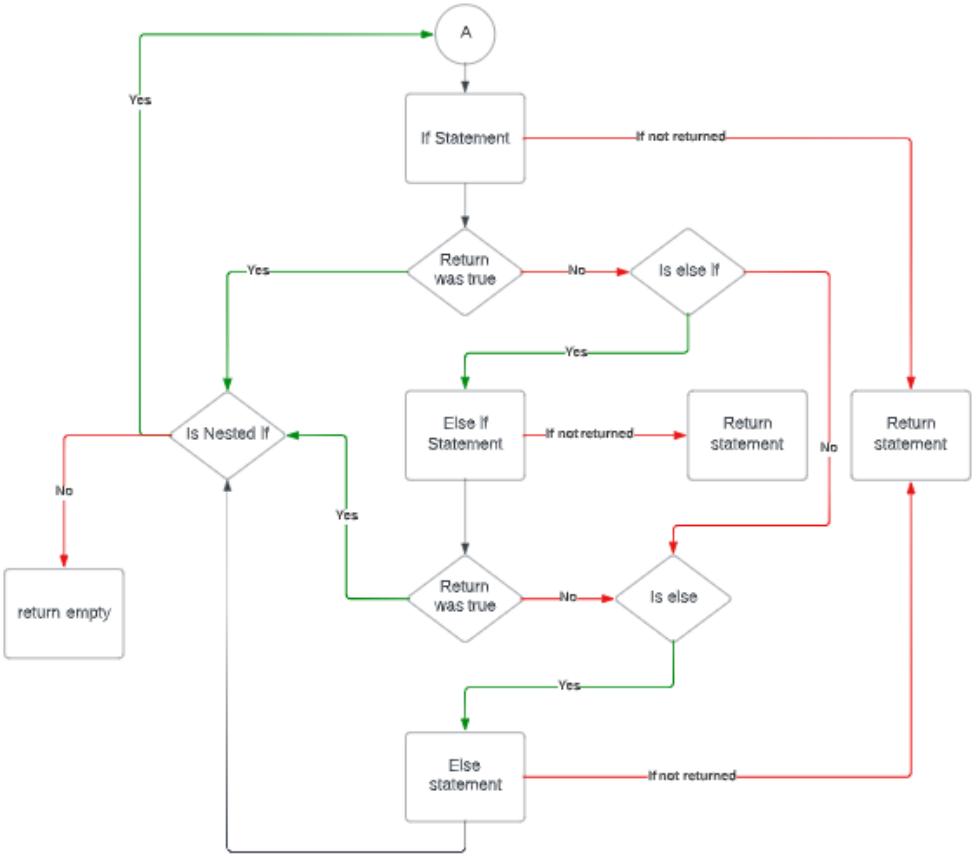
Käsittelijä käyttää CmakeParser-luokan ParseIfStatement-metodia, mikä tarkistaa, onko ehtolauseke tosi vai epätosi. Ehtolausekkeen ollessa tosi siirretään lauseke HandleCmakeCommands-metodille. Metodi lisää muuttujat ja alihakemistot analysointityökalun muistiin. Tässä vaiheessa ilmoitetaan IfStatement-luokalle ehtolausekkeen paluuarvo. IfStatement-luokka vaatii palautteen palautetusta ehtolausekkeesta. Palautteen avulla luokka valitsee seuraavan palautettavan ehtolausekkeen osan.

HandleIfStatement-metodin käsittely päättyy, kun kaikki ehtolausekkeiden osat on käsitelty tai jokin käsitelty ehtolausekkeen osa on tosi ja sen sisäiset lausekkeet on käsitelty. Sisäisiä lausekkeita voi olla ääretön määrä ja niiden pitää olla käsitelty.

DefinitionsLoader-luokka tarkistaa, onko projektista luotu definitions.txt-tiedostoa. Jos sitä ei ole, luokka luo tiedoston ja analysoi CMakeLists.txt-tiedoston. Analysoinnin jälkeen luokka tallentaa muuttujat definitions.txt-tiedostoon. Jos tiedosto löytyy, DefinitionsLoader-luokka tallentaa muuttujat muistiin.

IfStatement-luokka käsittelee ehtolausekkeet, jotka on sille välitetty. Ensimmäisenä luokka pilkkoo ehtolausekkeen pienempiin osiin. Sisäkkäisten ehtolausekkeiden käsittelyssä IfStatement-luokka luo itsestään uuden luokan, joka tallennetaan muistiin. Uudelle luokalle välitetään parametrina sisäkkäinen ehtolauseke. Tämä mahdollistaa ehtolausekkeiden palautuksen sisäkkäisistä luokista. Metodin avulla IfStatement-luokasta pystytään palauttamaan yksi ehtolauseke. Metodin käyttäjän täytyy vastata IfStatement-luokalle, oliko kyseinen ehtolauseke tosi vai epätosi.

Jos palautettu ehtolauseke on tosi, tarkistaa IfStatement-luokka, sisältääkö palautettu ehtolauseke sisäkkäistä ehtolauseketta. Jos sitä ei löydy, palauttaa IfStatement-luokka itsensä alkutilanteeseen sekä tyhjän rivin metodin käyttäjälle. Jos sisäinen lauseke löytyy, palautetaan sisäkkäisestä IfStatement-luokasta ehtolauseke. Tässä vaiheessa käyttäjän luoma IfStatement-luokka toimii ainoastaan tiedonvälittäjänä sisäkkäiselle luokalle. Tämä toteutuu, kunnes kaikki ehtolauseet ovat saaneet tosi vastauksen ja sisäisiä ehtolauseita ei löydy tai ne on käsitelty. Alla on kuvattu IfStatement-luokan toimintaperiaate (Kuva 13.).



KUVA 13. IfStatement-luokan toimintaperiaate

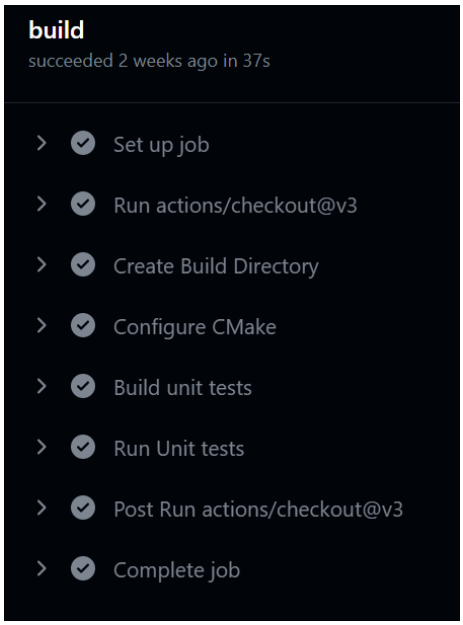
4.3 Analysointityökalun testaus

Työkalun lähdekoodiin sisällytettiin analysointityökalussa olevien luokkien metodeja varten tehdyt testitiedostot. Analysointityökalun CMakeLists.txt-tiedostoihin sisällytettiin muuttuja UT, johon käyttäjä kirjaa arvon komentorivillä. Tämän jälkeen käännösautomaatio luo suoritettavia testiohjelmia. Testiohjelmat testaavat luokkiin kuuluvien metodien toimintaa. Testiohjelmat varmistavat, etteivät uudet muutokset muuta metodien suunniteltua toimintatapaa. Testiohjelmien suoritus tulostaa terminaaliin testauksen tuloksen. Kuvassa 14 on Github actions -virtuaalikoneella suoritetut analysointityökalua testaavat testiohjelmat (18).

```
826  [----------] Global test environment tear-down
827  [==========] 49 tests from 1 test suite ran. (9 ms total)
828  [  PASSED  ] 49 tests.
```

KUVA 14. Kaikki 49 testiä ajettu Github actionsin avustuksella onnistuneesti

Githubissa pystyttiin luomaan virtuaalinen Linux-ympäristö, missä pystyttiin kääntämään ohjelmia (kuva 15). Jokainen uusi muutos lähdekoodiin voitiin tämän avulla automatisoida testattavaksi. Tämä helpotti uusien toiminnollisuuksien tarkkailua ohjelmistokokonaisuudessa.



KUVA 15. Github actions käynnistää virtuaalikoneen, kääntää ohjelman yksikkötestit ja suorittaa yksikkötestit

Testaus ohjelmoitiin käyttämällä GoogleTestiin integroitua ohjelmoinnin tukiympäristöä ja C++-ohjelmointikieltä (19). Testaus jaoteltiin luokkakohtaisesti testitiedostoihin. Jotta testaaminen on helpompaa, osa metodien ja muuttujien näkyvyyksistä on muutettu private-muodosta protected -muotoon. Tämä mahdollisti näiden muutettujen metodien ja muuttujien perimisen uuteen luokkaan. Samalla uuteen luokkaan luotiin julkinen rajapinta testausmetodeille. Käyttämällä uuden luokan julkista rajapintaa pystyttiin testaamaan perityn luokan protected -näkyvyyden metodeja. Näin vältettiin luomasta ylimääräisiä julkisia metodeja testattavaan luokkaan.

4.4 Lähdekoodista analysointityökaluksi

Analysointityökalu eli ohjelma käännettiin lähdekoodista ajettavaksi ohjelmaksi. Kääntäminen tapahtui Linux-terminaalissa. Linuxin täytyi sisältää CMake ja GCC-kääntäjä, jotta automaattinen kääntäminen onnistui (20). Lähdekoodin juurihakemistoon luotiin uusi alihakemisto, johon ohjelma haluttiin kääntää. Uudessa alihakemistossa kirjoitettiin terminaaliin seuraavasti:

- cmake -S ../ -B .

- cmake --build .

Tämä käänsi automaattisesti ajettavan ohjelman uuteen alihakemistoon.

Ohjelmaan oli mahdollista kääntää myös testiohjelmat, jolloin CMake -komennot kirjoitettiin seuraavasti:

- cmake -S ../ -B . -D UT=1
- cmake --build .

Cmake loi suoritettavat testiohjelmat luotuun alihakemistoon.

Kääntäminen tuottaa CircularDependency nimisen ajettavan ohjelman. Ohjelmaan välitetään parametrina analysoitava CMakeLists.txt-tiedoston absoluuttinen hakemistopolku. Ohjelma pyytää määrittelemään CMakeLists.txt-tiedostossa esiintyville muuttujille arvot. Määrittelyn jälkeen ohjelma tulostaa analysointi tuloksen Linux terminaaliin (Kuva 16).

```
Please define what you need and empty will not be defined
Give definition for UT, empty space will not define
FALSE
Variable: CMAKE_CXX_STANDARD
17
Variable: SOURCES
ParseSource.cpp
main.cpp
```

KUVA 16. CircularDependency-ohjelmalla analysoitu CircularDependency-ohjelman CMakeLists.txt-tiedosto

4.5 Kehitysideat

Analysointityökalu on prototyyppi. Sitä on mahdollista laajentaa tukemaan CMaken syntaksia laajemmin. Työkalun tehokkuutta hyödynnettäisiin paremmin muuntamalla työkalu kirjastoksi ja liittämällä se osaksi laajempaa staattista analyysia tekevää ohjelmistoa.

Käyttäjän käyttökokemusta parannettaisiin lisäämällä tuki useille yleisesti tunnetuille formaateille. Useat tunnetut formaatit kyetään siirtämään sellaisenaan graafisiin käyttöliittymiin. Formaatin avulla työkalun tuottaman datan muotoilu on helposti muunnettavissa eri käyttötarkoituksiin sopivaksi.

Analysointityökalun testaamista pitäisi laajentaa ja pyrkiä kattamaan kaikki rajatapaukset, jotka ohjelman suorituksessa pystyy tapahtumaan. Tämä helpottaisi ohjelman ylläpidettävyyttä ja jatkokehittämistä tulevaisuudessa. Koska kyseessä on prototyyppi, testauksen kattavuus ei ollut opinnäytetyön keskiössä.

5 POHDINTA

Opinnäytetyön tarkoituksena oli luoda työkalu, joka lukee ohjelmistoprojektiin liittyvät CMakeLists.txt-tiedostot ja tulostaa näistä käyttäjälle tuotteeseen liittyvien lähdekooditiedostojen nimet hakemistopolkuineen. CMake mahdollistaa useiden käännösversioiden luomisen, ja tämä piti huomioida analysointityökalua kehittäessä. Kyseinen ominaisuus lisäsi CMake-tiedostojen läpikäymisen kompleksisuutta, kun tiedostoista ei saatu noudettua suoraan lähdekooditiedostoja. CMakella on oma ohjelmointikielensä, mikä mahdollistaa ehtolauseiden luomisen. Niiden avulla kyetään ohjaamaan sitä, miten ohjelma käännetään. Ehtolauseiden läpikäyminen ohjelmallisesti vaati alkuperäistä aihetta enemmän resursseja. Tämän vuoksi opinnäytetyö jouduttiin rajaamaan. Se rajattiin koskemaan ainoastaan CMakeLists.txt-tiedostoista löytyviin projektiin kuuluvien lähdekooditiedostojen listaamiseen.

Opinnäytetyö syvensi osaamistani CMaken syntaksista. Tutustuin syvällisesti alihakemistojen lisäämiseen ja muuttujien toiminnallisuuteen. Aikaisemmin en ollut tutustunut kirjastojen luomiseen alihakemiston CMakeLists.txt-tiedoston avulla. Ehtolausekkeiden monimuotoisuus yllätti minut täysin, ja niiden käsittely syvensi osaamistani rekursiosta.

Ennen tätä työtä en ollut ohjelmoinut luokkaa, joka käyttää rekursiota sisäisen luokan tilan muuttamiseen. Havaitsin, kuinka haastavaa on rekursiivisten kutsujen seuraaminen etenkin virhetilanteissa. Virheiden etsimisen teki haasteelliseksi se, etteivät sisäkkäiset luokat eroa koodillisesti toisistaan. Virheiden havaitsemiseen käytin terminaalitulostuksia, joiden avulla selvitin, missä sisäkkäisessä luokassa ohjelman suoritus oli virhetilanteessa.

Yksikkötestauksen tehokkuus osoittautui erittäin hyödylliseksi uusien ominaisuuksien lisäämisessä. Luokkien metodeja pystyi varmemmin muuttamaan ilman pelkoa siitä, että aikaisempi toiminnallisuus kärsii uusista ominaisuuksista. Testauksen automatisointi CMakella nopeutti kehitystyön etenemistä ja Github actions toimi varmistajana, etteivät testit ole jääneet suorittamatta paikallisessa ympäristössä.

LÄHTEET

1. Kitware inc 2023. CMake Tutorial. Hakupäivä 15.5.2023. https://cmake.org/cmake/help/latest/guide/tutorial/index.html.
2. Simula, Kirsti 2023. SW Architect. Nokia Solutions and Networks Oy. Puhelinkeskustelu 13.3.2023.
3. Volle, Adam 2022. C++ computer language. Hakupäivä 12.4.2023. https://www.britannica.com/technology/C-computer-language.
4. Stroustrup, Bjarne 2013. The C++ Programming Language. Fourth Edition. Boston: Addisson-Wesley. Hakupäivä 12.4.2023. O'Reilly for Higher Education. Vaatii käyttöoikeuden.
5. cplusplus.com 2023. Compilers. Hakupäivä 12.4.2023. https://cplusplus.com/doc/tutorial/introduction/.
6. Alex 2019. 0.5 — Introduction to the compiler, linker, and libraries. Hakupäivä 12.4.2023. https://www.learncpp.com/cpp-tutorial/introduction-to-the-compiler-linker-and-libraries/.
7. Alex 2022. 16.5 — Dependencies. Hakupäivä 12.4.2023. https://www.learncpp.com/cpp-tutorial/dependencies/.
8. Kirsh, Amir 2021. About C++ Dependency Management. Hakupäivä 12.4.2023. https://www.incredibuild.com/blog/about-cpp-dependency-management.
9. Pvigier 2018. Circular Dependencies in C++. Hakupäivä 14.4.2023. https://pvigier.github.io/2018/02/09/dependency-graph.html.
10. Bendersky, Eli 2013. Library order in static linking. Hakupäivä 12.4.2023. https://eli.thegreenplace.net/2013/07/09/library-order-in-static-linking.
11. Suryanarayana, Girish, Samarthyam, Ganesh & Sharma, Tushar 2014. Refactoring for Software Design Smells. Managing Technical debt. Philadelphia: Elsivier Inc. Hakupäivä 13.4.2023. O'Reilly for Higher Education. Vaatii käyttöoikeuden.
12. Laaksonen, Antti 2004. Hakupäivä 18.4.2023 https://www.ohjelmointiputka.net/oppaat/opas.php?tunnus=rekursio.
13. Manga 2013. cplusplus.com. Hakupäivä 18.4.2023 https://cplusplus.com/articles/D2N36Up4/.
14. Świdziński, Rafał 2022. Modern CMake for C++. Birmingham: Packt Publishing Ltd. Hakupäivä 18.4.2023. O'Reilly for Higher Education. Vaatii käyttöoikeuden.

15. Sutter, Herb & Alexandrescu, Andrei. C++ Coding Standards. 101 Rules, Guidelines, and Best Practices. Boston: Addisson-Wesley. Hakupäivä 19.4.2023. O'Reilly for Higher Education. Vaatii käyttöoikeuden.
16. Cmake. About CMake. Hakupäivä 13.4.2023 https://cmake.org/overview/.
17. Gomes, Matheus. CMake – A Simple Introduction. Hakupäivä 13.4.2023. https://matgomes.com/a-simple-introduction-to-cmake/.
18. GitHub 2023. GitHub actions documentation. Hakupäivä 15.5.2023. https://docs.github.com/en/actions.
19. GoogleTest. GoogleTest User's Guide. Hakupäivä 15.5.2023. http://google.github.io/googletest/.
20. Free Software Foundation Inc. GCC, the GNU Compiler Collection. Hakupäivä 15.5.2023 .https://gcc.gnu.org/.