Eetu Kiiskilä

Clang-Tidyn laajentaminen yrityksen tarpeiden mukaisesti

Insinööri (AMK)
Tieto- ja viestintätekniikka
Syksy 2023

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Kiiskilä Eetu

Työn nimi: Clang-Tidyn laajentaminen yrityksen tarpeiden mukaisesti

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: Clang-Tidy, koodianalyysi, ohjelmointikäytänteet, ohjelmistoautomaatio, LLVM, C++

Tässä opinnäytetyössä kehitettiin laajennettu versio Clang-Tidysta, joka on osa LLVM-projektia. Tavoitteena oli lisätä työkaluun vähintään yksi uusi tarkistus sekä automaatiotestit ja dokumentaatio tälle. Tarkoituksena oli, että toimeksiantaja Bittium Wireless Oy voisi opinnäytetyön avulla arvioida, onko Clang-Tidyn kehittäminen yrityksen tarpeiden mukaan järkevää. Lisäksi opinnäytetyö toimisi tarvittaessa ohjeena uusien tarkistusten lisäämiselle työkaluun.

Aluksi tutustuttiin ohjelmointikäytänteiden teoriaan ja siihen, millaisia työkaluja ohjelmointikäytänteisiin liittyen on olemassa. Lisäksi tarkasteltiin toimeksiantajan tarpeita koodianalyysityökaluille. Clang-Tidyn käyttöönotto kuvattiin yksinkertaisen CMake-pohjaisen C++-projektin avulla. Esimerkkiprojektin avulla myös esiteltiin Clang-Tidyn tukemat määrittelytiedostot. Lisäksi selitettiin toimintaperiaate sille, miten ohjelmistokehityksen automaatioputkessa Clang-Tidya käyttäen on mahdollista estää ohjelmointikäytänteistä poikkeavan koodin pääsy versionhallinnan päähaaraan.

Käytännön toteutuksena saatiin aikaan oma moduuli toimeksiantajalle Clang-Tidyyn sekä tarkistus, jonka avulla voidaan havaita ei-julkinen luokkien periytyminen C++-koodissa. Tälle tarkistukselle kirjoitettiin automaatiotestit, jotka paljastivat virheen tarkistuksen toteutuksessa. Virhe korjattiin ja tarkistus todettiin testien päivittämisen jälkeen toimivaksi. Lopuksi uudelle tarkistukselle kirjoitettiin dokumentaatiosivu sekä lisäys julkaisutietoihin ja Clang-Tidyn tarkistusten luetteloon. Muokattu versio Clang-Tidysta otettiin onnistuneesti käyttöön toimeksiantajan projektissa. Aikaansaannos vastasi opinnäytetyölle asetettuja tavoitteita.

Työn aikana tuli selväksi, että LLVM:n dokumentaatio on monelta osin puutteellista. Clang-Tidyn laajentaminen projektin tarpeiden mukaan vahvisti käsitystä siitä, että ohjelmistojen kokoaminen lähdekoodeista ja avoimen lähdekoodin projektien muokkaaminen ovat hyödyllisiä taitoja. Testauksen paljastama virhe toimii hyvänä osoituksena ohjelmistotestaamisen tärkeydestä. Työn perusteella voidaan sanoa, että uuden moduulin lisääminen Clang-Tidyyn voi olla haastavaa puutteellisen dokumentaation takia. Uuden tarkistuksen lisääminen moduuliin on yksinkertainen prosessi, koska LLVM sisältää tähän tarkoitukseen apuohjelman. Ensimmäisen tarkistuksen toteuttaminen voi olla haastavaa, mutta koko prosessin testauksineen ja dokumentointeineen läpi käytyä uusien tarkistusten lisääminen pitäisi onnistua melko vaivattomasti tarkistusten vaikeustasoa vähitellen nostaen. Ammattilaiskäytön lisäksi Clang-Tidy sopii hyvin myös kouluprojekteihin, mutta näiden tarpeisiin Clang-Tidyn laajentaminen voi olla turhan työlästä.

KAMK • University of Applied Sciences

Abstract

Author: Kiiskilä Eetu

Title of the Publication: Extending Clang-Tidy for Company Needs

Degree Title: Bachelor of Engineering, Information and Communications Technology

Keywords: Clang-Tidy, code analysis, coding conventions, software automation, LLVM, C++

In this bachelor's thesis, an extended version of LLVM's Clang-Tidy was developed. The goal was to add at least one new check to the tool with automated tests and documentation for the check. The intent was that the mandator, Bittium Wireless Oy, could use the thesis to evaluate if developing Clang-Tidy for the company's needs would be worthwhile. The thesis could also be used as instructions for adding new checks to the tool if needed.

First, the theory of coding conventions and related tools were explored. The mandator's needs for code analysis tools were examined. Then, fundamental use of Clang-Tidy was demonstrated with a simple CMake-based C++ project. The example project was also used to showcase the use of Clang-Tidy configuration files. In addition, the principle for using Clang-Tidy in a CI/CD pipeline to block non-conforming code from reaching the main branch of a project's version control was explained.

As the practical part of the thesis, a new Clang-Tidy module for the mandator with a check for finding non-public inheritance in C++ code was implemented. Automated tests were written for the new check. With the tests, an error in the implementation of the check was discovered that was then fixed. The tests were updated to reflect the changes in the implementation and with the updated tests, the check was confirmed to work as intended. Finally, a documentation page as well as an addition to the release notes and the list of Clang-Tidy checks was written. The modified version of Clang-Tidy was successfully taken into use in the mandator's project that the new check was added for. The accomplishment was in line with the goals set for the thesis.

While working on the thesis, it became clear that the documentation of the LLVM project is in many ways imperfect. Extending Clang-Tidy for the needs of the mandator's project reinforced the idea that building software from source and modifying open-source projects for one's needs are useful skills. The error in implementation revealed by the automated testing is a good indication of the importance of software testing. Based on the work, adding a new module to Clang-Tidy can be challenging because of the problems in the documentation. Adding a new check to a module is a simple process because a helper script for that purpose is included in the LLVM project. Implementing one's first check can be difficult, but after going through the entire process with testing and documentation included, adding new checks should be possible without too much effort while gradually increasing the difficulty of the new checks. In addition to professional use, Clang-Tidy is also well suited for school projects. However, extending Clang-Tidy for use in school projects could be too much of an effort.

Sisällys

1 Johdanto 1

2 Ohjelmointikäytänteiden merkitys 3

2.1 Ohjelmointikäytänteet yleisesti 3

2.2 Tarve ohjelmointikäytänteille 3

2.3 Ohjelmointikäytänteiden ratkaisut 4

2.4 Toimeksiantajan projekti 7

3 Clang-Tidy ohjelmistokehityksen apuvälineenä 10

3.1 Valintaperusteet työkalulle 10

3.2 Käyttöönotto 12

3.3 Määrittelytiedostot 17

3.4 Clang-Tidy CI/CD-putkessa 18

4 Clang-Tidyn laajentaminen 20

4.1 Clang-Tidyn asentaminen lähdekoodeista 20

4.2 Uuden tarkistuksen lisääminen ohjelmallisesti 22

4.3 Tarkistuksen testaaminen 33

4.4 Dokumentaation päivittäminen 39

5 Lopputulos 43

5.1 Aikaansaannos 43

5.2 Oppiminen 43

5.3 Päätelmät 44

5.4 Jatkokehitys 47

6 Yhteenveto 49

Lähteet 50

Liitteet

Termit ja lyhenteet

AST	Abstract syntax tree. Abstrakti syntaksipuu.
ESSOR	European Secure Software defined Radio. Ohjelma, jossa kehitetään taktista viestintäteknologiaa Euroopan maiden yhteiskäyttöön.
LLVM	Aiemmin lyhenne sanoista Low-Level Virtual Machine. Kääntäjäinfrastruktuuriprojekti, joka koostuu modulaarisista ja uudelleenkäytettävistä kääntäjä- ja ohjelmistokokoelmateknologioista.
SDR	Software-defined radio. Ohjelmistoradio.

1 Johdanto

Tässä opinnäytetyössä kehitetään toimeksiantajalle laajennettu versio Clang-Tidysta automatisoitua koodianalyysiä varten. Tavoitteena on lisätä työkaluun vähintään yksi toimeksiantajan tarpeita vastaava tarkistus ja kirjoittaa tälle automaatiotestit ja dokumentaatio. Työn avulla toimeksiantaja voi arvioida, onko Clang-Tidyn kehittäminen uusilla tarkistuksilla järkevää, ja tarvittaessa työ myös toimii ohjeena uusien tarkistusten lisäämiselle.

Koodianalyysin avulla on mahdollista löytää virheitä ohjelmistosta varhaisessa vaiheessa. Myös ohjelmointityyliä on mahdollista valvoa, mikä voi auttaa pitämään koodin yhteneväisenä ja mahdollisimman helposti luettavana. Koodianalyysin tarkoitus on siis auttaa tuottamaan toimivia ohjelmia ja helpottamaan ohjelmoijien työtä.

Työn toimeksiantaja on Bittium Wireless Oy. Toimeksiantajan projekti liittyy taktiseen tiedonsiirtoon, joten laadukkaan koodin tuottaminen on erityisen tärkeää. Työssä keskitytään tietyn projektin ohjelmointikäytänteisiin, mutta kehitettävää koodianalyysityökalua on mahdollista käyttää muissakin projekteissa.

Vapaasti käytettävänä avoimen lähdekoodin ohjelmana Clang-Tidy sopii hyvin kehitettäväksi opinnäytetyönä yritykselle. Työkalun kehittäminen on mahdollista pitää irrallaan muista toimeksiantajan projekteista, jolloin kehitysprosessi on helppo kuvata opinnäytetyössä ilman tarvetta salassa pidettävien tietojen esittämiselle julkisesti. Yrityskäytön lisäksi Clang-Tidy sopii lisenssiltään hyvin myös esimerkiksi opiskelijaprojekteihin.

Työ alkaa teoriakatsauksella ohjelmointikäytänteisiin. Erityyppisiä koodianalyysityökaluja käydään läpi esimerkkien avulla. Tarkemmin perehdytään Clang-Tidyn käyttöön. Ohjelman käyttöönottoon opastetaan esimerkkiprojektin avulla, minkä lisäksi työkalua laajennetaan suunnitelman mukaisesti. Lopuksi tarkastellaan aikaansaannosta ja pohditaan työn merkitystä eri näkökulmista.

Lukijan on hyvä tietää, että opinnäytetyö sisältää kuvia, jotka on luotu kuvankäsittelyn avulla. Nämä kuvat esittävät komentosarjoja. Laaja tuloste on poistettu kuvaa muokkaamalla, jotta kuva mahtuisi opinnäytetyöhön. Kuvat on siis otettu todellisessa tilanteessa, mutta tuloste on rajattu tai poistettu esitysteknisistä syistä, kun tämä ei estä ymmärtämästä käsiteltävää asiaa. Lukija pystyy kuvien ja muun sisällön avulla toistamaan opinnäytetyön tulokset.

Opinnäytetyössä esitetään LLVM:ään ja Clang-Tidyyn liittyviä tiedostoja kuvina. Lisenssin vaatimusten mukaisesti tehdyt lisäykset ja muutokset käyvät ilmi tiedostoihin lisätyistä kommenteista, koodieditorin versionhallintalisäosan tilanseurannan ilmaisimista, opinnäytetyön sanallisesta kuvauksesta tai liitteistä.

2 Ohjelmointikäytänteiden merkitys

Ennen koodianalyysityökalun käyttöönottoa ja laajentamista on tarpeen tietää, mitä ohjelmointikäytänteillä tarkoitetaan. Olemassa olevien automaatioratkaisujen tunteminen auttaa kartoittamaan tarpeen uusien tarkistusten toteuttamiselle itse.

Koodianalyysityökalun tulee sopia käyttötarkoitukseensa. Työkalua valitessa täytyy tuntea projektin ohjelmointikäytänteet ja tunnistaa projektin tarpeet.

Tutustutaan ensin ohjelmointikäytänteisiin yleisesti ja käydään läpi olemassa olevia ratkaisuja koodianalyysin automatisointiin. Tarkastellaan myös toimeksiantajan projektia ohjelmointikäytänteitä ajatellen.

2.1 Ohjelmointikäytänteet yleisesti

Ohjelmointikäytänteet ovat sääntöjä, ohjeita ja hyviä tapoja koodin kirjoittamiseen. Ohjelmointikäytänteiden tarkoitus on auttaa kirjoittamaan turvallista, luotettavaa ja monialustaista koodia. Lisäksi ohjelmointikäytänteillä pyritään helpottamaan koodin testaamista ja ylläpitämistä. [1, viitattu 2.]

Ohjelmointikäytänteet eivät aina ole yleispäteviä. Eri ohjelmointikielille voi olla omia käytänteitään, ja samaa kieltä käytettäessäkin ohjelman käyttökohde voi vaikuttaa käytänteisiin. [1, viitattu 3.] Esimerkiksi turvallisuudeltaan ja suorituskyvyltään kriittisten reaaliaikajärjestelmien käytänteet voivat olla turhan rajoittavia muissa käyttötarkoituksissa [1, viitattu 4].

2.2 Tarve ohjelmointikäytänteille

Ohjelmointikäytänteitä tarvitaan, kun halutaan kirjoittaa laadukasta koodia. Ohjelmointikäytänteiden avulla pyritään säästämään ohjelmistokehityksen kustannuksissa ja saamaan tuote markkinoille nopeammin. [1, viitattu 2.]

Jotkin ohjelmointikielet ovat joustavia ja tehokkaita, mutta samalla riskialttiita. Väärin kirjoitettu koodi voi toimia ennalta arvaamattomasti ja johtaa haavoittuvuuksiin tai vikatilanteisiin. Näitä ongelmia on mahdollista välttää sopivien ohjelmointikäytänteiden avulla. [1, viitattu 2.]

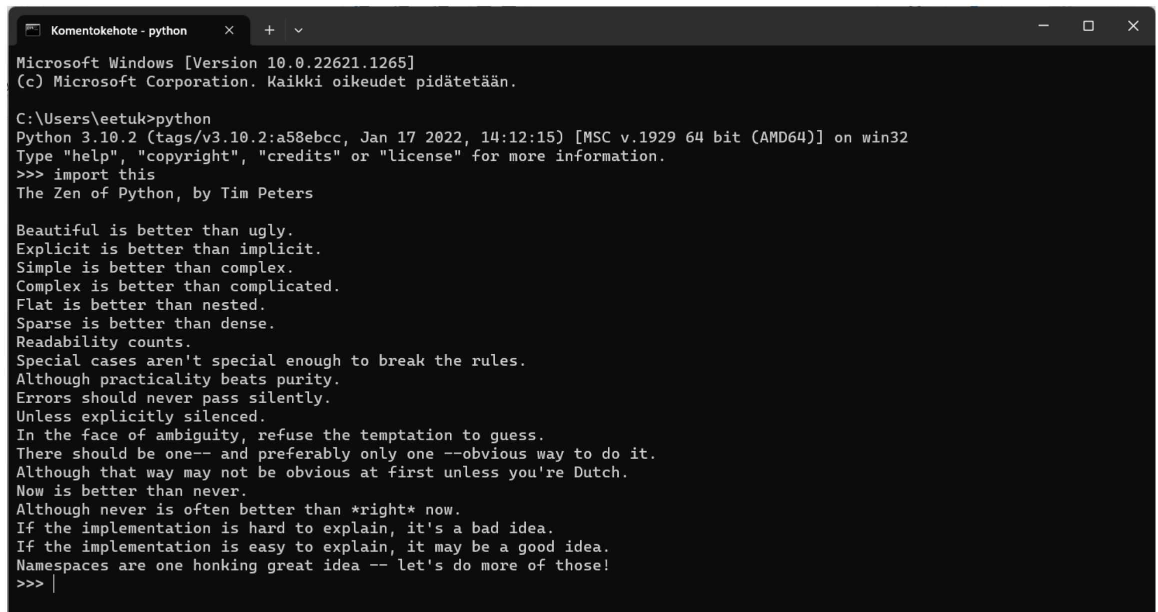
Ohjelmointikäytänteisiin voidaan lukea myös yhteisesti sovitun tiedostokoodauksen käyttäminen. Kaikista yksinkertaisimmatkaan tietokoneohjelmat eivät välttämättä toimi eri käyttöjärjestelmissä, jos tiedostokoodaus ei ole sopiva [1, viitattu 5]. Potentiaalisten käyttäjien vähenemisen lisäksi tämä voi vaikeuttaa testausta, jos halutaan esimerkiksi testata Windowsissa kirjoitettua koodia Linux-järjestelmässä [1].

Ohjelmointikäytänteillä on merkitystä myös ohjelmoijien työskentelyä helpottavana asiana, koska koodia luetaan enemmän kuin kirjoitetaan. Ohjelmointikäytänteiden on tarkoitus tehdä koodista yhteneväistä ja helpommin luettavaa. [1, viitattu 6.]

2.3 Ohjelmointikäytänteiden ratkaisut

Ohjelmointikäytänteitä varten on olemassa erilaisia standardeja ja ohjesääntöjä. Esimerkiksi C++:aan liittyen suosittuja sääntökokoelmia ovat C++ Core Guidelines ja JSF AV C++ [2; 4]. Ohjelmointikäytänteitä laativat esimerkiksi avoimen lähdekoodin yhteisöt ja ohjelmointikielten kehittäjät [7] sekä yritykset ja alaa tuntevat kirjailijat [4]. Myös projektin sisäisiä käytänteitä voidaan laatia, mutta tämä ei välttämättä ole tarpeen, jos olemassa olevat käytänteet sopivat projektiin [4].

Erityisesti Pythonin kohdalla korostetaan ohjelmointikäytänteiden merkitystä. Jopa kieleen itsessään on lisätty ohjeita hyvän koodin kirjoittamiseen. Nämä The Zen of Python -nimellä tunnetut ohjeet saadaan näkyviin kirjoittamalla Python-tulkkiin *import this*. Tämä näkyy kuvassa 1. [1, viitattu 8.]



Kuva 1. The Zen of Python komentoriviympäristössä [1].

Ohjeiden ja sääntöjen lisäksi on olemassa työkaluja, joilla voidaan määrittää projektin ohjelmointikäytänteet ja valvoa näiden noudattamista. Nämä eroavat toisistaan ominaisuuksiltaan ja käyttökohteiltaan. [1.]

EditorConfig on projekti, joka muodostuu tiedostojen tyyliä kuvaavasta tiedostomuodosta ja tätä varten kehitetyistä editorien lisäosista. EditorConfigin määritelmä on hyvin lyhyt, eikä se sisällä paljoa erilaisia asetuksia. EditorConfig on kattavasti eri editorien tukema ja on valmiiksi asennettuna moniin editoreihin. Erityisen hyödyllisen EditorConfigista tekee sen käytön helppous ja se, että työkalun avulla on mahdollista määritellä käytettävä tiedostokoodaus. EditorConfig sopiikin hyvin tiedostojen teknisen esitystavan määrittämiseen. [1, viitattu 5; 9; 10; 11; 12.]

Clang-Tidy on Clangiin pohjautuva lintteri [13], josta löytyy tarkistuksia C++-, C- ja Objective-C-kielille [14]. Ohjelman avulla on mahdollista etsiä yleisiä ohjelmointivirheitä koodista. Lisäksi määrättyyn tyyliin liittyviä poikkeuksia voidaan hakea [14]. Clang-Tidy toimii itsenäisesti komentorivin kautta sekä osana eräitä ohjelmointiympäristöjä ja editoreja [15].

Clang Static Analyzer on staattinen koodianalysaattori samalta kehittäjäyhteisöltä kuin Clang-Tidy [16]. Myös Clang Static Analyzer sisältää tuen C++-, C- ja Objective-C-kielille [17]. Clang Static Analyzerin tarkistuksia on mahdollista suorittaa Clang-Tidyn kautta [13; 14].

ClangFormat on koodin tyyliin liittyvä työkalu. Ohjelman avulla voidaan tarkistaa, että koodin tyyli on määrittelyn mukainen. Koodia voidaan myös automaattisesti muotoilla työkalun avulla halutunlaiseksi. Myös ClangFormat on saman yhteisön kehittämä kuin edellä mainitut Clang-pohjaiset työkalut. Ohjelma tukee useita kieliä, jotka muistuttavat syntaksiltaan C:tä: C, C++, Java, JavaScript, JSON, Objective-C, Protobuf ja C#. Myös ClangFormat toimii komentorivin kautta tai eräisiin ohjelmointiympäristöihin ja editoreihin integroituna. [18.]

Edellä lueteltujen työkalujen lisäksi myös ohjelmointikielten kääntäjät osaavat tunnistaa erilaisia koodiin liittyviä puutteita. Esimerkiksi GCC [19] ja Clang [20] sisältävät paljon mahdollisia tarkistuksia. Nämä kuitenkin eroavat toisistaan, joten osa asetuksista toimii vain tiettyä kääntäjää käytettäessä. Osa tarkistuksista voi myös olla eri tavoin nimetty eri kääntäjissä, jolloin näiden käyttö on taas kääntäjästä riippuvaista.

Jo mainitut työkalut pohjautuvat avoimeen lähdekoodiin ja ovat ilmaisia käyttää. On olemassa paljon muitakin vastaavanlaisia työkaluja, jotka voivat olla yleiskäyttöisiä tai keskittyä tiettyihin ohjelmointikieliin. Myös kaupallisia ohjelmia on olemassa. Esimerkiksi GitHubista löytyy projekteja, jotka listaavat lisää työkaluja [21; 22]. Ominaisuuksien lisäksi kaupalliset ohjelmat poikkeavat toisistaan lisenssiehdoiltaan ja hinnoittelultaan eikä näiden laajentaminen omilla tarkistuksilla ole välttämättä mahdollista, joten kaupallisia ohjelmia ei esitellä tarkemmin opinnäytetyössä.

Esitellyt työkalut ovat käyttötarkoitukseltaan erilaisia. EditorConfig on yleiskäyttöinen työkalu, jolla voidaan määrittää projektissa käytettäviä editorien asetuksia yhden tiedoston avulla. EditorConfig sopii tiedostoihin liittyvien yleisten asetusten määrittämiseen, mutta ei koodin tarkistamiseen. Lintterit, kuten Clang-Tidy, edustavat staattisten koodianalysaattorien yksinkertaisempaa päätä. Lintterit auttavat tunnistamaan yleisiä ohjelmointivirheitä ja koodin tyyliin liittyviä poikkeamia. [23; 24.] Clang Static Analyzer taas on edistyneempi staattinen koodianalysaattori. Edistyneemmät analysaattorit ymmärtävät ohjelman kulkua paremmin kuin yksinkertaisemmat lintterit. Näiden avulla voidaan tunnistaa ongelmia, jotka vaativat tarkempaa analyysiä kuin lintterit. [23; 24.] ClangFormat kuuluu formattereihin eli koodin asetteluun liittyviin työkaluihin. Näiden avulla voidaan tarkkailla koodin tyyliä ja korjata koodin muotoiluun liittyviä poikkeamia. Kääntäjien diagnostiikka taas keskittyy ohjelmointivirheiden etsimiseen, eikä niinkään koodin tyyliin liittyviin ongelmiin. Kääntäjien vianetsintä pyritään toteuttamaan siten, että se olisi mahdollisimman tehokasta ja vääriä hälytyksiä tulisi mahdollisimman vähän. [24; 25.]

Ohjelmointikäytänteitä varten on siis olemassa monenlaisia työkaluja. Erityyppisiä työkaluja käyttämällä voidaan saada kattavasti varmistettua projektin käytänteiden toteutuminen. Vaikka automaattisia työkaluja onkin monenlaisia, on aina olemassa tarvetta myös ihmisten tekemälle katselmoinnille. Joidenkin sääntöjen tarkistaminen voisi olla hyvin vaikeaa automaattisten työkalujen avulla, kun taas ihminen saattaa kyetä tekemään nämä tarkistukset helposti. Ihmisen arvostelukykyä tarvitaan esimerkiksi arvioitaessa, onko muuttuja nimetty hyvin. Koska manuaaliselle koodin tarkastelulle on tarvetta, myös tähän tarkoitukseen on kehitetty työkaluja [26]. Ihmismäistä ajattelua tarvitaan myös koodianalysaattoreiden tuottamien väärien hälytysten vuoksi. Nämä väärät hälytykset voidaan ainakin osassa työkaluista kytkeä pois päältä erityisillä NOLINT-kommenteilla [13], mutta tämä edellyttää ihmisen päätöksentekoa.

Vaikka ohjelmointikäytänteet voivat auttaa kirjoittamaan laadukasta koodia, eivät ne kuitenkaan ole tae hyvästä koodista. Ohjelmointikäytänteitä seuratessakin ihmisellä tulee olla kyky tuottaa toimivia algoritmeja. Ohjelmointikäytänteet ovat ihmisten kirjoittamia, joten myös käytänteet itse voivat olla puutteellisia tai jopa virheellisiä. Joihinkin ohjelmointikäytänteisiin kuuluu sääntö, jonka mukaan yhden operaattorin ollessa ylikuormitettu pitää myös tälle käänteisenä pidettävän operaattorin olla ylikuormitettu [27]. Tämän säännön noudattaminen ei kuitenkaan aina ole järkevää, koska esimerkiksi kahden matriisin kertolasku on, mutta kahden matriisin jakolaskua ei ole määritelty. Ohjelmointikäytänteisiin tuleekin osata suhtautua oikealla tavalla. The Zen of Pythonin mukaan poikkeustapaukset eivät ole riittävän poikkeuksellisia oikeuttamaan sääntöjen rikkomista, mutta tarkoituksenmukaisuus menee puhdasoppisuuden edelle [8]. Käytännössä tämän voidaan ajatella tarkoittavan sitä, että ohjelmointikäytänteisiin tulee suhtautua sääntöinä, joita pitää noudattaa. Säännöistä voidaan kuitenkin poiketa, jos tämä parantaa koodin laatua.

2.4 Toimeksiantajan projekti

Työn toimeksiantaja Bittium Wireless Oy on mukana taktiseen tiedonsiirtoon liittyvässä ESSOR-ohjelmassa. Ohjelman tavoitteena on kehittää turvalliseen viestintään liittyvää teknologiaa Euroopan maiden asevoimien käyttöön. Yhteinen kommunikointitapa on tarpeen esimerkiksi rauhanturvatehtävien suunnittelussa ja toteutuksessa. Ohjelmistoperustaisen SDR-teknologian avulla on mahdollista saavuttaa yhteensopivuus eri valtioiden radiolaitteiden välillä. [28.]

Toimeksiantajan toteuttamien ohjelmistojen tulee toimia vaativissakin olosuhteissa, joten on tärkeää havaita mahdolliset ongelmat varhaisessa vaiheessa. Automaatiotyökalujen käyttö auttaa

tässä tavoitteessa. Sopivien työkalujen valitseminen mahdollistaa kattavan koodianalyysin mahdollisimman vähäisellä manuaalisella työllä. Tämän takia työkalujen tulee sisältää jo valmiiksi mahdollisimman paljon tarkistuksia, joita voidaan valita käyttöön projektin tarpeiden mukaan. Koska mikään työkalu ei todennäköisesti sisällä valmiiksi kaikkia projektiin tarvittavia tarkistuksia, on tärkeää voida laajentaa työkalua omilla tarkistuksilla. Myös useiden eri työkalujen yhteiskäyttö on kannattavaa, koska esimerkiksi lintterit, edistyneemmät staattiset koodianalysaattorit ja formatterit sopivat toisistaan erilaisiin käyttötarkoituksiin.

ESSOR-ohjelmaa kehitetään yhteistyökumppaneiden kanssa. Toimeksiantajalla on käynnissä ohjelmaan liittyvä projekti. Koska ohjelmassa on käytössä yhteiset ohjelmointikäytänteet, on tärkeää, että kaikilla ohjelmassa mukana olevilla tahoilla on mahdollisuus saada samat koodianalyysityökalut käyttöönsä. Tällöin myös työkaluihin tehdyt muutokset tulee voida jakaa yhteistyökumppaneiden kanssa.

Opinnäytetyöprosessin alussa on tehty tarkempi tehtävän määrittely projektin tarpeiden mukaan. Projektin ohjelmointikäytänteet on käyty läpi ja tehty vertailua siitä, mitkä projektin säännöistä on mahdollista tarkistaa jo olemassa olevilla työkaluilla, joihin myös Clang-Tidy ja tämän kautta suoritettava Clang Static Analyzer kuuluvat. Lisäksi ClangFormat on otettu mukaan tarkasteluun. Projektin ohjelmointikäytänteissä listataan yhteensä 182 yleistä sekä C:hen tai C++:aan liittyvää sääntöä tai suositusta [29]. Koska osa näistä sisältää useita alakohtia, voidaan todellisen määrän ajatella olevan vielä suurempi.

Projektin käytänteistä 76 on todettu olevan mahdollista tarkistaa tyydyttävällä tavalla olemassa olevilla työkaluilla. Osa näistä voi vaatia myös jonkin verran manuaalista tarkkailua. Eräs projektin suosituksista sanoo, että esittelyjen tulee olla pienimmällä mahdollisella näkyvyysalueella kuin on järkevää [29, luku 11.2.5 DECLARATIONS]. On esimerkiksi mahdollista tarkistaa automaattisesti, voisiko muuttujasta tehdä paikallisen jollekin funktiolle. Ohjelmoijan on kuitenkin tarpeen tehdä päätös siitä, onko funktion sisäisen muuttujan esittely järkevää funktion alussa vai jonkin funktiosta löytyvän ohjelmalohkon sisässä. Järkevä tapa voi olla tilanteesta riippuvainen.

Käytänteistä 41 on todettu sellaiseksi, että näiden tarkistaminen vaatii manuaalista työtä tai tarkistuksia ei ole järkevää toteuttaa Clang-pohjaisiin työkaluihin. Projektin säännöissä esimerkiksi sanotaan, että kolmannen osapuolen kirjastojen tulee olla lisensseiltään sopivia projektiin [29, luku 11.2.18 INTELLECTUAL PROPERTY LIBRARIES USABILITY]. Tämän säännön noudattaminen edellyttää aina ihmisen tietoista päätöstä. Osalle käytänteistä olisi mahdollista toteuttaa tarkistus, mutta tämä olisi työlästä eikä saavutettava hyöty olisi riittävä työhön nähden. Esimerkiksi

kommenttien tulee sääntöjen mukaan olla englanniksi [29, luku 11.1 GENERAL CODING RULES FOR ANY LANGUAGE]. Tämän säännön tarkistaminen vaatisi luonnollisten kielten tunnistamista. Ongelmia voisi aiheutua esimerkiksi matemaattisista kaavoista tai muunlaisista kommenteista, joiden sisältö ei edusta mitään luonnollista kieltä. Tarkistus olisi työläs toteuttaa ja altis väärille hälytyksille. Kommenttien kieli on helppo tarkistaa koodin katselmoinnissa, joten tarkistusta ei ole järkevä toteuttaa. Projektin käytänteissä myös listataan sallitut merkit, joita kooditiedostoissa saa käyttää [29, luku 11.2.1 GENERAL ENVIRONMENT]. Tämän tarkistuksen voi toteuttaa helposti skriptinä, jossa hyödynnetään säännöllisiä lausekkeita. Lintterin laajentaminen olisi todennäköisesti työläämpää.

Lopuille projektin käytänteille olisi mahdollista ja järkevää tehdä tarkistuksia. Näistä 50 on arvioitu olevan mahdollista tarkistaa tyydyttävällä tavalla uusia tarkistuksia lisäämällä joko Clang-Tidyyn, Clang Static Analyzeriin tai ClangFormatiin. Osa kuitenkin vaatii myös jonkin verran ihmisten tekemää tarkkailua. Sääntöjen mukaan esimerkiksi break-lauseen käyttämisen switch-lohkon ulkopuolella tulee olla rajattua. Käyttö tulee perustella järkevästi kommentilla. [29, luku 11.2.9.1 Flow control.] On mahdollista kirjoittaa tarkistus, joka rajoittaa break-lauseen käyttämistä ilman kommenttia. Ihmisen on kuitenkin tehtävä päätös siitä, onko kommentissa esitetty perustelu järkevä vai ei. Jäljellä olevat 15 käytännettä on arvioitu jatkuvaa parantelua vaativiksi. Näille on mahdollista kirjoittaa tarkistuksia, mutta uusia tarkistuksia tulee todennäköisesti kirjoittaa lisää tarpeen vaatiessa. Projektin sääntöjen mukaan kaikilla ei-tyhjillä lauseilla tulee olla vaikutuksia ohjelman toimintaan [29, luku 11.2.9.1 Flow control]. Sääntö on laaja, ja uusia tarkistuksia voi lisätä jatkuvasti lintteriä kehitettäessä. Liitteessä 1 on joitain esimerkkejä hyödyttömistä lauseista.

Toimeksiantajan projektia varten Clang-Tidya voisi laajentaa monenlaisilla tarkistuksilla. Tehtävää olisi paljon, mutta opinnäytetyön aikana ei ole tarkoitus toteuttaa mahdollisimman paljon uusia tarkistuksia. Tavoitteena on selvittää ja dokumentoida käytännön esimerkin avulla yhden uuden tarkistuksen lisäämällä, miten Clang-Tidya voi laajentaa, sekä käytettävissä olevan ajan puitteissa mahdollisesti toteuttaa vielä lisää uusia tarkistuksia. Tehtävän tarkemman määrittelyn yhteydessä on laadittu taulukkomuotoinen asiakirja, josta on luettavissa projektin käytänteitä vastaavat tarkistukset eri työkaluissa. Asiakirjaa hyödyntämällä on mahdollista luoda projektiin sopivat määrittelytiedostot eri työkaluille. Taulukkoa voi käyttää tehtävälistana esimerkiksi Clang-Tidya laajennettaessa. Koska asiakirjasta myös ilmenee, mitkä käytänteet ihmisen on varmistettava, asiakirjan perusteella voisi myös laatia muistilistan koodin katselmointia varten.

3 Clang-Tidy ohjelmistokehityksen apuvälineenä

Kun tunnetaan ohjelmointikäytänteisiin liittyvää teoriaa ja olemassa olevia työkaluja ja on kartoitettu projektin tarpeet asiaan liittyen, voidaan tarkastella hieman perusteellisemmin projektissa käytettävää koodianalyysityökalua. Toimeksiantajan projektiin on valittu automaatiotyökaluksi Clang-Tidy.

Käydään aluksi läpi perusteet Clang-Tidyn valinnalle. Tutustutaan sen jälkeen ohjelman käyttöön yksinkertaisen esimerkkiprojektin avulla. Tarkastellaan myös työkalun tarkempaa määrittelemistä konfiguraatiotiedostolla ja käyttöä CI/CD-putkessa.

3.1 Valintaperusteet työkalulle

Vapaasti käytettävänä avoimen lähdekoodin ohjelmana Clang-Tidyn käyttö on ilmaista. Ohjelmistoprojekteihin liittyy aina riski, että kehittäjä hylkää projektin. Kaupallisissa, suljetun lähdekoodin ohjelmissa tämä voi aiheuttaa ohjelmistoa käyttävälle taholle ongelmia tukipalveluiden ja virheenkorjausten loppuessa. Avoimen lähdekoodin projekteissa tilalle voi löytyä uusi kehittäjä, minkä lisäksi ohjelman käyttäjä voi itse muokata ohjelmistoa tarpeidensa mukaan. [30, s. 11–12.]

Vaikka avoimen lähdekoodin ohjelmiston käyttö periaatteessa olisikin ilmaista, yritykselle voi kuitenkin koitua kustannuksia ohjelmiston käytöstä. Yritys voi joutua kouluttamaan henkilöstöä ohjelmiston käyttöön. [30, s. 11.] Vastaavasti kaupallisissa palveluissa ohjelmiston käyttöön perehdyttäminen voi sisältyä ohjelmiston hintaan. Jos avoimen lähdekoodin projektissa ilmenee ongelmia, näiden korjaaminen voi aiheuttaa yritykselle kustannuksia. Kaupallisten sovellusten kohdalla voidaan taas vaatia ohjelmiston toimittajaa korjaamaan ongelmat. [30, s. 11.]

Avoimen lähdekoodin projekteja pidetään yleisesti luotettavina [30, s. 11]. Jos käyttäjällä on pääsy ohjelmiston lähdekoodiin, tämä voi tarvittaessa käydä koodia läpi ja varmistua ohjelman turvallisuudesta. Koodin lukeminen voi toki olla aikaa vievää ja aiheuttaa taas tällä tavoin kustannuksia yritykselle.

Avoin lähdekoodi mahdollistaa sen, että käyttäjäyhteisön ollessa laaja voidaan odottaa myös vikojen korjaamisen olevan nopeaa [30, s. 12]. Myös uusia ominaisuuksia voidaan kehittää nopeaan tahtiin. Haittapuolena on se, että avoimen lähdekoodin projekteihin voidaan ujuttaa haitallista

koodia tahallisen vahingollisessa tarkoituksessa. Myös haavoittuvuuksia voidaan pyrkiä hyödyntämään haitallisessa tarkoituksessa ennen niiden korjaamista. Näitä riskejä liittyy toki myös suljetun lähdekoodin projekteihin, joissa ainoastaan kehittäjällä on mahdollisuus päästä käsiksi lähdekoodiin. [30, s. 12.]

Vapaasti käytettäviin avoimen lähdekoodin projekteihin liittyy siis sekä hyviä että huonoja puolia, joita joudutaan arvioimaan. Clang-Tidy on osa LLVM-projektia, joka projektin GitHub-sivun perusteella on suosittu ja aktiivisesti kehitetty [31]. LLVM-projektin käyttäjien ja kehittäjien joukossa on hyvin tunnettuja tahoja, kuten Apple [32; 33; 34; 35]. Laajan käyttäjä- ja kehittäjäjoukon perusteella vaikuttaa siltä, että Clang-Tidyn tapauksessa riski projektin hylkäämiselle pian on melko pieni.

Sen lisäksi, että ohjelmisto on ilmainen ja lähdekoodi avoin, lisenssin tulee sopia käyttäjän tarpeisiin. LLVM-projekti on julkaistu muunnellulla Apache 2.0 -lisenssillä [36]. Apache 2.0 on yleisesti ottaen kaupalliseen tarkoitukseen sopiva lisenssi, joka sallii ohjelmiston jakelun, muuttamisen ja yksityisen käytön. Lisenssi täytyy pitää lisensoidun materiaalin mukana ja muutokset lisensoituun materiaaliin pitää dokumentoida. Lisenssi sisältää vastuuvapautuslausekkeen, eikä sisällä takuuta ohjelmiston puolesta. Lisenssi myös sisältää huomautuksen, jonka mukaan lisenssi ei oikeuta käyttämään lisenssiantajan tavaramerkkejä, paitsi hyväksyttävistä syistä esimerkiksi alkuperäisen lähteen mainitsemisen yhteydessä. Lisenssiin sisältyy lisäksi patenttilisenssisopimus, jolla pyritään estämään se, että joku ohjelman kehittäjistä yrittäisi syyttää toista kehittäjää patenttirikkomuksesta lisensoidun materiaalin käyttöön liittyen. [37.] LLVM:n muunneltu lisenssi vapauttaa käyttäjän tietyistä lisenssissä mainituista velvollisuuksista, kun käyttäjä jakelee tietokoneen suoritettavaan muotoon käännettyä ohjelmaa, johon on sisällyttänyt osia lisenssin alaisesta ohjelmistosta. Lisäksi muunnellussa lisenssissä kuvataan, miten lisenssi toimii yhdessä muiden lisenssien kanssa. Myös vanha LLVM:n käyttämä lisenssi, joka on eri kuin Apache 2.0, on liitteenä muunnellussa lisenssissä. [36.]

Edellä mainittujen seikkojen lisäksi ohjelman tulee olla käyttötarkoitukseen sopiva. Clang-Tidy on esimerkiksi yleiskäyttöisempi kuin Clazy, joka myös on Clang-pohjainen, mutta erityisesti Qt-kirjastoa varten kehitetty työkalu [22; 38]. Cppcheck taas sisältää hyvän määrän tarkistuksia [39], mutta avoimen lähdekoodin versio on GPLv3.0-lisensoitu [40]. Tämä lisenssi voi olla turhan rajoittava, koska muunneltu ohjelma saadaan julkaista kokonaisuudessaan pelkästään samalla lisenssillä. Tehdyt muutokset voidaan julkaista myös vapaammalla lisenssillä, mutta ohjelmakokonaisuuden tulee olla edelleen alkuperäisen lisenssin alainen. [41.] cpplint on Googlen kehittämä työkalu, joka sisältää tarkistuksia Googlen ohjelmointitapojen noudattamiseen [22; 42]. Ohjelmaa ei

näytä kehitettävän kovinkaan aktiivisesti [43], mutta projektista on forkattu hieman aktiivisemmin kehitettävä versio [21; 44]. Myös muita vaihtoehtoja läpi käydessä vaikuttaa siltä, että Clang-Tidy on hyvä valinta: Työkalu sisältää paljon valmiita tarkistuksia [14], minkä lisäksi ohjelman laajentaminen on mahdollista ja asiasta on olemassa dokumentaatiota [24]. Kuten edellä on jo kuvattu, Clang-Tidy on myös lisenssiltään sopiva ja työkalua kehitetään aktiivisesti.

3.2 Käyttöönotto

Käytettäessä Linuxia yksinkertainen tapa asentaa Clang-Tidy on järjestelmän paketinhallinta. Kuvassa 2 näkyy, kuinka Clang-Tidy asennetaan Ubuntussa.

```
essor3d@essor3d-VirtualBox:~$ sudo apt update
essor3d@essor3d-VirtualBox:~$ apt search clang-tidy
essor3d@essor3d-VirtualBox:~$ sudo apt install clang-tidy
```

Kuva 2. Clang-Tidyn asentaminen Ubuntun paketinhallinnalla.

Clang-Tidy voidaan ajaa komentoriviohjelmalla clang-tidy. Dokumentaatiossa mainitaan, että tarkistettavia tiedostoja voidaan valikoida säännöllisten lausekkeiden avulla. Dokumentaatio ei kuitenkaan kerro, minkä tyyppisistä säännöllisistä lausekkeista on kyse. [13.] Muilla käyttäjillä on ollut vastaavanlaisia ongelmia ClangFormatin kanssa. Ohjelman lähdekoodeja tutkimalla on saatu selvitettyä, että ClangFormat käyttää POSIX ERE -tyylisiä säännöllisiä lausekkeita. [45.] Tämän tiedon avulla saadaan vahvistus sille, että LLVM käyttää yleisesti kyseistä regex-tyyppiä, joten myös Clang-Tidyn pitäisi toimia samalla syntaksilla [46].

Clang-Tidyn ajamiseen on pari muutakin tapaa. Voidaan käyttää komentoriviohjelmaa run-clang-tidy [47], joka ainakin Ubuntussa tulee Clang-Tidyn mukana paketinhallinnan kautta asennettaessa. Tämä komento mahdollistaa Clang-Tidyn ajamisen useina rinnakkaisina prosesseina [47; 48]. Ohjelma käyttää säännöllisiin lausekkeisiin Pythonin re-kirjastoa [48]. Ohjelman clang-tidy-diff avulla taas voidaan ajaa Clang-Tidyn tarkistukset tiedostoihin tehdyille muutoksille [49].

Käytetään Clang-Tidyn ajamiseen ohjelmaa run-clang-tidy. Luodaan ensin yksinkertainen C++-projekti. Projektin kansiorakenne näkyy kuvassa 3.

```
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy$ tree
.
├── CMakeLists.txt
├── Include
│   └── Utils.h
└── Source
    ├── main.cpp
    └── Utils.cpp

2 directories, 4 files
```

Kuva 3. Esimerkkiprojektin kansiorakenne.

Esimerkkiprojekti on CMake-pohjainen C++-projekti. Projekti koostuu *CMakeLists.txt*-tiedostosta ja kolmesta kooditiedostosta. Näiden tiedostojen sisältö näkyy kuvissa 4, 5, 6 ja 7.

```
CMakeLists.txt
1   cmake_minimum_required(VERSION 3.22)
2
3   project(Hello-World)
4
5   set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
6
7   add_executable(main
8                   Source/main.cpp
9                   Source/Utils.cpp)
10
11  target_include_directories(main PRIVATE .)
12
```

Kuva 4. Esimerkkiprojektin *CMakeLists.txt*-tiedosto.

```
Source > main.cpp > ...
 1  #include <iostream>
 2
 3  #include "Include/Utils.h"
 4
 5  int main()
 6  {
 7      int uninitialized;
 8      std::cout << uninitialized << "\n";
 9
10      int zero = getIntZero();
11      std::cout << zero << "\n";
12
13      return 0;
14  }
15
```

Kuva 5. Esimerkkiprojektin *main.cpp*-tiedosto.

```
Include > C Utils.h > ...
 1  int getIntZero();
 2
```

Kuva 6. Esimerkkiprojektin *Utils.h*-tiedosto.

```
Source > Utils.cpp > ...
 1  #include "Include/Utils.h"
 2
 3  int getIntZero()
 4  {
 5      int* result = new int(0);
 6      return *result;
 7  }
 8
```

Kuva 7. Esimerkkiprojektin *Utils.cpp*-tiedosto.

CMakeLists.txt on sisällöltään tyypillinen yksinkertaiselle C++-projektille. Tiedosto sisältää pelkästään esimerkkiä varten tarvittavat asiat. Tiedostossa olisi hyvä esimerkiksi asettaa käytettävä C++-standardi, mutta tämä ei ole esimerkin kannalta välttämätöntä. *CMAKE_EXPORT_COMPILE_COMMANDS ON* tarvitaan, jotta CMake luo *compile_commands.json*-tiedoston. Muuttujan

voisi myös asettaa komentorivin kautta. Clang-Tidy käyttää tätä JSON-tiedostoa projektia käsitellessään. [13.]

Projektin päätiedosto sisältää muuttujan *uninitialized*, joka nimensä mukaisesti on alustamaton. Tämän muuttujan arvo tulostetaan, jolloin arvo voi käytännössä olla mitä vain tietotyypille ominaista. Tämä on ongelma, joka halutaan havaita Clang-Tidyn avulla. Toinen ongelma on *Utils.cpp*-tiedoston funktiossa oleva muistinvaraus. Varattua muistia ei vapauteta ennen funktiosta poistumista. Myös tämä ongelma halutaan havaita Clang-Tidylla.

Projekti tulee generoida CMakella, jolloin *compile_commands.json*-tiedosto luodaan. Tämä tapahtuu kuvan 8 mukaisilla komennoilla, kun ollaan projektin juurihakemistossa.

```
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy$ mkdir Build
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy$ cd Build/
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy/Build$ cmake -S .. -B .
```

Kuva 8. Esimerkkiprojektin generoiminen CMakella.

Nyt voidaan käyttää Clang-Tidya projektin tarkistamiseen. Monista komentoriviohjelmista tuttuun tapaan aiemmin mainitut Clang-Tidyn ajamiseen sopivat komennot tulostavat ohjeen työkalun käyttöön argumentilla *-h* tai *--help*. Tämän ohjeen avulla saadaan muodostettua projektiin sopiva komento, jonka suoritus näkyy kuvissa 9 ja 10.

```
essor3d@essor3d-VirtualBox: ~/code/thesis-clang-tidy
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy$ run-clang-tidy -checks="-*,cppcoreguidelines-init-variables,clang-analyzer-cplusplus.NewDeleteLeaks" -p "$(pwd)/Build" "$(pwd)"
Enabled checks:
    clang-analyzer-core.CallAndMessage
    clang-analyzer-core.CallAndMessageModeling
    clang-analyzer-core.DivideZero
    clang-analyzer-core.DynamicTypePropagation
    clang-analyzer-core.NonNullParamChecker
    clang-analyzer-core.NonnilStringConstants
    clang-analyzer-core.NullDereference
    clang-analyzer-core.StackAddrEscapeBase
    clang-analyzer-core.StackAddressEscape
    clang-analyzer-core.UndefinedBinaryOperatorResult
    clang-analyzer-core.VLASize
    clang-analyzer-core.builtin.BuiltinFunctions
    clang-analyzer-core.builtin.NoReturnFunctions
    clang-analyzer-core.uninitialized.ArraySubscript
    clang-analyzer-core.uninitialized.Assign
    clang-analyzer-core.uninitialized.Branch
    clang-analyzer-core.uninitialized.CapturedBlockVariable
    clang-analyzer-core.uninitialized.UndefReturn
    clang-analyzer-cplusplus.NewDeleteLeaks
    cppcoreguidelines-init-variables
```

Kuva 9. Esimerkkiprojektin tarkistaminen Clang-Tidylla.

```
essor3d@essor3d-VirtualBox: ~/code/thesis-clang-tidy

clang-tidy-14 --use-color -checks=-*,cppcoreguidelines-init-variables,clang-analyzer-cplusplus.NewDe
leteLeaks -p=/home/essor3d/code/thesis-clang-tidy/Build /home/essor3d/code/thesis-clang-tidy/Source/
Utils.cpp
/home/essor3d/code/thesis-clang-tidy/Source/Utils.cpp:6:5: warning: Potential leak of memory pointed
 to by 'result' [clang-analyzer-cplusplus.NewDeleteLeaks]
    return *result;
    ^
/home/essor3d/code/thesis-clang-tidy/Source/Utils.cpp:5:19: note: Memory is allocated
    int* result = new int(0);
                  ^~~~~~~~~~
/home/essor3d/code/thesis-clang-tidy/Source/Utils.cpp:6:5: note: Potential leak of memory pointed to
 by 'result'
    return *result;
    ^
1 warning generated.
clang-tidy-14 --use-color -checks=-*,cppcoreguidelines-init-variables,clang-analyzer-cplusplus.NewDe
leteLeaks -p=/home/essor3d/code/thesis-clang-tidy/Build /home/essor3d/code/thesis-clang-tidy/Source/
main.cpp
/home/essor3d/code/thesis-clang-tidy/Source/main.cpp:7:9: warning: variable 'uninitialized' is not i
nitialized [cppcoreguidelines-init-variables]
    int uninitialized;
        ^
                     = 0
11 warnings generated.
Suppressed 10 warnings (9 in non-user code, 1 with check filters).
Use -header-filter=.* to display errors from all non-system headers. Use -system-headers to display
errors from system headers as well.
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy$
```

Kuva 10. Esimerkkiprojektin koodianalyysin tulokset.

Halutut tarkistukset annetaan ohjelmalle run-clang-tidy argumentin *-checks* arvona [50]. Tässä tapauksessa tarkistetaan projekti alustamattomien muuttujien [51] ja muistivuotojen [17; 52] varalta. Listattujen tarkistusten alussa oleva -* laittaa pois päältä kaikki oletustarkistukset [13]. Ohjelman tulosteesta nähdään, että *clang-analyzer*-luokkaan kuuluvia tarkistuksia on päällä, vaikka kaikki oletustarkistukset on laitettu pois päältä. Nämä ovat Clang Static Analyzerin tarkistuksia [13]. Clang Static Analyzerin tarkistukset ovat riippuvaisia kyseisen ohjelman *core*-luokan tarkistuksista, joten näiden tulee aina olla päällä työkalua käytettäessä [17]. Koska muistivuotojen tarkistus on osa Clang Static Analyzeria, ohjelma on laittanut muut tarvittavat tarkistukset päälle.

Argumentin *-p* arvo on kansio, josta *compile_commands.json* löytyy. Viimeisenä näkyvä nimetön argumentti on säännöllinen lauseke, jonka mukaiset tiedostot annetaan Clang-Tidyn käsiteltäväksi. Näitä nimettömiä argumentteja voi olla useita. Projektin tiedostoja verrataan näihin säännöllisiin lausekkeisiin, ja jos polku on halutunlainen, tiedosto käydään läpi. [50.] Usean säännöllisen lausekkeen käyttämisestä voi olla hyötyä, jos projekti sisältää paljon alikansioita, joista vain osa on tarkistettavia ja osa kolmannen osapuolen koodia, jota ei haluta tarkistaa. run-clang-tidy ymmärtää myös muita argumentteja, jotka on kuvattu työkalun ohjetulosteessa.

3.3 Määrittelytiedostot

Clang-Tidy osaa lukea tiedostoja, joiden avulla voidaan määritellä ohjelman tekemät tarkistukset. Oletuksena etsitään tiedostoa nimellä *.clang-tidy*. [13.] Käytössä oleva versio run-clang-tidy-työkalusta ei osaa käsitellä argumenttina annettavaa määrittelytiedostoa [50], mutta uusin versio osaa [48]. Käytössä olevan version kanssa on siis aina käytettävä *.clang-tidy*-nimistä tiedostoa, jos halutaan käyttää määrittelytiedostoa. Tiedoston käyttäminen kannattaa, koska Clang-Tidyn ajamiseen käytettävä komento yksinkertaistuu tällöin merkittävästi. Esimerkiksi kuvan 9 tapauksessa argumentti *-checks* jäisi pois, jolloin komento olisi paljon lyhyempi. Tällä on merkitystä erityisesti, kun tarkistuksia on paljon.

Clang-Tidyn määrittelytiedostot ovat syntaksiltaan YAML- tai JSON-tyylisiä. Dokumentaatiossa on esimerkki siitä, miltä tiedoston sisällön tulee näyttää. [13.] Kuvassa 11 näkyy esimerkkiprojektiin sopiva määrittelytiedosto. Tiedosto on laitettu projektin juurihakemistoon.

.clang-tidy

```
1   ---
2   Checks: >
3     -*,
4     clang-analyzer-cplusplus.NewDeleteLeaks,
5     cppcoreguidelines-init-variables,
6     readability-identifier-naming
7   WarningsAsErrors: >
8     clang-analyzer-cplusplus.NewDeleteLeaks,
9     cppcoreguidelines-init-variables,
10    readability-identifier-naming
11  HeaderFilterRegex: .*
12  CheckOptions:
13    - key: readability-identifier-naming.LocalVariableCase
14      value: camelBack
15    - key: readability-identifier-naming.GlobalFunctionCase
16      value: camelBack
17  ...
18
```

Kuva 11. Esimerkkiprojektin *.clang-tidy*-määrittelytiedosto.

Määrittelytiedostoon on aiemmasta esimerkistä poiketen lisätty uusi tarkistus, jonka avulla voidaan määrittää projektin nimeämiskäytänteet [53]. Tämän tarkistuksen tarkemmat asetukset voidaan asettaa halutunlaiseksi *CheckOptions*-kohdan avain–arvo-pareina [13]. Jotta myös otsikkotiedostot tarkistetaan [13], *HeaderFilterRegex* asetetaan arvoon *.** kuvan 10 tulosteessa näkyvän

ohjeen tapaisesti. *WarningsAsErrors* sisältää tarkistukset, joista halutaan varoituksen sijasta virheilmoitus [13].

Kohdan *CheckOptions* syntaksi eroaa Clang-Tidyn verkkodokumentaatiosta, jossa avain ja arvo kirjoitetaan samalle riville ilman *key*- ja *value*-sanoja [13]. Kuvassa näkyvä syntaksi on käytössä olevan Clang-Tidy-version ohjetulosteen mukainen [50; 54]. Jossain uudemmassa versiossa dokumentaation mukainen syntaksi on lisätty vaihtoehtoiseksi tavaksi asetusten kirjoittamiseen. Vanhaa tapaa on ehdotettu käytöstä poistettavaksi. [55.]

3.4 Clang-Tidy CI/CD-putkessa

Clang-Tidy voidaan lisätä osaksi jatkuvan integraation ja toimituksen putkea. Tällä tavoin voidaan tehokkaasti pyrkiä varmistamaan ohjelmointikäytänteiden noudattaminen projektissa. Kuten jo aiemmin mainittiin, Clang-Tidy on monissa Linux-jakeluissa mahdollista asentaa järjestelmän paketinhallinnan kautta. Tällöin Clang-Tidyn käyttäminen esimerkiksi Ubuntu-pohjaisessa Docker-kontissa on hyvin helppoa. Samaa Docker-kuvaa käyttäen Clang-Tidyn tarkistukset voidaan ajaa samanlaisessa ympäristössä sekä paikallisesti että projektin yhteisessä versionhallintajärjestelmässä.

Ohjelmointikäytänteiden noudattamisen varmistamiseksi voidaan määrittää Clang-Tidy ilmoittamaan ongelmat varoitusten sijaan virheinä. Versionhallintajärjestelmässä voidaan estää haaran yhdistäminen projektin päähaaraan, jos kaikki ohjelmistokehityksen automaatioputken vaiheet eivät mene hyväksytysti läpi. Jos muutosten tekeminen suoraan päähaaraan on tehty mahdottomaksi, voidaan virheitä käyttämällä tehokkaasti estää ei-ohjelmointikäytänteitä-noudattavan koodin pääseminen päähaaraan. Esimerkiksi Bashissa edellisen komennon onnistunutta suoritusta kuvaava tila voidaan lukea muuttujasta *$?*. Kuvassa 12 näkyy, kun tämän muuttujan arvo tulostetaan run-clang-tidyn ajamisen jälkeen, kun ohjelma on havainnut ongelmia koodissa.

```
11 warnings generated.
Suppressed 10 warnings (9 in non-user code, 1 with check filters).
Use -header-filter=.* to display errors from all non-system headers.
Use -system-headers to display errors from system headers as well.
1 warning treated as error
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy$ echo $?
1
essor3d@essor3d-VirtualBox:~/code/thesis-clang-tidy$
```

Kuva 12. run-clang-tidyn paluuarvon tulostaminen.

Yleinen tapa on, että paluuarvolla 0 ilmaistaan ohjelman suorituksen tapahtuneen ilman ongelmia. Kuten kuvasta näkee, Clang-Tidy toimii tämän tavan mukaisesti kertoen ongelmasta nollasta poikkeavalla arvolla. Tätä ohjelman paluuarvoa voidaan käyttää CI/CD-putkessa, kun halutaan hyödyntää tietoa siitä, etenikö suoritus halutulla tavalla. Versionhallinnassa käytettävät järjestelmät eroavat toisistaan toimintatavoiltaan ja syntaksiltaan, joten käytännön esimerkkiä Clang-Tidyn lisäämisestä CI/CD-putkeen ei tässä opinnäytetyössä anneta.

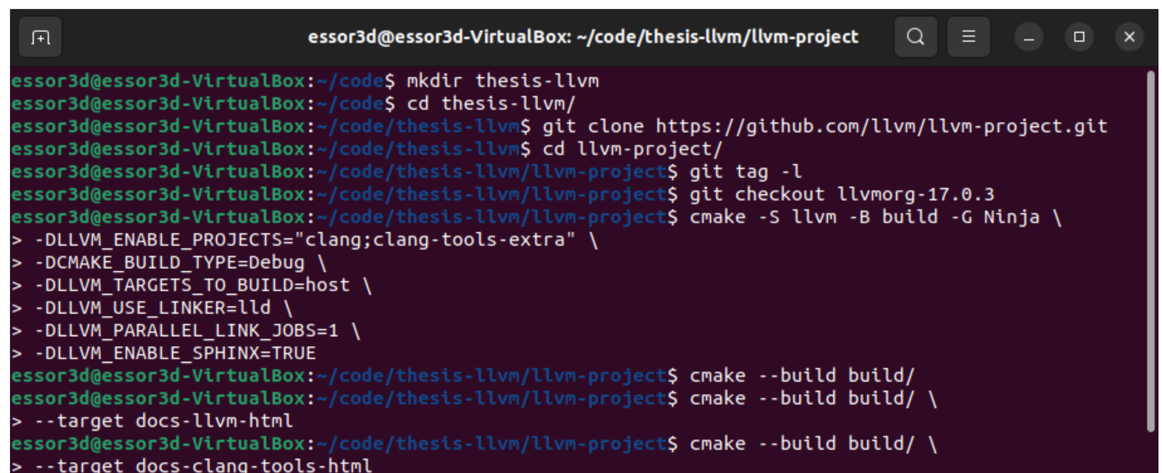
4 Clang-Tidyn laajentaminen

Toimeksiantajan projektin ohjelmointikäytänteet on käyty läpi ja selvitetty, mitä tarkistuksia käytänteiden valvomiseksi on tarpeen toteuttaa itse. Myös Clang-Tidyn peruskäyttö on esitelty. Seuraavaksi voidaan siirtyä käytännön toteutukseen eli ohjelman laajentamiseen uudella tarkistuksella.

Clang-Tidyn laajentaminen edellyttää työkalun lähdekoodien muokkaamista, joten ensiksi ohjelman kokoaminen lähdekoodeista pitää onnistua. Yrityskäytössä Clang-Tidyn laajentaminen on hyvä tehdä uutena moduulina. Moduulin lisäämisen jälkeen voidaan toteuttaa uusi tarkistus, jolle kirjoitetaan automaatiotestit ja dokumentaatio. Käydään läpi koko prosessi vaiheittain.

4.1 Clang-Tidyn asentaminen lähdekoodeista

Clang-Tidyn laajentaminen ohjelmallisesti on mahdollista kokoamalla LLVM ja Clang-Tidy muunnelluista lähdekoodeista tai luomalla jaettu kirjasto, joka linkitetään valmiiksi koottuun LLVM:ään. Jaettua kirjastoa käytettäessä tämä kirjasto tulee kääntää samaa Clang-Tidy-lähdekoodiversiota käyttäen kuin Clang-Tidy, jolla kirjasto aiotaan ladata. [24.] Kootaan LLVM ja Clang-Tidy lähdekoodeista aloittaen puhtaasta koodista ilman muutoksia. Tarvittavat komennot näkyvät kuvassa 13.



Kuva 13. LLVM:n ja Clang-Tidyn koonti lähdekoodeista.

Aluksi luodaan kansio, johon LLVM kloonataan. Kloonatussa hakemistossa tarkistetaan LLVM:n uusin versio, johon siirrytään. Tämän jälkeen projekti generoidaan CMaken avulla. [24; 56.] Koontityökaluksi valitaan Ninja, koska LLVM:n dokumentaation mukaan tämä nopeuttaa merkittävästi projektin koontia [56]. Clang-Tidyn kokoaminen edellyttää aliprojektien *clang* ja *clang-tools-extra* valitsemista mukaan koontiin [24]. Projektista tehdään debug-käännös, koska vianetsinnälle voi olla tarvetta LLVM:ää kehitettäessä [56; 57]. Kohdearkkitehtuuriksi valitaan pelkästään käytössä olevan järjestelmän arkkitehtuuri tilan säästämiseksi ja koonnin nopeuttamiseksi. Linkkerinä käytetään LLD:tä, koska tämä voi nopeuttaa linkittämisvaihetta, jos oletuslinkkerinä on ld. Muistin riittävyyden vuoksi rinnakkaisten linkkeriprosessien määrää rajoitetaan. [56.] Myös Sphinxin avulla luotava HTML-muotoinen dokumentaatio otetaan käyttöön [24; 56; 57]. CMake-märittelyn jälkeen varsinainen koonti tapahtuu [56; 57]. Lopuksi vielä generoidaan dokumentaatio LLVM:lle sekä Clang-Tidylle ja eräille muille työkaluille [24; 56; 57].

LLVM:n kokoaminen käyttää paljon muistia [56]. Opinnäytetyötä tehdessä on huomattu, että myös prosessorin käyttöaste on korkea. Kuten edellä on kuvattu, käytettävän muistin määrän pienentämiseksi on mahdollista rajoittaa rinnakkaisten linkkeriprosessien määrää määriteltäessä projektia CMaken avulla. Tämäkään ei välttämättä riitä, vaan koonti saattaa epäonnistua lähellä loppua. Aloitettaessa koonti CMaken kautta uudelleen on mahdollista vielä muuttaa koontiin käytettävien prosessien enimmäismäärää alkuperäisestä määrittelystä argumentin *-j* avulla [58]. Tämän seurauksena linkkeriprosessien lisäksi myös kääntämisprosessien määrää rajoitetaan, mikä vähentää muistin ja prosessorin käyttöastetta. Tällöin koonti hidastuu merkittävästi, mutta saattaa onnistua loppuun asti. Prosessien määrää rajoittava komento on muotoa *cmake --build build/ -j 1*, jossa prosessien määrä voidaan valita lukua muuttamalla. Virtuaalikonetta käytettäessä voi olla järkevää lisätä koneen käyttöön annettavan muistin ja prosessorin ytimien määrää, jos mahdollista. Tällöin koonti onnistuu varmemmin myös ilman, että lopussa joudutaan rajoittamaan koontiin käytettävien prosessien määrää alkuperäistä määrittelyä pienemmäksi.

LLVM voidaan koonnin jälkeen asentaa järjestelmään komennolla *sudo cmake --build build/ --target install* [56]. Tässä *sudo* ei välttämättä ole tarpeen, jos asennus tehdään johonkin muualle kuin oletussijaintiin, joka edellyttää järjestelmänvalvojan oikeuksia. Asennus ei ole pakollinen ja kootut ohjelmat toimivat suoraan build-kansiosta ilman asennuksen suorittamista, joten tallennustilan säästämiseksi on mahdollista jättää asennus tekemättä. Tämä on erityisen perusteltua debug-koonnille ja silloin, kun kootuista ohjelmista halutut kopioidaan Docker-konttiin.

Ennen muutosten tekemistä on hyvä testata, että koonti on onnistunut. LLVM:n ja Clangin yksikkö- ja regressiotestit voidaan ajaa komennolla *cmake --build build/ --target check-all* [56; 59].

Tämä voi kuitenkin viedä paljon aikaa ja tallennustilaa. Clang-Tidyn testit voidaan ajaa myös komennolla *cmake --build build/ --target check-clang-tools* [24]. Jälkimmäinen vaihtoehto voi olla huomattavasti nopeampi ja vähemmän tallennustilaa vaativa, koska kaikkia LLVM:ään liittyviä testejä ei tarvitse suorittaa.

4.2 Uuden tarkistuksen lisääminen ohjelmallisesti

Clang-Tidyn tarkistukset on jaettu moduuleihin, joita ovat esimerkiksi performance, llvm ja cert [14; 24]. Omia tarkistuksia varten on hyvä lisätä uusi moduuli. Tämä auttaa pysymään perillä siitä, mitä tarkistuksia on itse luotu. Myös nimeämiseen ja forkin synkronointiin liittyviä konflikteja voidaan välttää oman moduulin avulla. LLVM sisältää apuohjelman add_new_check.py, jonka avulla voidaan lisätä uusi tarkistus olemassa olevaan moduuliin [24]. Aloitetaan uusien tarkistusten lisääminen moduulin luomisella. Ennen muutosten tekemistä on hyvä luoda päähaara forkatulle projektille ja sivuhaara, jossa varsinainen kehitys tapahtuu.

Uuden moduulin luomiseen ei löydy ohjetta LLVM:n dokumentaatiosta. Projektia tutkimalla voidaan kuitenkin päätellä, miten moduulin lisääminen tapahtuu. LLVM-projektin hakemisto *llvm/clang-tools-extra/clang-tidy* sisältää alihakemiston jokaiselle moduulille [24; 31]. Tutkimalla *boost*-moduulia saadaan selville, että moduulin hakemisto sisältää *CMakeLists.txt*-tiedoston, moduulin määrittelevän kooditiedoston *BoostTidyModule.cpp* sekä tarkistuksen sisältävät tiedostot *UseToStringCheck.h* ja *UseToStringCheck.cpp*. Muiden moduulien hakemistorakenne on samantyyppinen, mutta nimiltään ja tarkistuksiltaan erilainen. [31.] Netistä löytyy pari esimerkkiä, joista nähdään, että uusi moduuli pitää myös lisätä hakemiston *llvm/clang-tools-extra/clang-tidy* tiedostoon *CMakeLists.txt* [60; 61]. Tämän lisäksi samasta hakemistosta löytyvään *ClangTidyForceLinker.h*-tiedostoon tulee lisätä muuttuja, jonka avulla moduuli linkitetään Clang-Tidyyn [24; 60; 61]. Moduulin lisäämiseen tarvittavat koodit näkyvät kuvissa 14, 15, 16, 17 ja 18.

```
clang-tools-extra > clang-tidy > M CMakeLists.txt
50
51  # Checks.
52  # If you add a check, also add it to ClangTidyForceLinker.h in this directory.
53  add_subdirectory(android)
54  add_subdirectory(abseil)
55  add_subdirectory(altera)
56  add_subdirectory(bittium) # Notice: This line was added by Bittium.
57  add_subdirectory(boost)
58  add_subdirectory(bugprone)
```

Kuva 14. Clang-Tidyn muokattu *CMakeLists.txt*-tiedosto, osa 1.

```
80  set(ALL_CLANG_TIDY_CHECKS
81    clangTidyAndroidModule
82    clangTidyAbseilModule
83    clangTidyAlteraModule
84    clangTidyBittiumModule # Notice: This line was added by Bittium.
85    clangTidyBoostModule
86    clangTidyBugproneModule
```

Kuva 15. Clang-Tidyn muokattu *CMakeLists.txt*-tiedosto, osa 2.

Ylemmässä kuvassa näkyvän lisäyksen jälkeen CMake lukee määritellystä hakemistosta löytyvän *CMakeLists.txt*-tiedoston, joka kuuluu uudelle moduulille. Alemmassa kuvassa taas moduuli lisätään muuttujaan *ALL_CLANG_TIDY_CHECKS*. LLVM:n lähdekoodeja lukemalla selviää, että muuttujan sisältämät moduulit linkitetään Clang-Tidyyn kirjastoina [62; 63].

```
clang-tools-extra > clang-tidy > C ClangTidyForceLinker.h > {} clang > {} tidy
26
27  // This anchor is used to force the linker to link the AndroidModule.
28  extern volatile int AndroidModuleAnchorSource;
29  static int LLVM_ATTRIBUTE_UNUSED AndroidModuleAnchorDestination =
30      AndroidModuleAnchorSource;
31
32  // ----- Notice: These lines were added by Bittium. -----
33  //
34  // This anchor is used to force the linker to link the BittiumModule.
35  extern volatile int BittiumModuleAnchorSource;
36  static int LLVM_ATTRIBUTE_UNUSED BittiumModuleAnchorDestination =
37      BittiumModuleAnchorSource;
38  //
39  // ----- Notice: Bittium additions end. -----
40
```

Kuva 16. Clang-Tidyn muokattu *ClangTidyForceLinker.h*-tiedosto.

Yläpuolella olevassa kuvassa on C++-koodia, jonka avulla saadaan linkitettyä moduuli Clang-Tidyyn. Muuttujan *BittiumModuleAnchorSource* määrittely tapahtuu toisessa C++-tiedostossa, joka näkyy hieman alempana. Muuttuja on *volatile*-määreinen, koska tällä tavoin voidaan kertoa kääntäjälle, että käyttämätöntä muuttujaa ei tulisi optimoida pois, jos sen arvo luetaan ohjelman aikana [64; 65; 66, s. 11, 173]. Linkitys tapahtuu *extern*-määreellä, jonka avulla muuttuja linkitetään ulkoisesti jossain muualla määriteltyyn muuttujaan. Alempi muuttuja *BittiumModuleAnchor-Destination* lukee edellä mainitun muuttujan arvon. Tämä muuttuja on staattinen, jolloin sillä on käännösyksikön sisäinen linkitys. Kääntäjän varoitukset käyttämättömästä muuttujasta saadaan pois attribuutilla *LLVM_ATTRIBUTE_UNUSED*, mutta varoitusten pois jättäminen toimii vain yhteensopivilla kääntäjillä [67].

```
clang-tools-extra > clang-tidy > bittium > CMakeLists.txt
1  # Notice: This file was added by Bittium.
2
3  set(LLVM_LINK_COMPONENTS
4    FrontendOpenMP
5    Support
6    )
7
8  add_clang_library(clangTidyBittiumModule
9    BittiumTidyModule.cpp
10
11   LINK_LIBS
12   clangTidy
13   clangTidyUtils
14   )
15
16 clang_target_link_libraries(clangTidyBittiumModule
17   PRIVATE
18   clangAST
19   clangASTMatchers
20   clangBasic
21   clangLex
22   )
23
```

Kuva 17. Uuden moduulin *CMakeLists.txt*-tiedosto.

Moduulin *CMakeLists.txt*-tiedosto on luotu muiden moduulien pohjalta. LLVM:n dokumentaatiosta ei ole löytynyt ohjetta, millainen tiedoston pitäisi olla Clang-Tidyn moduuleille. Dokumentaatiossa on kuitenkin esimerkkitiedosto uutta työkalua varten [68], ja toisaalla on kehotus katsoa mallia olemassa olevista työkaluista [69]. LLVM myös sisältää mallipohjan uudelle työkalulle hakemistossa *clang-tools-extra/tool-template* [31].

CMakeLists.txt määrittelee moduulin Clang-kirjastoksi ja sisältää linkitettävät komponentit ja kirjastot. Dokumentaatiosta ei käy ilmi, miten komponentit ja kirjastot eroavat toisistaan. Dokumentaation perusteella ei myöskään selviä, miten muuttuja *LLVM_LINK_COMPONENTS*, argumentti *LINK_LIBS* ja funktio *clang_target_link_libraries* eroavat toisistaan. Muiden moduulien perusteella voidaan kuitenkin luottaa siihen, että määrittely on oikein. Jos jotain puuttuu, viimeistään linkkerivirheiden pitäisi paljastaa ongelmat tiedostossa. Jos jotain kirjastoa ei taas tarvitakaan, turhaa linkitystä ei pitäisi tapahtua. Dokumentaatiossa on kuvattu, mitä eri kirjastot sisältävät. Kirjastojen toiminnallisuuteen kuuluvat esimerkiksi lähdekoodin käsittely ja diagnostiikka, abstraktit syntaksipuut, tietorakenteet sekä komentoriviargumenttien käsittely [24; 70].

```
clang-tools-extra > clang-tidy > bittium > BittiumTidyModule.cpp > ...
 1  // Notice: This file was added by Bittium.
 2
 3  #include "../ClangTidy.h"
 4  #include "../ClangTidyModule.h"
 5  #include "../ClangTidyModuleRegistry.h"
 6
 7  namespace clang {
 8  namespace tidy {
 9  namespace bittium {
10
11  class BittiumModule : public ClangTidyModule {
12  public:
13    void addCheckFactories(ClangTidyCheckFactories &CheckFactories) override {
14    }
15  };
16
17  // Register the BittiumModule using this statically initialized variable.
18  static ClangTidyModuleRegistry::Add<BittiumModule> X("bittium-module",
19                                                        "Add Bittium checks.");
20
21  } // namespace bittium
22
23  // This anchor is used to force the linker to link in the generated object file
24  // and thus register the BittiumModule.
25  volatile int BittiumModuleAnchorSource = 0;
26
27  } // namespace tidy
28  } // namespace clang
29
```

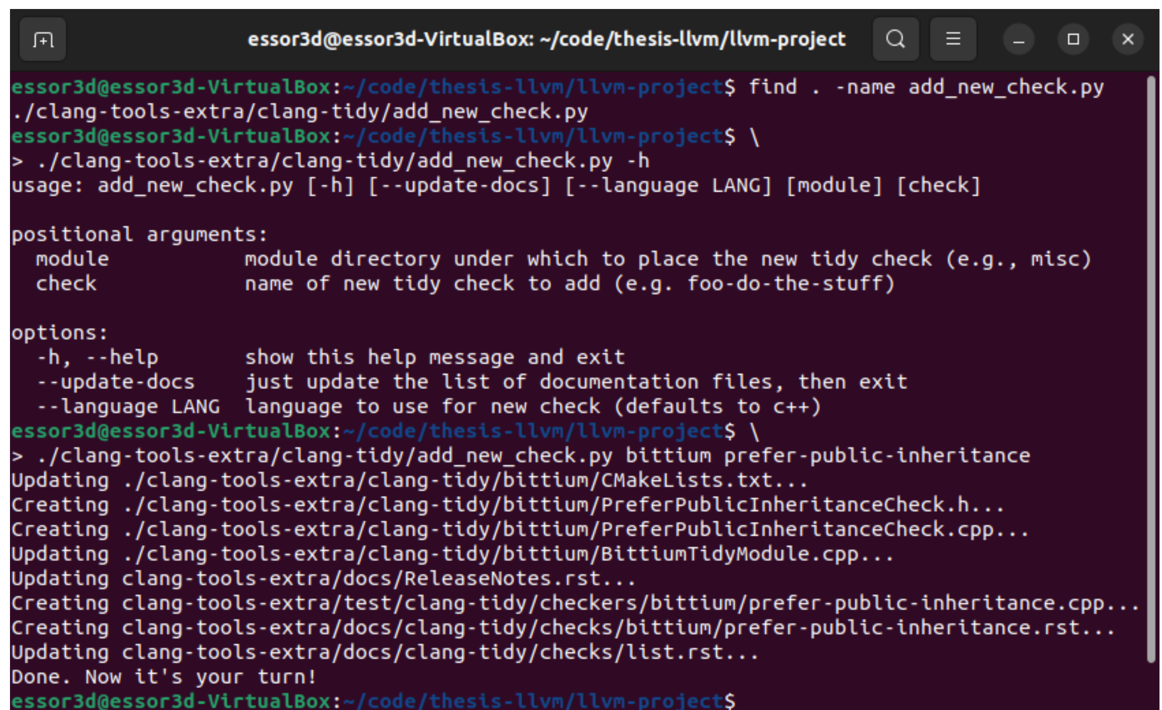
Kuva 18. Uuden moduulin tiedosto *BittiumTidyModule.cpp*.

Moduulin päätiedosto vastaa moduulin ja tarkistusten rekisteröinnistä. Tiedoston sisältä perustuu taas muihin moduuleihin. Moduulia varten luodaan luokka, joka perii *ClangTidyModule*-luokan. Yliluokka on abstrakti ja sisältää puhtaan virtuaalisen metodin *addCheckFactories*, joka pitää ylikuormittaa aliluokassa [71]. Koska moduuli ei vielä sisällä tarkistuksia, metodin toteutus on tyhjä. Moduulin rekisteröinti tapahtuu olion *X* avulla samalla tavoin kuin muissakin moduuleissa. *BittiumModuleAnchorSource* on määritelty, jotta muuttujan arvon toisessa tiedostossa lukemalla saadaan varmistettua moduulin linkitys aiemmin kuvatulla tavalla.

Uuden moduulin lisäämisen jälkeen projekti voidaan määritellä uudestaan CMakella ja koonti toistaa. Tässä ei pitäisi mennä kauaa, koska muutokset projektiin ovat pieniä. Testien avulla voidaan varmistaa, että mitään ei ole mennyt rikki.

Uuden tarkistuksen lisääminen onnistuu skriptillä add_new_check.py. Skriptin suoritus kuitenkin epäonnistuu, jos uuden moduulin testeille ja dokumentaatiolle ei ole luotu kansiota. Kansioilla tulee olla sama nimi kuin moduulin sisältävällä kansiolla, ja nämä tulee laittaa hakemistoon *clang-tools-extra/docs/clang-tidy/checks* ja *clang-tools-extra/test/clang-tidy/checkers*. [72.] Kansioiden

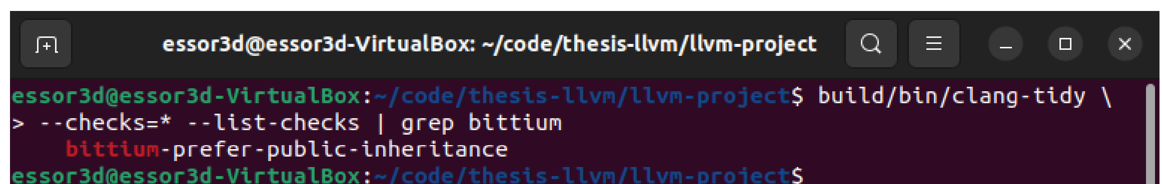
luomisen jälkeen uusi tarkistus voidaan lisätä edellä mainitun skriptin avulla. Skriptin ajo näkyy kuvassa 19.



Kuva 19. Uuden tarkistuksen lisääminen Clang-Tidyyn.

Uuden tarkistuksen lisäävä skripti luo uuden otsikkotiedoston ja toteutuksen sisältävän kooditiedoston. Toteutuksen sisältävä tiedosto lisätään *CMakeLists.txt*-tiedostossa kirjaston määrittelyyn. Otsikkotiedosto sisällytetään moduulin päätiedostoon, jossa tarkistus rekisteröidään *addCheckFactories*-metodissa. Uusi tarkistus lisätään julkaisutietoihin ja tälle luodaan testi, dokumentaatiosivu sekä lisäys tarkistusten luetteloon.

Tarkistuksen lisäämisen jälkeen projekti voidaan taas määritellä ja koota. Muutokset ovat taas pieniä, joten aikaa ei pitäisi nytkään mennä kauaa. Uuden tarkistuksen nimen pitäisi tulostua, kun listataan kaikki Clang-Tidyn tarkistukset. Kuvassa 20 näkyy, kun uuden tarkistuksen nimi etsitään ohjelman tulosteesta.



Kuva 20. Uuden tarkistuksen hakeminen tarkistusten luettelosta.

Aloitetaan tarkistuksen toteuttaminen esimerkkitiedoston avulla. Tiedoston sisältö näkyy kuvassa 21. Tiedosto voidaan sijoittaa johonkin LLVM:n ulkopuoliseen hakemistoon.

main.cpp > ...

```
1   class Base
2   {
3   };
4
5   class DerivedPublic : public Base
6   {
7   };
8
9   class DerivedProtected : protected Base
10  {
11  };
12
13  class DerivedPrivate : private Base
14  {
15  };
16
17  int main()
18  {
19      return 0;
20  }
21
```

Kuva 21. Esimerkkitiedosto uutta tarkistusta varten.

Uudella tarkistuksella halutaan etsiä ei-julkiset periytymiset. Apuna voidaan käyttää Clang-Checkiä, jonka avulla voidaan tulostaa koodi AST-muodossa [24]. Ohjelma toimii myös yksittäisten tiedostojen kanssa, joten tarvetta varsinaiselle testiprojektille ei ole [73]. AST-solmuja on myös mahdollista hakea merkkijonojen avulla [74]. Dokumentaation avulla saadaan muodostettua komento, jonka ajo näkyy kuvassa 22.

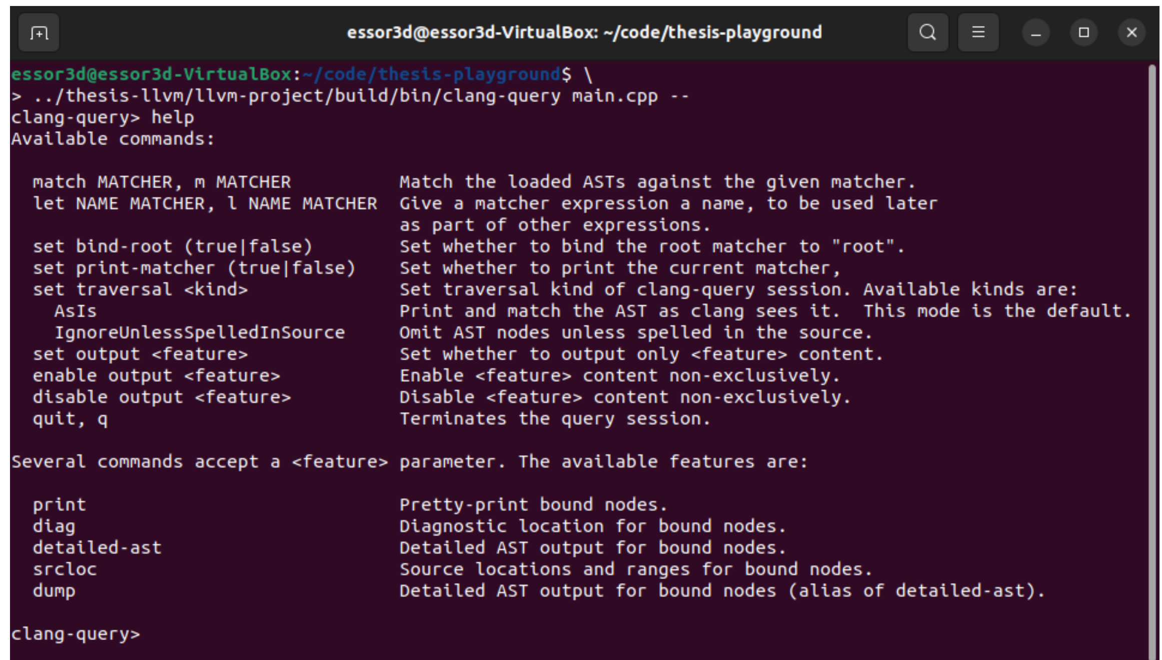
```
essor3d@essor3d-VirtualBox: ~/code/thesis-playground
essor3d@essor3d-VirtualBox:~/code/thesis-playground$ \
> ../thesis-llvm/llvm-project/build/bin/clang-check \
> -ast-dump -ast-dump-filter=Derived main.cpp --
Dumping DerivedPublic:
CXXRecordDecl 0x555cad03ff00 </home/essor3d/code/thesis-playground/main.cpp:5:1, line:7:1> l
ine:5:7 class DerivedPublic definition
|-DefinitionData pass_in_registers empty aggregate standard_layout trivially_copyable trivia
l literal has_constexpr_non_copy_move_ctor can_const_default_init
| |-DefaultConstructor exists trivial constexpr needs_implicit defaulted_is_constexpr
| |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveConstructor exists simple trivial needs_implicit
| |-CopyAssignment simple trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveAssignment exists simple trivial needs_implicit
| `-Destructor simple irrelevant trivial needs_implicit
|-public 'Base':'Base'
`-CXXRecordDecl 0x555cad040090 <col:1, col:7> col:7 implicit class DerivedPublic

Dumping DerivedProtected:
CXXRecordDecl 0x555cad040138 </home/essor3d/code/thesis-playground/main.cpp:9:1, line:11:1>
line:9:7 class DerivedProtected definition
|-DefinitionData pass_in_registers empty standard_layout trivially_copyable trivial literal
has_constexpr_non_copy_move_ctor can_const_default_init
| |-DefaultConstructor exists trivial constexpr needs_implicit defaulted_is_constexpr
| |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveConstructor exists simple trivial needs_implicit
| |-CopyAssignment simple trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveAssignment exists simple trivial needs_implicit
| `-Destructor simple irrelevant trivial needs_implicit
|-protected 'Base':'Base'
`-CXXRecordDecl 0x555cad040298 <col:1, col:7> col:7 implicit class DerivedProtected

Dumping DerivedPrivate:
CXXRecordDecl 0x555cad040340 </home/essor3d/code/thesis-playground/main.cpp:13:1, line:15:1>
 line:13:7 class DerivedPrivate definition
|-DefinitionData pass_in_registers empty standard_layout trivially_copyable trivial literal
has_constexpr_non_copy_move_ctor can_const_default_init
| |-DefaultConstructor exists trivial constexpr needs_implicit defaulted_is_constexpr
| |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveConstructor exists simple trivial needs_implicit
| |-CopyAssignment simple trivial has_const_param needs_implicit implicit_has_const_param
| |-MoveAssignment exists simple trivial needs_implicit
| `-Destructor simple irrelevant trivial needs_implicit
|-private 'Base':'Base'
`-CXXRecordDecl 0x555cad040498 <col:1, col:7> col:7 implicit class DerivedPrivate
```

Kuva 22. Esimerkkitiedoston aliluokkien AST-tuloste.

Argumentin *-ast-dump* avulla voidaan tulostaa muodostettava AST. Tulostusta suodatetaan argumentilla *-ast-dump-filter*, joka etsii merkkijonoa koodista. [74.] Viimeisenä argumenttina on käsiteltävä tiedosto ja --, jonka avulla estetään työkalua etsimästä käännökseen liittyvää *compile-commands.json*-tiedostoa [73][75].

AST:n tulostamisen jälkeen voidaan alkaa etsiä sopivaa hakulausetta ei-julkiselle periytymiselle. Tässä voidaan hyödyntää ohjelmaa clang-query [24]. Ohjelman avulla on mahdollista tutkia AST-muotoista koodia interaktiivisesti komentorivin kautta [76]. Komentojen kirjoittamista helpottaa automaattinen täydennys, joka toimii sarkainnäppäimen avulla [77]. Opinnäytetyön ohjeiden mukaan kootussa clang-queryssa automaattinen täydennys ei välttämättä toimi, kun taas pakettinhallinnan kautta asennetussa toimii. Tämä johtuu puuttuvasta libedit-kirjastosta [78]. Kirjaston

voi asentaa järjestelmän paketinhallinnasta, minkä jälkeen CMake-määrittelyn ja koonnin uudelleen ajamalla automaattinen täydennys toimii myös itse kootussa clang-queryssa. Koonnissa ei pitäisi taaskaan mennä kauaa, koska koko projektia ei tarvitse koota uudestaan. Kuvassa 23 näkyy clang-queryn käynnistäminen ja ohjetuloste.



Kuva 23. clang-queryn ohjetuloste.

Ohjelman avulla on mahdollista tutkia yksittäistä tiedostoa antamalla argumentteina tiedoston nimi ja -- [76] samaan tapaan kuin ClangCheckin tapauksessa. Ohjetulosteen saa näkyviin interaktiivisille ohjelmille tuttuun tapaan komennolla *help*. Kirjoittamatta mitään ja painamalla pelkkää sarkainnäppäintä saadaan tulostettua lista mahdollisista komennoista ilman tarkempaa selitettä.

AST-tulosteen perusteella haku voidaan aloittaa *CXXRecordDecl*-tyyppisestä AST-solmusta. Dokumentaation mukaan tätä vastaa hakulause *cxxRecordDecl* [79]. Kuvassa 24 näkyy hakulause, joka clang-queryn avulla saadaan muodostettua. Kuvaan liittyen on hyvä huomioida, että kuvassa näkyy vain lopputulos, joka on saatu aikaiseksi useiden kokeilujen jälkeen. Oikean hakulauseen löytämisessä automaattisesta täydennyksestä ja AST-dokumentaatiosta on paljon apua. Myös muiden kirjoittamia lauseita muista moduuleista ja esimerkkejä netistä kannattaa hyödyntää.

```
essor3d@essor3d-VirtualBox: ~/code/thesis-playground

essor3d@essor3d-VirtualBox:~/code/thesis-playground$ \
> ../thesis-llvm/llvm-project/build/bin/clang-query main.cpp --
clang-query> match cxxRecordDecl(hasDirectBase(anyOf(isPrivate(), isProtected())))

Match #1:

/home/essor3d/code/thesis-playground/main.cpp:9:1: note: "root" binds here
    9 | class DerivedProtected : protected Base
      | ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   10 | {
      | ~
   11 | };
      | ~

Match #2:

/home/essor3d/code/thesis-playground/main.cpp:13:1: note: "root" binds here
   13 | class DerivedPrivate : private Base
      | ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   14 | {
      | ~
   15 | };
      | ~
2 matches.
clang-query>
```

Kuva 24. AST-hakulauseen testaaminen clang-querylla.

Nyt voidaan toteuttaa tarkistus koodiin. Liitteessä 2 näkyy tiedostot, jotka toimivat pohjana uudelle tarkistukselle. Kuvassa 25 näkyy tarkistuksen toteutus.

```
clang-tools-extra > clang-tidy > bittium > PreferPublicInheritanceCheck.cpp > ...
1  // Notice: This file was added by Bittium.
2
3  //===--- PreferPublicInheritanceCheck.cpp - clang-tidy --------------------===//
4  //
5  // Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
6  // See https://llvm.org/LICENSE.txt for license information.
7  // SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
8  //
9  //===----------------------------------------------------------------------===//
10
11 #include "PreferPublicInheritanceCheck.h"
12 #include "clang/AST/ASTContext.h"
13 #include "clang/ASTMatchers/ASTMatchFinder.h"
14
15 using namespace clang::ast_matchers;
16
17 namespace clang::tidy::bittium {
18
19 static constexpr llvm::StringLiteral NonpublicInheritanceBoundName
20     = "nonpublicInheritance";
21
22 void PreferPublicInheritanceCheck::registerMatchers(MatchFinder *Finder) {
23   Finder->addMatcher(cxxRecordDecl(hasDirectBase(anyOf(isPrivate(),
24                                                        isProtected())))
25                          .bind(NonpublicInheritanceBoundName),
26                      this);
27 }
28
29 void PreferPublicInheritanceCheck::check(const MatchFinder::MatchResult &Result) {
30   const auto *MatchedDecl
31       = Result.Nodes.getNodeAs<CXXRecordDecl>(NonpublicInheritanceBoundName);
32
33   for (const auto &base : MatchedDecl->bases()) {
34     if (base.getAccessSpecifier() != AccessSpecifier::AS_public) {
35       diag(base.getBaseTypeLoc(), "avoid using non-public inheritance")
36           << MatchedDecl
37           << FixItHint::CreateInsertion(base.getBaseTypeLoc(), "public");
38       diag(base.getBaseTypeLoc(),
39            "consider using public inheritance and composition",
40            DiagnosticIDs::Note);
41     }
42   }
43 }
44
45 } // namespace clang::tidy::bittium
46
```

Kuva 25. Uuden tarkistuksen toteutus.

Metodissa *registerMatchers* nimensä mukaisesti rekisteröidään tarkistuksen käyttämät hakulauseet. Tämä tapahtuu *MatchFinder*-luokan metodilla *addMatcher*. Parametrina annetaan käytettävä hakulause ja *MatchCallback*-osoitin. AST-solmuja läpikäydessä suoritetaan toisena parametrina annettavan osoittimen kautta löytyvä toiminnallisuus niille solmuille, jotka vastaavat hakulausetta [80]. Clang-Tidyyn lisättävät tarkistukset perivät *ClangTidyCheck*-luokan, joka taas perii luokan *MatchCallback*. Ylikuormitettava metodi *check* sisältää toiminnallisuuden, jota kutsutaan aina halutunlaisen solmun löytyessä. [81.] Tarkistukset toimivat siis polymorfismin avulla. *Match-*

Result-tyyppinen tietue sisältää hakua vastaavaan löydökseen liittyvää tietoa. Tietueen kautta esimerkiksi päästään käsiksi löydettyyn AST-solmuun *getNodeAs*-metodin avulla [80; 82]. Solmua käsitellään *bind*-metodilla annetun nimen avulla [82; 83].

Rekisteröitävä hakulause on sama, joka muodostettiin ClangCheckin ja clang-queryn avulla. Hakulause etsii koodista luokat, tietuetyypit ja unionit, jotka perivät jonkin toisen luokan yksityisellä tai aliluokkanäkyvyysmääreellä [79; 84]. Tällaisen luokan löytyessä käydään läpi kaikki yliluokat [85] ja annetaan diagnostiikkaviesti [81] kaikista ei-julkisista perimisistä [85]. Yliluokat käydään läpi, koska luokka voi periä useita luokkia. Tällöin voidaan varmistaa, että kaikki perimiset ovat halutunlaisia. Tarvittaessa siis annetaan useita varoituksia, jos luokka perii useita luokkia ei-halutulla tavalla.

Uusi testi saadaan käyttöön kokoamalla LLVM uudestaan. Kuvassa 26 näkyy, kun tarkistus ajetaan aiemmin esitellylle esimerkkitiedostolle.

```
essor3d@essor3d-VirtualBox: ~/code/thesis-playgr...
essor3d@essor3d-VirtualBox:~/code/thesis-playground$ \
> ../thesis-llvm/llvm-project/build/bin/clang-tidy \
> --checks="-*,bittium-prefer-public-inheritance" \
> main.cpp --
2 warnings generated.
/home/essor3d/code/thesis-playground/main.cpp:9:36: warning: avoid using
non-public inheritance [bittium-prefer-public-inheritance]
    9 | class DerivedProtected : protected Base
      |                                    ^
      |                                    public
/home/essor3d/code/thesis-playground/main.cpp:9:36: note: consider using
public inheritance and composition
/home/essor3d/code/thesis-playground/main.cpp:13:32: warning: avoid using
 non-public inheritance [bittium-prefer-public-inheritance]
   13 | class DerivedPrivate : private Base
      |                                ^
      |                                public
/home/essor3d/code/thesis-playground/main.cpp:13:32: note: consider using
 public inheritance and composition
essor3d@essor3d-VirtualBox:~/code/thesis-playground$
```

Kuva 26. Uuden tarkistuksen ajaminen esimerkkitiedostolle.

Kuten kuvasta näkee, tarkistus toimii oikein esimerkkitiedoston tapauksessa. Diagnostiikkaan kuuluu varoitus löydetystä ongelmasta ja vihje ongelman korjaamiseksi. Kuvassa näkyvän korjausehdotuksen nuoli osoittaa perittävän luokan nimeen eikä periytymisen näkyvyysmääreeseen. Tämä on tarkoituksellista, koska C++ sallii eksplisiittisen näkyvyysmääreen pois jättämisen, jolloin käytetään implisiittistä oletusmäärettä. Tällaisessa tapauksessa varmastikin selkeintä on osoittaa

perittävän luokan nimeen. Tämä on selkeä ilmaisutapa myös silloin, kun näkyvyysmääre on kirjoitettu koodiin, eikä nykyinen ratkaisu myöskään edellytä uutta ehtoa koodiin nuolen paikan määräämiseksi.

Muuttamalla esimerkkitiedostoa saadaan varmistettua, että tarkistus toimii halutulla tavalla myös moniperinnälle, oletusnäkyvyysmääreille ja tietuetyypeille. Tarkistus näyttää siis toimivan suunnitellusti.

4.3 Tarkistuksen testaaminen

Uuden tarkistuksen toteutuksen jälkeen voidaan taas ajaa testit jo aiemmin esitellyllä komennolla *cmake --build build/ --target check-clang-tools*. Uudelle tarkistukselle luotu testi epäonnistuu, koska tarkistuksen toteutus on muuttunut generoidusta. Kirjoitetaan seuraavaksi testit uudelle tarkistukselle käyttäen pohjana ohjelman add_new_check.py generoimaa testitiedostoa. Tämä tiedosto näkyy liitteessä 2. Tarkistusta varten kirjoitettu testi näkyy kuvassa 27.

```
clang-tools-extra > test > clang-tidy > checkers > bittium > prefer-public-inheritance.cpp > ...
 1  // RUN: %check_clang_tidy %s bittium-prefer-public-inheritance %t
 2
 3  // CHECK-FIXES: class Base1
 4  class Base1
 5  {
 6  };
 7
 8  class Base2
 9  {
10  };
11
12  // CHECK-FIXES: class Derived1 : public Base1
13  class Derived1 : public Base1
14  {
15  };
16
17  // CHECK-MESSAGES: :[[@LINE+2]]:28: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
18  // CHECK-FIXES: class Derived2 : public Base1
19  class Derived2 : protected Base1
20  {
21  };
22
23  // CHECK-MESSAGES: :[[@LINE+2]]:26: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
24  // CHECK-FIXES: class Derived3 : public Base1
25  class Derived3 : private Base1
26  {
27  };
28
29  // CHECK-MESSAGES: :[[@LINE+2]]:32: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
30  // CHECK-FIXES: class Derived4 : public Base1, public Base2
31  class Derived4 : public Base1, Base2
32  {
33  };
34
```

Kuva 27. Uuden tarkistuksen testitiedosto.

Tiedosto sisältää 2 yliluokkaa, jotka eivät ole tarkistuksen kannalta ongelmallisia. Lisäksi tiedostossa on 4 aliluokkaa, joista 3 on puutteellisia. *CHECK-FIXES*-alkuisten kommenttien avulla tarkistetaan, että kommentin mukainen sisältö löytyy tiedostosta. Tällä tavalla voidaan varmistaa, että korjausten jälkeen tiedoston sisältö on halutunlainen. Voidaan siis varmistaa, että oikeanlaista koodia ei muuteta tarpeettomasti, mutta toisaalta havaitut puutteet korjataan. *CHECK-MESSAGES* taas määrittelee halutut diagnostiikkaviestit, joiden tulostus testeillä tarkastetaan. [24.]

Testien avulla paljastuu, että tarkistuksen toteutus sisältää virheen. Ongelman ilmaiseva kohta tulosteesta näkyy kuvassa 28.

```
 //
 //
-class Derived2 : protected Base1
+class Derived2 : protected publicBase1
 {
 };

 //
 //
-class Derived3 : private Base1
+class Derived3 : private publicBase1
 {
 };

 //
 //
-class Derived4 : public Base1, Base2
+class Derived4 : public Base1, publicBase2
 {
 };
```

Kuva 28. Testauksen paljastama ongelma.

Clang-Tidyn automaattinen korjaus kirjoittaa näkyvyysmääreen *public* välittömästi perittävän luokan nimen eteen ilman väliä, eikä alkuperäistä näkyvyysmäärettä poisteta. Tämä ei ole oikeanlaista C++-koodia, joten tarkistuksen toteutus pitää korjata. Korjattu tarkistus näkyy kuvassa 29.

clang-tools-extra > clang-tidy > bittium > PreferPublicInheritanceCheck.cpp > ...

```cpp

void PreferPublicInheritanceCheck::check(const MatchFinder::MatchResult &Result) {
  const auto *MatchedDecl
      = Result.Nodes.getNodeAs<CXXRecordDecl>(NonpublicInheritanceBoundName);

  for (const auto &base : MatchedDecl->bases()) {
    if (base.getAccessSpecifier() == AccessSpecifier::AS_public) {
      continue;
    }

    if (base.getAccessSpecifierAsWritten() != AccessSpecifier::AS_none) {
      if (!CreateReplacements) {
        diag(base.getBaseTypeLoc(), "avoid using non-public inheritance")
            << MatchedDecl;
        diag(base.getBaseTypeLoc(),
            "consider using public inheritance and composition",
            DiagnosticIDs::Note);
      }
      else {
        auto &sourceManager = *Result.SourceManager;
        auto &langOpts = Result.Context->getLangOpts();
        auto baseDeclSourceText = Lexer::getSourceText(
                                      Lexer::getAsCharRange(
                                          base.getSourceRange(),
                                          sourceManager,
                                          langOpts),
                                      sourceManager,
                                      langOpts);
        llvm::Regex regex("(private|protected)[[:space:]]+");
        auto fixedDecl = regex.sub("public ", baseDeclSourceText);

        diag(base.getBaseTypeLoc(), "avoid using non-public inheritance")
            << MatchedDecl
            << FixItHint::CreateReplacement(base.getSourceRange(), fixedDecl);
        diag(base.getBaseTypeLoc(),
            "consider using public inheritance and composition",
            DiagnosticIDs::Note);
      }
    }
    else {
      diag(base.getBaseTypeLoc(), "avoid using non-public inheritance")
          << MatchedDecl
          << FixItHint::CreateInsertion(base.getBaseTypeLoc(), "public ");
      diag(base.getBaseTypeLoc(),
          "consider using public inheritance and composition",
          DiagnosticIDs::Note);
    }
  }
}

```

Kuva 29. Korjattu tarkistus.

Jos perimisen näkyvyysmäärettä ei ole kirjoitettu, tarkistus toimii aiemman tapaisesti, mutta automaattisen korjauksen yhteydessä lisättävän näkyvyysmääreen jälkeen lisätään myös väli. Jos taas näkyvyysmääre on kirjoitettu, mutta se ei ole halutunlainen, oletuksena tästä annetaan varoitus ilman korjausehdotusta automaattiselle korjaukselle. Korjausehdotuksen myös näihin ti-

lanteisiin saa päälle tarkistukseen liittyvällä asetuksella *CreateReplacements*, jolloin automaattisen korjauksen ollessa päällä näkyvyysmääre korvataan automaattisesti halutunlaisella. Oletuksena korjausehdotus on rajoitettu, koska korjaus tehdään säännöllisten lausekkeiden avulla. On mahdollista, että säännöllinen lauseke ei toimi oikein esimerkiksi makrojen kanssa, joten virhetilanteiden välttämiseksi korjausehdotuksen tarjoaminen on rajoitettua. Automaattiseen korjaukseen liittyen on hyvä huomata, että automaattisen korjauksen saa päälle Clang-Tidyn komentoriviargumentin avulla [13], kun taas tarkistukseen lisätyn asetuksen avulla voidaan tarkemmin määrittää, millaisia korjauksia Clang-Tidy tekee automaattisesti, kun automaattiset korjaukset on otettu käyttöön.

Säännöllisten lausekkeiden käyttö on tässä tapauksessa tarpeen, koska AST:ihin liittyen ei näytä löytyvän toiminnallisuutta periytymisen näkyvyysmääreen käsittelyyn yksinkertaisemmin. Lähdekoodia käsitellään *Lexer*-luokan avulla [86; 87]. Säännölliset lausekkeet toimivat *Regex*-luokan avulla ja noudattavat POSIX ERE -syntaksia [88]. Luokat saadaan käyttöön sisällyttämällä dokumentaatiossa mainitut otsikkotiedostot tarkistuksen toteutuksen sisältävään tiedostoon.

Asetukseen *CreateReplacements* liittyvä koodi näkyy kuvissa 30 ja 31. Koodi perustuu Clang-Tidyn dokumentaatioon, jossa asetuksen lisääminen tarkistukselle selitetään [24].

```
clang-tools-extra > clang-tidy > bittium > C PreferPublicInheritanceCheck.h > ...
22  class PreferPublicInheritanceCheck : public ClangTidyCheck {
23  public:
24    PreferPublicInheritanceCheck(StringRef Name, ClangTidyContext *Context)
25        : ClangTidyCheck(Name, Context),
26          CreateReplacements(Options.get("CreateReplacements", false)) {}
27    void registerMatchers(ast_matchers::MatchFinder *Finder) override;
28    void check(const ast_matchers::MatchFinder::MatchResult &Result) override;
29    void storeOptions(ClangTidyOptions::OptionMap &Opts) override;
30  private:
31    const bool CreateReplacements;
32  };
```

Kuva 30. *CreateReplacements*-asetuksen koodi otsikkotiedostossa.

```
clang-tools-extra > clang-tidy > bittium > C+ PreferPublicInheritanceCheck.cpp > ...
80  void PreferPublicInheritanceCheck::storeOptions(ClangTidyOptions::OptionMap &Opts) {
81    Options.store(Opts, "CreateReplacements", CreateReplacements);
82  }
```

Kuva 31. *CreateReplacements*-asetuksen koodi toteutuksen sisältävässä tiedostossa.

Korjauksen jälkeen tarkistus toimii hieman eri tavalla kuin aiemmin. Automaattisen korjauksen toiminta riippuu tarkistuksen määrittelystä. Muutosten takia myös testejä pitää päivittää. Uusi testitiedosto näkyy kuvassa 32.

```
clang-tools-extra > test > clang-tidy > checkers > bittium > prefer-public-inheritance.cpp > ...
1   // RUN: %check_clang_tidy -check-suffix=DEFAULT %s bittium-prefer-public-inheritance %t
2   // RUN: %check_clang_tidy -check-suffix=REPLACEMENTS %s bittium-prefer-public-inheritance %t
    -config="{CheckOptions: {bittium-prefer-public-inheritance.CreateReplacements: true}}"
3
4   // CHECK-FIXES-DEFAULT: class Base1
5   // CHECK-FIXES-REPLACEMENTS: class Base1
6   class Base1
7   {
8   };
9
10  class Base2
11  {
12  };
13
14  // CHECK-FIXES-DEFAULT: class Derived1 : public Base1
15  // CHECK-FIXES-REPLACEMENTS: class Derived1 : public Base1
16  class Derived1 : public Base1
17  {
18  };
19
20  // CHECK-MESSAGES-DEFAULT: :[[@LINE+4]]:28: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
21  // CHECK-FIXES-DEFAULT: class Derived2 : protected Base1
22  // CHECK-MESSAGES-REPLACEMENTS: :[[@LINE+2]]:28: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
23  // CHECK-FIXES-REPLACEMENTS: class Derived2 : public Base1
24  class Derived2 : protected Base1
25  {
26  };
27
28  // CHECK-MESSAGES-DEFAULT: :[[@LINE+4]]:26: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
29  // CHECK-FIXES-DEFAULT: class Derived3 : private Base1
30  // CHECK-MESSAGES-REPLACEMENTS: :[[@LINE+2]]:26: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
31  // CHECK-FIXES-REPLACEMENTS: class Derived3 : public Base1
32  class Derived3 : private Base1
33  {
34  };
35
36  // CHECK-MESSAGES-DEFAULT: :[[@LINE+4]]:32: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
37  // CHECK-FIXES-DEFAULT: class Derived4 : public Base1, public Base2
38  // CHECK-MESSAGES-REPLACEMENTS: :[[@LINE+2]]:32: warning: avoid using non-public inheritance
    [bittium-prefer-public-inheritance]
39  // CHECK-FIXES-REPLACEMENTS: class Derived4 : public Base1, public Base2
40  class Derived4 : public Base1, Base2
41  {
42  };
43
```

Kuva 32. Päivitetty testitiedosto.

Uudistettu testitiedosto määrittelee 2 testiajoa. Näistä ensimmäinen tapahtuu oletusasetuksilla, kun taas toisessa otetaan käyttöön tarkistukselle lisätty toiminnaltaan kattavampi automaattinen korjaus. Testiajot nimetään argumentilla *-check-suffix*, jonka avulla määriteltyyn nimeen viitataan myös tulosteita ja korjauksia tarkistettaessa. Tämä mahdollistaa usean eri määrittelyn testaamisen samassa tiedostossa useina eri ajoina. [24.] Tarkistuksen toteutukseen liittyvien korjausten ja testien päivittämisen jälkeen voidaan taas koota projekti ja ajaa testit. Testien perusteella tarkistus näyttää toimivan halutulla tavalla.

4.4 Dokumentaation päivittäminen

Ohjelma add_new_check.py generoi uudelle tarkistukselle dokumentaation. Tähän sisältyy tarkistuksen otsikkotiedoston Doxygen-tyylinen luokan kuvaus, lisäykset julkaisutietoihin ja tarkistusten luetteloon sekä oma dokumentaatiosivu tarkistukselle. Generoitu dokumentaatio näkyy liitteessä 2.

Clang-Tidyn dokumentaation mukaan on suositeltavaa kuvata tarkistuksen toiminta tiiviisti yhdellä virkkeellä, joka kirjoitetaan julkaisutietoihin, tarkistuksen Doxygen-kommentin alkuun sekä tarkistuksen dokumentaatiosivun alkuun. Muuten dokumentaatio voi ja kannattaakin olla kattavampaa. [24.] Kuvissa 33, 34 ja 35 näkyy uuden tarkistuksen dokumentaatio. Dokumentaatio on kirjoitettu muita tarkistuksia apuna käyttäen.

```
clang-tools-extra > docs > ReleaseNotes.rst
117  New checks
118  ^^^^^^^^^^
119
120  - New :doc:`bittium-prefer-public-inheritance
121    <clang-tidy/checks/bittium/prefer-public-inheritance>` check.
122
123    Finds non-public inheritance.
124
```

Kuva 33. Lisäys julkaisutietoihin.

```
clang-tools-extra > clang-tidy > bittium > C PreferPublicInheritanceCheck.h > ...
17
18  /// Finds non-public inheritance.
19  ///
20  /// For the user-facing documentation see:
21  /// http://clang.llvm.org/extra/clang-tidy/checks/bittium/prefer-public-inheritance.html
22  class PreferPublicInheritanceCheck : public ClangTidyCheck {
```

Kuva 34. Tarkistuksen Doxygen-kommentti.

```
clang-tools-extra > docs > clang-tidy > checks > bittium > prefer-public-inheritance.rst
 1  .. title:: clang-tidy - bittium-prefer-public-inheritance
 2
 3  bittium-prefer-public-inheritance
 4  =================================
 5
 6  Finds non-public inheritance.
 7
 8  Works with classes, structs and unions.
 9  Works with default access specifiers, giving no warning if inheritance is
    implicitly public.
10  Works with multiple inheritance, giving warnings for all non-public
    inheritances.
11
12  Examples:
13
14  .. code-block:: c++
15
16    class Base1
17    {
18    };
19
20    class Base2
21    {
22    };
23
24    class Derived1 : public Base1 // No warning
25    {
26    };
27
28    class Derived2 : protected Base1 // Warning
29    {
30    };
31
32    class Derived3 : private Base1 // Warning
33    {
34    };
35
36    class Derived4 : public Base1, Base2 // 1 warning
37    {
38    };
39
40
41  Options
42  -------
43
44  .. option:: CreateReplacements
45
46      Boolean flag to replace `private` and `protected` with `public` when
        applying automatic fixes.
47      Automatic fixes work with regular expressions and could be prone to errors
        with macros, for example.
48      Default value is `false`.
49
```

Kuva 35. Tarkistuksen merkintäkielinen dokumentaatiosivu.

Dokumentaatio voidaan generoida HTML-muotoon jo aiemmin esitellyllä komennolla *cmake --build build/ --target docs-clang-tools-html*. Komento ei kuitenkaan vielä toimi, vaan päättyy virheeseen. Virheen mukaan mikään *toctree* ei sisällä tarkistuksen dokumentaatiotiedostoa. Tämä tarkoittaa, että tiedostoa ei ole lisätty dokumentaation sisällysluetteloon [89]. Ongelma saadaan korjattua käyttämällä muita moduuleja mallina. Kuvassa 36 näkyy moduulin dokumentaation lisääminen osaksi muuta dokumentaatiota.

clang-tools-extra > docs > clang-tidy > checks > list.rst

```
1   .. title:: clang-tidy - Clang-Tidy Checks
2
3   Clang-Tidy Checks
4   =================
5
6   .. toctree::
7      :glob:
8      :hidden:
9
10     abseil/*
11     altera/*
12     android/*
13     bittium/*
14     boost/*
```

Kuva 36. Moduulin sisällyttäminen dokumentaatioon.

Edellä kuvatun muutoksen jälkeen dokumentaation generoiminen onnistuu. Dokumentaatio on HTML-muodossa, joten lukeminen onnistuu selaimella. Kuvissa 37, 38 ja 39 näkyy, miltä tarkistukselle kirjoitettu dokumentaatio näyttää selaimessa.

New checks

- New **bittium-prefer-public-inheritance** check.

  Finds non-public inheritance.

Kuva 37. Julkaisutiedot selaimessa.

android-comparison-in-temp-failure-retry	
bittium-prefer-public-inheritance	Yes
boost-use-to-string	Yes
bugprone-argument-comment	Yes

Kuva 38. Clang-Tidy-tarkistuksia listattuna selaimessa.

Extra Clang Tools 17.0.3 documentation

CLANG-TIDY - BITTIUM-PREFER-PUBLIC-INHERITANCE

« android-comparison-in-temp-failure-retry :: Contents :: boost-use-to-string »

bittium-prefer-public-inheritance

Finds non-public inheritance.

Works with classes, structs and unions. Works with default access specifiers, giving no warning if inheritance is implicitly public. Works with multiple inheritance, giving warnings for all non-public inheritances.

Examples:

```
class Base1
{
};

class Base2
{
};

class Derived1 : public Base1 // No warning
{
};

class Derived2 : protected Base1 // Warning
{
};

class Derived3 : private Base1 // Warning
{
};

class Derived4 : public Base1, Base2 // 1 warning
{
};
```

Options

`CreateReplacements`

Boolean flag to replace *private* and *protected* with *public* when applying automatic fixes. Automatic fixes work with regular expressions and could be prone to errors with macros, for example. Default value is *false*.

Kuva 39. Dokumentaatiosivu selaimessa.

Dokumentaation kirjoittamisen myötä uusi tarkistus on valmis. LLVM ja Clang-Tidy voidaan koota release-tilassa ja lisätty tarkistus ottaa käyttöön oikeassa projektissa.

5 Lopputulos

Käytännön toteutus on valmis ja muutokset päivitetty versionhallintaan. Projekti on siis opinnäytetyön suhteen tullut päätökseen.

Projektin aikana saatiin hyvin asioita aikaiseksi, ratkottiin pulmia ja opittiin uutta. Opinnäytetyötä tehdessä muodostui monenlaisia päätelmiä työn aiheeseen liittyen. Jatkokehityksen kannalta syntyi monia ideoita.

5.1 Aikaansaannos

Opinnäytetyön aikana kehitettiin laajennettu versio Clang-Tidysta. Tämä versio sisältää uuden moduulin toimeksiantajalle ja yhden uuden tarkistuksen toimeksiantajan projektin ohjelmointikäytänteitä varten. Tarkistus on testattu toimivaksi automaatiotestien avulla ja dokumentoitu loppukäyttäjiä varten. LLVM:n lisenssin mukaisesti kaikkiin uusiin tai muunneltuihin tiedostoihin on kirjoitettu kommentit tehdyistä muutoksista.

Muunneltu työkalu on onnistuneesti lisätty osaksi toimeksiantajan projektin CI/CD-putkea. Opinnäytetyö on onnistunut, ja aikaansaannos on työlle asetettujen tavoitteiden mukainen. Toimeksiantaja on tyytyväinen opinnäytetyöhön.

5.2 Oppiminen

Opinnäytetyön aihe tarjosi paljon mahdollisuuksia oppimisen kannalta. Ohjelmistojen kokoaminen lähdekoodeista on hyödyllinen taito, minkä lisäksi voi tulla tarvetta muokata olemassa olevia projekteja omiin tarpeisiin sopivaksi. Muiden kirjoittaman koodin kehittäminen parantaa valmiuksia työskennellä ohjelmistoprojekteissa muiden kehittäjien kanssa. Kyky hallita laajoja ohjelmistokokonaisuuksia on tärkeä, jos tavoitteena on edetä pitkälle ohjelmistoalalla. Suunnitelmallinen ja järjestelmällinen työskentely sekä kyky hakea tietoa ja löytää ratkaisuja itsenäisesti ovat oleellisia taitoja alalla työskentelevälle.

Työ tarjosi mahdollisuuden tutustua monipuolisesti C++-kieleen: Uusien tarkistusten toteuttaminen vaatii ymmärrystä kielen toiminnasta. C++ on semanttisesti ja syntaktisesti monimutkainen

kieli. Uudet standardit muuttavat kieltä sekä lisäämällä että poistamalla ominaisuuksia. Moniparadigmaluonne mahdollistaa koodin kirjoittamisen usealla eri tavalla. Koontijärjestelmiä, kehitysympäristöjä, paketinhallintatyökaluja sekä muita kielen käyttöön liittyviä työkaluja on monenlaisia. Monikäyttöisenä kielenä C++:aan liittyvät ohjelmointikäytänteet vaihtelevat käyttötarkoituksen mukaan.

Työtä tehdessä tuli tutustuttua paremmin versionhallinnan toimintaan. Oli esimerkiksi tarpeen selvittää, miten forkattu projekti saadaan pidettyä ajan tasalla synkronoimalla aika ajoin muutokset alkuperäisen projektin kanssa Gitiä käyttäen. Esimerkiksi GitHubia käytettäessä tämä on yksinkertaista, mutta komentorivin kautta alkuperäisen projektin ollessa GitHubissa ja forkatun projektin ollessa GitLabissa tämä vaati aiheeseen tutustumista. Opinnäytetyödokumentin kirjoittamisessa taas SVN:stä on ollut apua. Työkalusta ei ole ennestään paljoa kokemusta, joten tämäkin on vaatinut uuden opettelua.

Työtä tehdessä on tullut tutustuttua moniin LLVM:n sisältämiin työkaluihin sekä projektiin yleisesti. Lisäksi on tullut käytettyä ennestään tuntemattomia komentoriviohjelmia. Erityisesti kootun ohjelmiston asentamiseen liittyen myös Linuxin Filesystem Hierarchy Standard on tullut tutummaksi.

5.3 Päätelmät

Opinnäytetyön aikana tuli selväksi, että LLVM on joiltain osin hyvin heikosti dokumentoitu. Esimerkiksi käytössä olevaa säännöllisten lausekkeiden syntaksia ei ole mainittu työkalujen ohjeessa. Myös uuden moduulin lisäämiseen liittyvä dokumentaatio on puutteellista. Heikkotasoisen dokumentaation vuoksi oli luettava paljon muiden kirjoittamaa koodia selvittäessä, miten asiat toimivat. Virheilmoitukset ja yleinen alaan liittyvä osaaminen auttoivat merkittävästi ongelmien ratkomisessa. Koska LLVM on suosittu projekti, kolmansilta osapuolilta löytyy hyödyllistä tietoa esimerkiksi blogikirjoitusten ja foorumikeskustelujen muodossa. Näitä lähteitä kannattaa hyödyntää, kunhan esitetyille väitteille löytyy hyvät perustelut. Avoimen lähdekoodin projekteihin liittyen tiedon oikeellisuus voidaan usein varmistaa melko helposti.

Työ toimii hyvänä esimerkkinä ohjelmistotestaamisen tärkeydestä. Työn aikana uuden tarkistuksen ensimmäinen toteutus oli automaattisten korjausten osalta virheellinen, mikä paljastui automaatiotestauksen avulla. Ongelman paljastuminen varhaisessa vaiheessa mahdollisti vian korjaamisen ennen työkalun käyttöönottoa oikeassa projektissa.

Työn perusteella voidaan sanoa, että Clang-Tidyn laajentaminen vaatii melko paljon työtä. Uuden moduulin luominen Clang-Tidyn dokumentaation perusteella on haasteellista dokumentaatioon liittyvien puutteiden takia. Tarkistuksen lisääminen olemassa olevaan moduuliin on kuitenkin yksinkertaisempaa LLVM-projektin sisältämää apuohjelmaa käyttäen. Ensimmäisen tarkistuksen toteuttaminen vaatii perehtymistä aiheeseen, koska dokumentaatio on pirstaleista, eikä kaikkea oleellista ole kerrottu dokumentaatiossa. Opinnäytetyön esimerkkiä seuraten yksinkertaisen tarkistuksen lisääminen pitäisi onnistua hieman helpommin kuin pelkästään LLVM:n dokumentaation avulla. Kun uuden tarkistuksen lisäämiseen liittyvän prosessin on käynyt kertaalleen läpi, pitäisi olla mahdollista toteuttaa lisää tarkistuksia vaativuustasoa vähitellen lisäten.

LLVM:n kokoaminen on tietokoneelle vaativa tehtävä. Prosessorin ja muistin käyttöaste on korkea, ja erityisesti debug-koonti vaatii paljon tallennustilaa. Virtuaalikonetta käytettäessä on tarpeen varata riittävästi resursseja koneen käyttöön. Kannattaa koota LLVM-projektista vain ne osat, joita tarvitsee. Varsinkaan kehityksen aikana debug-koontia käyttäessä ei ole tarpeen asentaa koottua ohjelmistoa järjestelmälaajuisesti, vaan ohjelmia voi käyttää suoraan build-kansiosta. Jos tarkoituksena on asentaa yksittäinen työkalu konttiin, voidaan käyttövalmis release-koonti jättää asentamatta järjestelmään ja kopioida tarvittavat koonnin jälkeiset tiedostot konttiin.

Asennettaessa LLVM järjestelmälaajuisesti voi olla hyvä harkita asennuksen tekemistä johonkin muualle kuin oletussijaintiin. Opinnäytetyötä tehdessä tallennustilan loppuminen muodostui ongelmaksi debug-koonnin asennuksessa. Tapaa asennuksen poistamiseen CMaken avulla ei löytynyt. CMaken pitäisi osata generoida tekstitiedosto, joka sisältää listan asennetuista tiedostoista. Nämä tiedostot poistamalla asennus saadaan käytännössä poistettua. Tallennustilan loppumisen vuoksi asennettuja tiedostoja listaavaa tiedostoa ei luotu. Asennuksen aikainen tuloste kuitenkin listasi asennetut tiedostot, joten tämän tulosteen kopioiminen tekstitiedostoon ja tiedoston käsitteleminen sedin, xargsin ja rm:n avulla mahdollisti epäonnistuneen asennuksen poistamisen melko vähäisellä manuaalisella työllä. Asennuksen poistaminen olisi ollut huomattavasti nopeampaa ja helpompaa poistamalla asennuksen hakemisto. Oletuksena tämä oli kuitenkin */usr/local*, jonka alihakemistoihin asennettavat tiedostot kopioitiin. Asennettuja tiedostoja oli siis useassa eri hakemistossa, joita ei olisi järkevä poistaa. Tämän vuoksi asennus voi olla hyvä tehdä johonkin muualle kuin oletussijaintiin, koska sopivan hakemiston valitsemalla koko asennuksen voi poistaa turvallisesti yhden kansion poistamalla. Kannattaa edelleen myös harkita, onko järjestelmälaajuisen asennuksen tekeminen tarpeen.

Clang-Tidyn tarkistuksiin liittyvä dokumentaatio kirjoitetaan reStructuredText-kielellä. Tämä on LLVM:n puolesta hyvä valinta, koska kyseisellä merkintäkielellä kirjoitettuihin tiedostoihin on

mahdollista lisätä kommentteja, joita ei renderöidä lukuohjelman avulla, ja jotka eivät tule osaksi generoitavaa HTML-dokumentaatiota. LLVM:n lisenssi edellyttää projektiin tehtyjen muutosten dokumentoimista selkeästi, joten reST sopii dokumentaation kirjoittamiseen hyvin, koska muutokset tiedostoihin ja uudet tiedostot on helppo merkitä kommenttien avulla.

Opinnäytetyö sisältää ohjeen Clang-Tidyn käyttöön CMake-pohjaisessa C++-projektissa. Ohjeen avulla Clang-Tidy pitäisi olla helppo ottaa käyttöön esimerkiksi kouluprojektissa. Työkalun voi esimerkiksi asentaa Docker-konttiin ja ottaa käyttöön projektin CI/CD-putkessa. Työkalun laajentaminen kouluprojektia varten ei kuitenkaan välttämättä ole järkevää uuden moduulin ja uusien tarkistusten lisäämiseen liittyvän työn määrän takia. Myös koontiin kuluva aika ja tallennustilaan liittyvät vaatimukset lisäävät haasteita muokata Clang-Tidya kouluprojektissa käytettäväksi. Kouluprojektissa onkin järkevää käyttää Clang-Tidya järjestelmän paketinhallinnan kautta asennettuna.

Opinnäytetyötä tehdessä erityisen haastavaa on ollut kuvien lisääminen työhön. Kuvat sisältävät paljoa tekstiä, ja tekstin tulee olla riittävän kokoista luettavaksi. Komentoriviympäristössä komennot on usein mahdollista jakaa monelle riville. Ikkunan kokoa säätämällä on mahdollista saada tuloste leveydeltään sopivaksi. Koodieditoreja käytettäessä taas on usein mahdollista rajoittaa rivin pituutta siten, että rivi näyttää editorissa jakautuvan usealle riville, mutta rivinumerointi paljastaa kyseessä olevan yksi rivi tiedostossa. Kuvia on myös mahdollista rajata, ja asian esittämiseen voi käyttää useampaakin kuvaa. Kuvankäsittelyn avulla on myös mahdollista esittää komentosarjoja ilman, että näitä komentoja todellisuudessa suoritetaan. Tämä onnistuu monessa komentoriviympäristössä näppäinyhdistelmällä *Ctrl + C*, joka voi tulostaa rivin loppuun esimerkiksi merkkijonon *^C*. Tämä ei-haluttu merkkijono voidaan poistaa kuvankäsittelyohjelman avulla.

Haastavaa on ollut myös huolehtia opinnäytetyön saavutettavuudesta. Työssä esiintyvät kuvat sisältävät paljon tekstiä ja ovat työn sisällön ymmärtämisen kannalta oleellisia. Kaikki tieto ei käy ilmi opinnäytetyön sanallisesta kuvauksesta, joten näkörajoitteisten ihmisten voi olla vaikea lukea työtä. Kuvilla ei ole tekstivastinetta, koska hyvän tekstivastineen kirjoittaminen olisi ollut hyvin haastavaa tai jopa mahdotonta. Kuvat sisältävät paljon tekstiä, johon myös sisältyy paljon erikoismerkkejä. Sisällön asettelu voi vaikuttaa tiedostojen toimintaan, joten asettelu pitäisi pystyä kuvaamaan virheettömästi ja selkeästi. Englanninkielinen sisältö yhdessä suomenkielisen kuvauksen kanssa ei välttämättä toimi näytönlukijalla kovin hyvin. Näistä syistä hyvien tekstivastineiden kirjoittaminen ei ole onnistunut. Huonolaatuiset tekstivastineet todennäköisesti haittaisivat sisällön ymmärtämistä enemmän kuin näistä olisi apua, joten tekstivastineet on jätetty pois. Näkörajoitteisten ihmisten on mahdollisesti pyydettävä apua opinnäytetyön lukemiseen muilta. Tämä on

saavutettavuuden kannalta ongelma, johon tulisi löytää ratkaisu. Saavutettavuuteen on tullut opinnäytetyötä tehdessä tutustuttua, mutta aihe on kuitenkin laaja ja haastava. Saavutettavuutta ei ole käsitelty opetuksessa, joten opinnäytetyötä tehdessä syvällisempi perehtyminen asiaan vaatisi kohtuuttoman paljon työtä. Tämän takia aihe olisi hyvä lisätä osaksi opintoja. Yhdenvertaisuuslain 2 luvun 6 § mukaan koulutuksen järjestäjällä on velvollisuus edistää yhdenvertaisuutta [90], joten tästäkin syystä koulun olisi syytä ottaa asia huomioon opetuksessa.

5.4 Jatkokehitys

Aikaan saatua ohjelmistoa voisi parannella kehittämällä lisättyä tarkistusta eteenpäin. Ei-julkinen periytyminen voi joissain tilanteissa olla perusteltu ratkaisu, joten tarkistukselle voisi lisätä asetuksen tämän sallimiseksi määritellyille luokille. Asetuksen lisäämisen jälkeen testejä pitäisi taas päivittää. Tarkistuksen tämänhetkisen toteutuksen testikattavuutta voisi parantaa esimerkiksi testaamalla, kuinka tarkistus toimii makrojen ja luokkamallien kanssa ja Windowsissa [24].

Opinnäytetyön alussa syntyneen suunnitteludokumentin avulla toimeksiantajan olisi mahdollista luoda projektin käytänteitä vastaavat määrittelytiedostot työkaluille, joita tehtävän määrittelyn yhteydessä tarkasteltiin. Suunnitteludokumentin avulla olisi myös mahdollista laatia muistilista koodin katselmointia varten. Lisäksi dokumentin perusteella voisi lähteä toteuttamaan lisää tarkistuksia toimeksiantajan tarpeiden mukaan. Toimeksiantajaa varten luodun moduulin ja opinnäytetyön esimerkin pohjalta uusien tarkistusten lisääminen pitäisi onnistua melko vaivattomasti.

Jatkokehityksen kannalta voisi olla aiheellista selvittää, miten työkalun kehittäminen yhdessä muiden ESSOR-ohjelmassa mukana olevien tahojen kanssa onnistuisi. Voisi myös olla hyvä selvittää, kannattaisiko muunneltu ohjelmisto julkaista avoimena lähdekoodina tai suoritettavassa muodossa. Muutosten lisääminen osaksi LLVM-projektia voisi esimerkiksi tuoda yritykselle näkyvyyttä.

Jatkokehityksen helpottamiseksi voisi olla hyvä ajaa LLVM:n testit CI/CD-putkessa. Opinnäytetöytä tehdessä testit on ajettu paikallisesti. Voisi myös olla hyvä tarkistaa koodi Clang-Tidyn ja ClangFormatin avulla, jotta LLVM:n ohjelmointikäytänteet toteutuvat. Ohjelmointikäytänteiden noudattamiseen on pyritty opinnäytetyön aikana mahdollisimman hyvin, mutta asiaa ei ole tarkastettu automaatiotyökaluilla.

clang-query sisältää asetuksen *srcloc*, jonka avulla voidaan tulostaa AST-solmujen kiinnostavia kohtia, joihin päästään käsiksi eri metodien avulla [91]. Tämä ominaisuus voi auttaa uusien tarkistusten kehittämisessä. Opinnäytetyötä tehdessä ominaisuus ei kuitenkaan toiminut: Kehityksen aikana debug-koottu ohjelma ilmoitti, ettei ominaisuus ole saatavilla. Valmiin release-koonnin ohjelmasta ominaisuus löytyy, mutta tämä toimii vain joillekin hauille, kun taas toisten hakujen kanssa ohjelma kaatuu asetusta käytettäessä. Pikaisella virheilmoitusten ja lähdekoodien tutkimisella ongelman syy ei selvinnyt, mutta tarkemmalla selvittelyllä asia voisi ratketa. On mahdollista, että ominaisuus on vielä kehitteillä ja että ongelma on kehittäjien tiedossa. On myös mahdollista, että kyseessä on bugi, jolloin asiasta olisi hyvä tehdä raportti. Myös käyttäjästä johtuva virhe on mahdollinen, mutta tällöinkään ohjelman ei todennäköisesti olisi tarkoitus kaatua. Ongelma on kuvattu tarkemmin liitteessä 3, jonka avulla ongelman voi yrittää toisintaa.

6 Yhteenveto

Opinnäytetyön alussa käytiin läpi teoriaa ohjelmointikäytänteisiin liittyen. Olemassa olevia ratkaisuja automaattiseen ja manuaaliseen koodin analysoimiseen esiteltiin. Lisäksi kuvattiin toimeksiantajan tarpeet opinnäytetyön aiheeseen liittyen.

Ennen varsinaista käytännön toteutusta tarkasteltiin hieman tarkemmin Clang-Tidya. Työkalun valinnalle esitettiin perusteet ja työkalun käyttöönotto esiteltiin yksinkertaisen projektin avulla. Lisäksi tarkasteltiin Clang-Tidyn määrittelytiedostoja ja käyttöä CI/CD-putkessa.

Projektin toteutus aloitettiin LLVM:n ja Clang-Tidyn kokoamisella lähdekoodeista. Tämän jälkeen lisättiin uusi moduuli ja tarkistus työkaluun. Tarkistukselle myös kirjoitettiin automaatiotestit ja dokumentaatio. Testien avulla paljastui tarkistuksen toteutukseen liittyvä ongelma, joka korjattiin.

Työ onnistui hyvin. Työn aikana opittiin paljon uusia asioita, ratkottiin ongelmia ja pohdittiin, mitä päätelmiä työn pohjalta voidaan tehdä. Työstä oli hyötyä tekijälleen ja toimeksiantajalle. Työstä saattaa olla apua myös muille opiskelijoille sekä opettajille. Lisäksi muut ohjelmistokehittäjät voivat pystyä hyödyntämään opinnäytetyötä projekteissaan.

Lähteet

1. Kiiskilä Eetu. EditorConfig ohjelmistokehityksen apuvälineenä. [AMK-kurssityö]. Kajaanin ammattikorkeakoulu; 2023. Ei julkisesti saatavilla.
2. Coding Standards For Quality and Compliance. Perforce Software, Inc. [Internet]. [viitattu 11.10.2023]. Saatavilla: https://www.perforce.com/resources/qac/coding-standards
3. Stroustrup Bjarne. Bjarne Stroustrup's C++ Style and Technique FAQ. [Internet]. [viitattu 11.10.2023]. Saatavilla: https://www.stroustrup.com/bs_faq2.html
4. Coding Standards. Standard C++ Foundation. [Internet]. [viitattu 11.10.2023]. Saatavilla: https://isocpp.org/wiki/faq/coding-standards
5. P2295R6. Support for UTF-8 as a portable source file encoding. ISO/IEC JTC1/SC22/WG21. 2022. Saatavilla: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2295r6.pdf
6. Van Rossum Guido, Warsaw Barry, Coghlan Nick. PEP 8 – Style Guide for Python Code. Python Software Foundation. [Internet]. [viitattu 11.10.2023]. Saatavilla: https://peps.python.org/pep-0008/
7. Warsaw Barry, Hylton Jeremy, Goodger David, Coghlan Alyssa. PEP 1 – PEP Purpose and Guidelines. Python Software Foundation. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://peps.python.org/pep-0001/
8. Peters Tim. PEP 20 – The Zen of Python. Python Software Foundation. [Internet]. [viitattu 23.10.2023]. Saatavilla: https://peps.python.org/pep-0020/
9. EditorConfig. EditorConfig Team. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://editorconfig.org/
10. EditorConfig Specification. EditorConfig Team. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://spec.editorconfig.org/
11. C++ compiler support. CPP Reference. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://en.cppreference.com/w/cpp/compiler_support

12. Parr Kealan. Unicode Characters – What Every Developer Should Know About Encoding. freeCodeCamp. [Internet]. 1.3.2021 [viitattu 12.10.2023]. Saatavilla: https://www.freecodecamp.org/news/everything-you-need-to-know-about-encoding/

13. Extra Clang Tools 18.0.0git documentation > Clang-Tidy. The Clang Team. [Internet]. [viitattu 21.11.2023]. Saatavilla: https://clang.llvm.org/extra/clang-tidy/

14. Extra Clang Tools 18.0.0git documentation > Clang-Tidy > Clang-Tidy Checks. The Clang Team. [Internet]. [viitattu 31.10.2023]. Saatavilla: https://clang.llvm.org/extra/clang-tidy/checks/list.html

15. Extra Clang Tools 18.0.0git documentation > Clang-Tidy > Clang-tidy IDE/Editor Integrations. The Clang Team. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://clang.llvm.org/extra/clang-tidy/Integrations.html

16. Clang 18.0.0git documentation > Using Clang as a Compiler > Clang Static Analyzer. The Clang Team. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://clang.llvm.org/docs/ClangStaticAnalyzer.html

17. Clang 18.0.0git documentation > Using Clang as a Complier > Clang Static Analyzer > Available Checkers. The Clang Team. [Internet]. [viitattu 17.10.2023]. Saatavilla: https://clang.llvm.org/docs/analyzer/checkers.html

18. Clang 18.0.0git documentation > Using Clang Tools > ClangFormat. The Clang Team. [Internet]. [viitattu 16.10.2023]. Saatavilla: https://clang.llvm.org/docs/ClangFormat.html

19. GCC online documentation > GCC 13.2 Manual > GCC Command Options > Options to Request or Suppress Warnings. Free Software Foundation, Inc. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/Warning-Options.html

20. Clang 18.0.0git documentation > Using Clang as a Compiler > Diagnostic flags in Clang. The Clang Team. [Internet]. [viitattu 12.10.2023]. Saatavilla: https://clang.llvm.org/docs/DiagnosticsReference.html

21. Martins Hugo [@caramelomartins] ym. Awesome Linters [awesome-linters]. [GitHub-säilö]. GitHub. [viitattu 12.10.2023]. Saatavilla: https://github.com/caramelomartins/awesome-linters

22. Analysis Tools [@analysis-tools-dev] ym. [static-analysis]. [GitHub-säilö]. GitHub. [viitattu 16.10.2023]. Saatavilla: https://github.com/analysis-tools-dev/static-analysis

23. What Is Lint Code? And What Is Linting and Why Is Linting Important? Perforce Software, Inc. [Internet]. 19.3.2019 [viitattu 16.10.2023]. Saatavilla: https://www.perforce.com/blog/qac/what-lint-code-and-what-linting-and-why-linting-important

24. Extra Clang Tools 18.0.0git documentation > Clang-Tidy > Getting Involved. The Clang Team. [Internet]. [viitattu 24.11.2023]. Saatavilla: https://clang.llvm.org/extra/clang-tidy/Contributing.html

25. GCC online documentation > GCC Internals Manual > User Experience Guidelines > Guidelines for Diagnostics. Free Software Foundation, Inc. [Internet]. [viitattu 16.10.2023]. Saatavilla: https://gcc.gnu.org/onlinedocs/gccint/Guidelines-for-Diagnostics.html

26. Verbina Evgenia. Best Code Review Tools for 2023 – Survey Results. JetBrains s.r.o. [Internet]. 15.12.2021. [viitattu 23.10.2023]. Saatavilla: https://blog.jetbrains.com/space/2021/12/15/best-code-review-tools/

27. Lockheed Martin Corporation. Uudistettu versio C. JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM. 2005. Saatavilla: https://www.stroustrup.com/JSF-AV-rules.pdf

28. Savolainen Sami. ESSOR - The Way Towards Communications Interoperability Amongst European Armed Forces. Bittium. [Internet]. 21.10.2021 [viitattu 19.10.2023]. Saatavilla: https://www.bittium.com/about-bittium/blog/essor-communications-interoperability-amongst-european-armed-forces

29. a4ESSOR SAS, Bittium Wireless Ltd, Indra Sistemas S.A., Leonardo S.p.A., RADMOR S.A., Rohde & Schwarz GmbH & Co. KG, Thales SIX GTS France S.A.S. Uudistettu versio AA. SOFTWARE RULES AND METHODOLOGIES FOR WAVEFORM DEVELOPMENT. 2022. Ei julkisesti saatavilla.

30. Kallio Sander. Avoin lähdekoodi: kulttuuri ja toimintamallit. [AMK-opinnäytetyö]. Tampereen ammattikorkeakoulu; 2020. Saatavilla: https://urn.fi/URN:NBN:fi:amk-2020060115859

31. LLVM [@llvm]. LLVM Project [llvm-project]. [GitHub-säilö]. GitHub. [viitattu 2.11.2023]. Saatavilla: https://github.com/llvm/llvm-project

32. Apple Open Source. Apple Inc. [Internet]. [viitattu 13.10.2023]. Saatavilla: https://opensource.apple.com/

33. Apple Open Source > Projects. Apple Inc. [Internet]. [viitattu 13.10.2023]. Saatavilla: https://opensource.apple.com/projects/

34. The LLVM Compiler Infrastructure > LLVM Users. LLVM. [Internet]. [viitattu 13.10.2023]. Saatavilla: https://llvm.org/Users.html

35. The LLVM Compiler Infrastructure > Projects built with LLVM. LLVM. [Internet]. [viitattu 13.10.2023]. Saatavilla: https://llvm.org/ProjectsWithLLVM/

36. LLVM [@llvm] ym. LLVM Project [llvm-project] > LICENSE.TXT. [Tiedosto]. GitHub. [viitattu 13.10.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/LICENSE.TXT

37. GitHub, Inc. ym. Choose a License > Licenses > Apache License 2.0. GitHub, Inc. [Internet]. [viitattu 13.10.2023]. Saatavilla: https://choosealicense.com/licenses/apache-2.0/

38. KDE [@KDE] ym. Clazy [clazy]. [GitHub-säilö]. GitHub. [viitattu 16.10.2023]. Saatavilla: https://github.com/KDE/clazy

39. Marjamäki Daniel [@danielmarjamaki]. Cppcheck [cppcheck] > Wiki > ListOfChecks. [Wikisivu]. SourceForge. [viitattu 16.10.2023]. Saatavilla: https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/

40. Marjamäki Daniel [@danmar] ym. Cppcheck [cppcheck] > COPYING. [Tiedosto]. GitHub. [viitattu 16.10.2023]. Saatavilla: https://github.com/danmar/cppcheck/blob/main/COPYING

41. GitHub, Inc. ym. Choose a License > Licenses > GNU General Public License v3.0. GitHub, Inc. [Internet]. [viitattu 16.10.2023]. Saatavilla: https://choosealicense.com/licenses/gpl-3.0/

42. Google [@google] ym. Google Style Guides [styleguide]. [GitHub-säilö]. GitHub. [viitattu 16.10.2023]. Saatavilla: https://github.com/google/styleguide

43. Google [@google]. Google Style Guides [styleguide] > cpplint > History. [GitHub-säilö]. GitHub. [viitattu 16.10.2023]. Saatavilla: https://github.com/google/styleguide/commits/gh-pages/cpplint

44. [@cpplint]. [cpplint]. [GitHub-säilö]. GitHub. [viitattu 16.10.2023]. Saatavilla: https://github.com/cpplint/cpplint

45. clang-format regex syntax reference. [Foorumikeskustelu]. Stack Overflow. [viitattu 17.10.2023]. Saatavilla: https://stackoverflow.com/questions/39986368/clang-format-regex-syntax-reference

46. LLVM Project [llvm-project] > Issues > clang-tidy header filter fails with regex. [Foorumikeskustelu]. GitHub. [viitattu 17.10.2023]. Saatavilla: https://github.com/llvm/llvm-project/issues/25590

47. Funk Kevin. Clang-Tidy, part 1: Modernize your source code using C++11/C++14. KDAB. [Internet]. 16.3.2017 [viitattu 17.10.2023]. Saatavilla: https://www.kdab.com/clang-tidy-part-1-modernize-source-code-using-c11c14/

48. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang-tools-extra > clang-tidy > tool > run-clang-tidy.py. [Tiedosto]. GitHub. [viitattu 17.10.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang-tools-extra/clang-tidy/tool/run-clang-tidy.py

49. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang-tools-extra > clang-tidy > tool > clang-tidy-diff.py. [Tiedosto]. GitHub. [viitattu 17.10.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang-tools-extra/clang-tidy/tool/clang-tidy-diff.py

50. LLVM. run-clang-tidy [Ubuntu LLVM version 14.0.0, Optimized build, Default target: x86_64-pc-linux-gnu, Host CPU: alderlake]. [Tietokoneohjelmisto]. LLVM Packaging Team. [viitattu 18.10.2023]. Saatavilla Ubuntu-paketinhallinnasta nimellä clang-tidy, versionumerolla 1:14.0-55~exp2.

51. Extra Clang Tools 18.0.0git documentation > Clang-Tidy > Clang-Tidy Checks > cpp-coreguidelines-init-variables. The Clang Team. [Internet]. [viitattu 17.10.2023]. Saatavilla: https://clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines/init-variables.html

52. Extra Clang Tools 18.0.0git documentation > Clang-Tidy > Clang-Tidy Checks > clang-analyzer-cplusplus.NewDeleteLeaks. The Clang Team. [Internet]. [viitattu 17.10.2023]. Saatavilla: https://clang.llvm.org/extra/clang-tidy/checks/clang-analyzer/cplusplus.NewDeleteLeaks.html

53. Extra Clang Tools 18.0.0git documentation > Clang-Tidy > Clang-Tidy Checks > readability-identifier-naming. The Clang Team. [Internet]. [viitattu 18.10.2023]. Saatavilla: https://clang.llvm.org/extra/clang-tidy/checks/readability/identifier-naming.html

54. LLVM. clang-tidy [Ubuntu LLVM version 14.0.0, Optimized Build, Default target: x86_64-pc-linux-gnu, Host CPU: alderlake]. [Tietokoneohjelmisto]. LLVM Packaging Team. [viitattu 18.10.2023]. Saatavilla Ubuntu-paketinhallinnasta nimellä clang-tidy, versionumerolla 1:14.0-55~exp2.

55. [RFC] Deprecate old key-value format in CheckOptions. [Foorumikeskustelu]. LLVM Discourse. [viitattu 18.10.2023]. Saatavilla: https://discourse.llvm.org/t/rfc-deprecate-old-key-value-format-in-checkoptions/72233

56. LLVM 18.0.0git documentation > Getting Started/Tutorials > Getting Started with the LLVM System. LLVM Project. [Internet]. [viitattu 30.10.2023]. Saatavilla: https://llvm.org/docs/GettingStarted.html

57. LLVM 18.0.0git documentation > User Guides > LLVM Builds and Distributions > Building LLVM with CMake. LLVM Project. [Internet]. [viitattu 27.10.2023]. Saatavilla: https://llvm.org/docs/CMake.html

58. Kitware, Inc. ym. CMake 3.27 Documentation > cmake(1). Kitware, Inc. [Internet]. [viitattu 27.10.2023]. Saatavilla: https://cmake.org/cmake/help/v3.27/manual/cmake.1.html

59. LLVM 18.0.0git documentation > Reference > LLVM Testing Infrastructure Guide. LLVM Project. [Internet]. [viitattu 30.10.2023]. Saatavilla: https://llvm.org/docs/TestingGuide.html

60. [clang-tidy] Add a module for the Linux kernel. [Foorumikeskustelu]. LLVM Phabricator. [viitattu 1.11.2023]. Saatavilla: https://reviews.llvm.org/D59963

61. Why are my custom clang-tidy checks not showing up after adding custom module? [Foorumikeskustelu]. Stack Overflow. [viitattu 1.11.2023]. Saatavilla: https://stackoverflow.com/questions/73531240/why-are-my-custom-clang-tidy-checks-not-showing-up-after-adding-custom-module

62. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang-tools-extra > clang-tidy > CMakeLists.txt. [Tiedosto]. GitHub. [viitattu 1.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang-tools-extra/clang-tidy/CMakeLists.txt

63. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang-tools-extra > clang-tidy > tool > CMakeLists.txt. [Tiedosto]. GitHub. [viitattu 1.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang-tools-extra/clang-tidy/tool/CMakeLists.txt

64. C++ Core Guidelines [CppCoreGuidelines] > Issues > Q: How to express that something is accessed for its side effect in C++? [Foorumikeskustelu]. GitHub. [viitattu 2.11.2023]. Saatavilla: https://github.com/isocpp/CppCoreGuidelines/issues/1048

65. Is it allowed for a compiler to optimize away a local volatile variable? [Foorumikeskustelu]. Stack Overflow. [viitattu 2.11.2023]. Saatavilla: https://stackoverflow.com/questions/51472394/is-it-allowed-for-a-compiler-to-optimize-away-a-local-volatile-variable

66. N4878. Working Draft, Standard for Programming Language C++. ISO/IEC JTC1/SC22/WG21. 2020. Saatavilla: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4878.pdf

67. LLVM [@llvm] ym. LLVM Project [llvm-project] > llvm > include > llvm > Support > Compiler.h. [Tiedosto]. GitHub. [viitattu 2.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/Support/Compiler.h

68. Clang 18.0.0git documentation > Using Clang as a Library > Tutorial for building tools using LibTooling and LibASTMatchers. The Clang Team. [Internet]. [viitattu 2.11.2023]. Saatavilla: https://clang.llvm.org/docs/LibASTMatchersTutorial.html

69. Clang 18.0.0git documentation > Using Clang as a Library > LibTooling. The Clang Team. [Internet]. [viitattu 2.11.2023]. Saatavilla: https://clang.llvm.org/docs/LibTooling.html

70. Clang 18.0.0git documentation > Design Documents > "Clang" CFE Internals Manual. The Clang Team. [Internet]. [viitattu 3.11.2023]. Saatavilla: https://clang.llvm.org/docs/InternalsManual.html

71. clang-tools 18.0.0git > Classes > Class List > clang > tidy > ClangTidyModule. LLVM. [Internet]. [viitattu 3.11.2023]. Saatavilla: https://clang.llvm.org/extra/doxygen/classclang_1_1tidy_1_1ClangTidyModule.html

72. LLVM Project [llvm-project] > Issues > add_new_check.py needs refactoring. [Foorumikeskustelu]. GitHub. [viitattu 3.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/issues/59606

73. Clang 18.0.0git documentation > Using Clang Tools > ClangCheck. The Clang Team. [Internet]. [viitattu 6.11.2023]. Saatavilla: https://clang.llvm.org/docs/ClangCheck.html

74. Clang 18.0.0git documentation > Using Clang as a Library > How To Setup Clang Tooling For LLVM. The Clang Team. [Internet]. [viitattu 6.11.2023]. Saatavilla: https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html

75. LLVM. ClangCheck [LLVM version 17.0.3, DEBUG build with assertions]. [Tietokoneohjelmisto]. LLVM. [viitattu 6.11.2023]. Saatavilla lähdekoodeista kokoamalla: https://github.com/llvm/llvm-project

76. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang-tools-extra > clang-query > tool > ClangQuery.cpp. [Tiedosto]. GitHub. [viitattu 7.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang-tools-extra/clang-query/tool/ClangQuery.cpp

77. Kelly Stephen. Exploring Clang Tooling Part 2: Examining the Clang AST with clang-query. Microsoft. [Internet]. 23.10.2018 [viitattu 7.11.2023]. Saatavilla: https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-2-examining-the-clang-ast-with-clang-query/

78. Nixpkgs [nixpkgs] > Issues > Add libedit to clang-tools-extra. [Foorumikeskustelu]. GitHub. [viitattu 7.11.2023]. Saatavilla: https://github.com/NixOS/nixpkgs/issues/85217

79. Clang 18.0.0git documentation > AST Matcher Reference. The Clang Team. [Internet]. [viitattu 9.11.2023]. Saatavilla: https://clang.llvm.org/docs/LibASTMatchersReference.html

80. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang > include > clang > ASTMatchers > ASTMatchFinder.h. [Tiedosto]. GitHub. [viitattu 8.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang/include/clang/ASTMatchers/ASTMatchFinder.h

81. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang-tools-extra > clang-tidy > ClangTidyCheck.h. [Tiedosto]. GitHub. [viitattu 9.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang-tools-extra/clang-tidy/ClangTidyCheck.h

82. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang > include > clang > ASTMatchers > ASTMatchers.h. [Tiedosto]. GitHub. [viitattu 8.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang/include/clang/ASTMatchers/ASTMatchers.h

83. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang > include > clang > ASTMatchers > ASTMatchersInternal.h. [Tiedosto]. GitHub. [viitattu 8.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang/include/clang/ASTMatchers/ASTMatchersInternal.h

84. clang 18.0.0git > Classes > Class List > clang > CXXRecordDecl. LLVM. [Internet]. [viitattu 9.11.2023]. Saatavilla: https://clang.llvm.org/doxygen/classclang_1_1CXXRecordDecl.html

85. LLVM [@llvm] ym. LLVM Project [llvm-project] > clang > include > clang > AST > DeclCXX.h. [Tiedosto]. GitHub. [viitattu 9.11.2023]. Saatavilla: https://github.com/llvm/llvm-project/blob/main/clang/include/clang/AST/DeclCXX.h

86. Kreuzkamp Anton, Funk Kevin. Can't the Compiler Do That? KDAB. [Internet]. 12.5.2021 [viitattu 21.11.2023]. Saatavilla: https://www.kdab.com/cpp-with-clang-libtooling/

87. clang 18.0.0git > Classes > Class List > clang > Lexer. LLVM. [Internet]. [viitattu 21.11.2023]. Saatavilla: https://clang.llvm.org/doxygen/classclang_1_1Lexer.html

88. LLVM 18.0.0git > Classes > Class List > llvm > Regex. LLVM. [Internet]. [viitattu 21.11.2023]. Saatavilla: https://llvm.org/doxygen/classllvm_1_1Regex.html

89. Sphinx documentation > Using Sphinx > reStructuredText > Directives. The Sphinx Developers. [Internet]. [viitattu 21.11.2023]. Saatavilla: https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html

90. L 30.12.2014/1325. Yhdenvertaisuuslaki. 2014. Saatavilla: https://www.finlex.fi/fi/laki/ajantasa/2014/20141325

91. Kelly Stephen. Location, Location, Location. [Internet]. 27.4.2021 [viitattu 30.11.2023]. Saatavilla: https://steveire.wordpress.com/2021/04/27/location-location-location/

Esimerkkejä hyödyttömistä koodilauseista

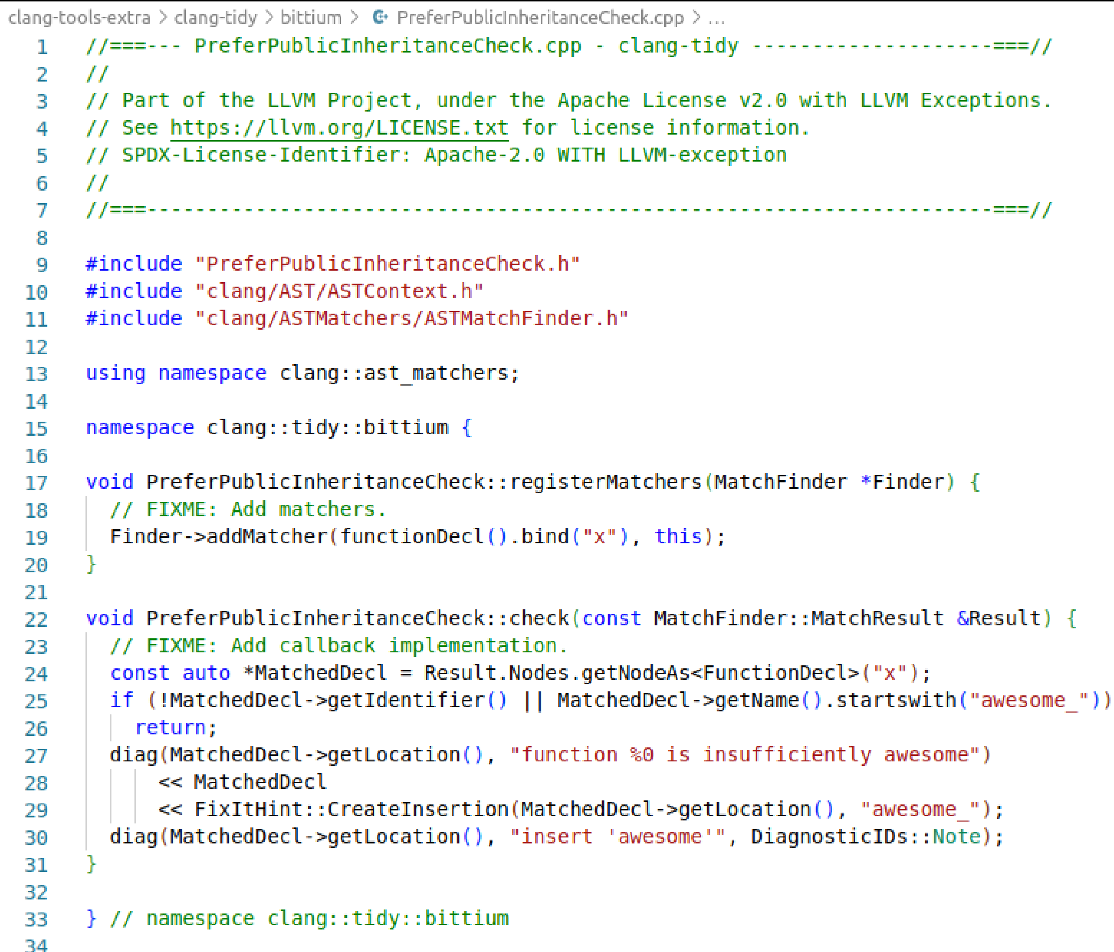
main.cpp

```cpp
#include <iostream>

void doNothing();

class Integer
{
public:
    Integer& operator=(const Integer& other);

    int getValue();
private:
    int value;
};

int main()
{
    1; // Useless, we do nothing with the literal
    true == true; // Useless, we do nothing with the result

    int a = 0;
    a; // Useless, we do nothing with the variable
    a = a; // Useless, self assignment
    a = a + 0; // Useless, the value does not change
    a = a + 1 - 1; // Useless, the value does not change

    // Useless, does nothing
    if (true)
    {
    }

    doNothing(); // Useless, does nothing

    Integer integer;
    integer.getValue(); // Useless, getter return value is discarded
    (void)integer.getValue(); // Useless, return value still discarded
    integer = integer; // Not useless, as the assignment also prints something

    return 0;
}

void doNothing()
{
}

Integer& Integer::operator=(const Integer& other)
{
    std::cout << "Hello, I hope you have a nice day\n";

    value = other.value;
    return *this;
}

int Integer::getValue()
{
    return value;
}

```

Ohjelman add_new_check.py luomat ja muokkaamat tiedostot

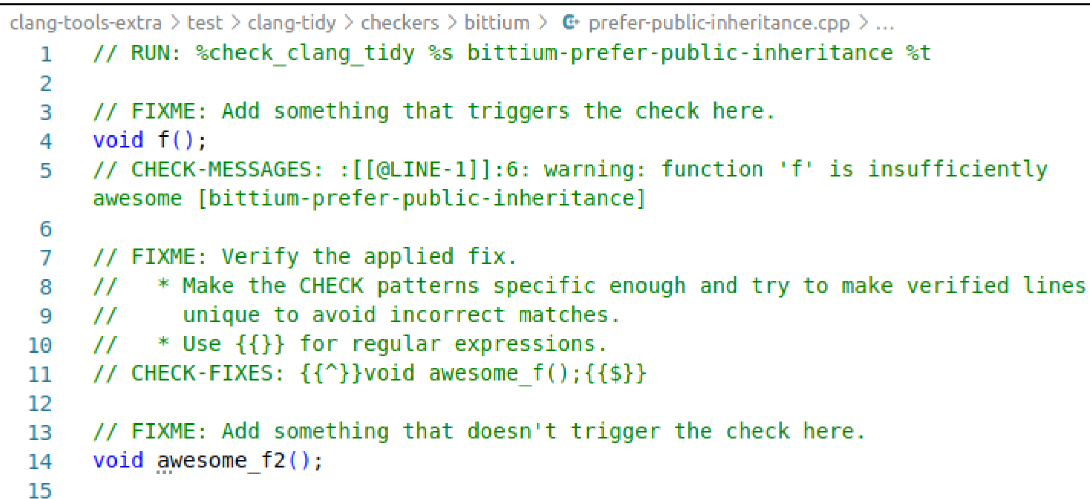
clang-tools-extra > clang-tidy > bittium > C PreferPublicInheritanceCheck.h > ...

```
1   //===--- PreferPublicInheritanceCheck.h - clang-tidy ------------*- C++ -*-===//
2   //
3   // Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
4   // See https://llvm.org/LICENSE.txt for license information.
5   // SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
6   //
7   //===----------------------------------------------------------------------===//
8
9   #ifndef LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_BITTIUM_PREFERPUBLICINHERITANCECHECK_H
10  #define LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_BITTIUM_PREFERPUBLICINHERITANCECHECK_H
11
12  #include "../ClangTidyCheck.h"
13
14  namespace clang::tidy::bittium {
15
16  /// FIXME: Write a short description.
17  ///
18  /// For the user-facing documentation see:
19  /// http://clang.llvm.org/extra/clang-tidy/checks/bittium/prefer-public-inheritance.html
20  class PreferPublicInheritanceCheck : public ClangTidyCheck {
21  public:
22    PreferPublicInheritanceCheck(StringRef Name, ClangTidyContext *Context)
23        : ClangTidyCheck(Name, Context) {}
24    void registerMatchers(ast_matchers::MatchFinder *Finder) override;
25    void check(const ast_matchers::MatchFinder::MatchResult &Result) override;
26  };
27
28  } // namespace clang::tidy::bittium
29
30  #endif // LLVM_CLANG_TOOLS_EXTRA_CLANG_TIDY_BITTIUM_PREFERPUBLICINHERITANCECHECK_H
31
```



clang-tools-extra > clang-tidy > bittium > PreferPublicInheritanceCheck.cpp > ...

```cpp
//===--- PreferPublicInheritanceCheck.cpp - clang-tidy --------------------===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//

#include "PreferPublicInheritanceCheck.h"
#include "clang/AST/ASTContext.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"

using namespace clang::ast_matchers;

namespace clang::tidy::bittium {

void PreferPublicInheritanceCheck::registerMatchers(MatchFinder *Finder) {
  // FIXME: Add matchers.
  Finder->addMatcher(functionDecl().bind("x"), this);
}

void PreferPublicInheritanceCheck::check(const MatchFinder::MatchResult &Result) {
  // FIXME: Add callback implementation.
  const auto *MatchedDecl = Result.Nodes.getNodeAs<FunctionDecl>("x");
  if (!MatchedDecl->getIdentifier() || MatchedDecl->getName().startswith("awesome_"))
    return;
  diag(MatchedDecl->getLocation(), "function %0 is insufficiently awesome")
      << MatchedDecl
      << FixItHint::CreateInsertion(MatchedDecl->getLocation(), "awesome_");
  diag(MatchedDecl->getLocation(), "insert 'awesome'", DiagnosticIDs::Note);
}

} // namespace clang::tidy::bittium

```



clang-tools-extra > test > clang-tidy > checkers > bittium > prefer-public-inheritance.cpp > ...

```cpp
// RUN: %check_clang_tidy %s bittium-prefer-public-inheritance %t

// FIXME: Add something that triggers the check here.
void f();
// CHECK-MESSAGES: :[[@LINE-1]]:6: warning: function 'f' is insufficiently awesome [bittium-prefer-public-inheritance]

// FIXME: Verify the applied fix.
//   * Make the CHECK patterns specific enough and try to make verified lines
//     unique to avoid incorrect matches.
//   * Use {{}} for regular expressions.
// CHECK-FIXES: {{^}}void awesome_f();{{$}}

// FIXME: Add something that doesn't trigger the check here.
void awesome_f2();

```

clang-tools-extra > docs > ReleaseNotes.rst

```
117  New checks
118  ^^^^^^^^^^
119
120  - New :doc:`bittium-prefer-public-inheritance
121    <clang-tidy/checks/bittium/prefer-public-inheritance>` check.
122
123    FIXME: add release notes.
124
125  - New :doc:`bugprone-empty-catch
126    <clang-tidy/checks/bugprone/empty-catch>` check.
127
128    Detects and suggests addressing issues with empty catch statements.
129
```

clang-tools-extra > docs > clang-tidy > checks > list.rst

```
36  .. csv-table::
37     :header: "Name", "Offers fixes"
38
39     `abseil-cleanup-ctad <abseil/cleanup-ctad.html>`_, "Yes"
40     `abseil-duration-addition <abseil/duration-addition.html>`_, "Yes"
41     `abseil-duration-comparison <abseil/duration-comparison.html>`_, "Yes"
42     `abseil-duration-conversion-cast <abseil/duration-conversion-cast.html>`_, "Yes"
43     `abseil-duration-division <abseil/duration-division.html>`_, "Yes"
44     `abseil-duration-factory-float <abseil/duration-factory-float.html>`_, "Yes"
45     `abseil-duration-factory-scale <abseil/duration-factory-scale.html>`_, "Yes"
46     `abseil-duration-subtraction <abseil/duration-subtraction.html>`_, "Yes"
47     `abseil-duration-unnecessary-conversion <abseil/duration-unnecessary-conversion.html>`_, "Yes"
48     `abseil-faster-strsplit-delimiter <abseil/faster-strsplit-delimiter.html>`_, "Yes"
49     `abseil-no-internal-dependencies <abseil/no-internal-dependencies.html>`_,
50     `abseil-no-namespace <abseil/no-namespace.html>`_,
51     `abseil-redundant-strcat-calls <abseil/redundant-strcat-calls.html>`_, "Yes"
52     `abseil-str-cat-append <abseil/str-cat-append.html>`_, "Yes"
53     `abseil-string-find-startswith <abseil/string-find-startswith.html>`_, "Yes"
54     `abseil-string-find-str-contains <abseil/string-find-str-contains.html>`_, "Yes"
55     `abseil-time-comparison <abseil/time-comparison.html>`_, "Yes"
56     `abseil-time-subtraction <abseil/time-subtraction.html>`_, "Yes"
57     `abseil-upgrade-duration-conversions <abseil/upgrade-duration-conversions.html>`_, "Yes"
58     `altera-id-dependent-backward-branch <altera/id-dependent-backward-branch.html>`_,
59     `altera-kernel-name-restriction <altera/kernel-name-restriction.html>`_,
60     `altera-single-work-item-barrier <altera/single-work-item-barrier.html>`_,
61     `altera-struct-pack-align <altera/struct-pack-align.html>`_, "Yes"
62     `altera-unroll-loops <altera/unroll-loops.html>`_,
63     `android-cloexec-accept <android/cloexec-accept.html>`_, "Yes"
64     `android-cloexec-accept4 <android/cloexec-accept4.html>`_, "Yes"
65     `android-cloexec-creat <android/cloexec-creat.html>`_, "Yes"
66     `android-cloexec-dup <android/cloexec-dup.html>`_, "Yes"
67     `android-cloexec-epoll-create <android/cloexec-epoll-create.html>`_, "Yes"
68     `android-cloexec-epoll-create1 <android/cloexec-epoll-create1.html>`_, "Yes"
69     `android-cloexec-fopen <android/cloexec-fopen.html>`_, "Yes"
70     `android-cloexec-inotify-init <android/cloexec-inotify-init.html>`_, "Yes"
71     `android-cloexec-inotify-init1 <android/cloexec-inotify-init1.html>`_, "Yes"
72     `android-cloexec-memfd-create <android/cloexec-memfd-create.html>`_, "Yes"
73     `android-cloexec-open <android/cloexec-open.html>`_, "Yes"
74     `android-cloexec-pipe <android/cloexec-pipe.html>`_, "Yes"
75     `android-cloexec-pipe2 <android/cloexec-pipe2.html>`_, "Yes"
76     `android-cloexec-socket <android/cloexec-socket.html>`_, "Yes"
77     `android-comparison-in-temp-failure-retry <android/comparison-in-temp-failure-retry.html>`_,
78     `bittium-prefer-public-inheritance <bittium/prefer-public-inheritance.html>`_, "Yes"
79     `boost-use-to-string <boost/use-to-string.html>`_, "Yes"
```

clang-tools-extra > docs > clang-tidy > checks > bittium > prefer-public-inheritance.rst

```
1  .. title:: clang-tidy - bittium-prefer-public-inheritance
2
3  bittium-prefer-public-inheritance
4  =================================
5
6  FIXME: Describe what patterns does the check detect and why. Give examples.
7
```

Ohjelman clang-query ongelmat *srcloc*-asetukseen liittyen

Ohjelma

clang-query.

Versio

LLVM version 17.0.3. Koonti on tehty opinnäytetyössä kuvatulla tavalla.

Käyttöjärjestelmä

Ubuntu 22.04.3 LTS.

Ongelman kuvaus

Ongelma ilmenee opinnäytetyön kuvan 21 mukaista tiedostoa tutkimalla, kun *srcloc* otetaan käyttöön. Debug-tilassa koottu ohjelma ilmoittaa, ettei ominaisuus ole saatavilla. Release-tilassa koottua ohjelmaa käytettäessä ominaisuus toimii osalla hauista, kun taas joidenkin hakujen tapauksessa ohjelma kaatuu asetuksen ollessa päällä ja hakua tehdessä.

Ongelman toisintaminen

Tarkastellaan edellä mainittua tiedostoa clang-querylla:

1. *enable output srcloc*
2. *match cxxRecordDecl(hasDirectBase(anyOf(isPrivate(), isProtected())))*

Toteutunut käyttäytyminen

Debug-koonnilla ohjelma tulostaa kohdan 1 jälkeen, ettei ominaisuus ole saatavilla.

Release-koonnilla edellä mainittua ilmoitusta ei tule, mutta ohjelma kaatuu kohdan 2 jälkeen. Ennen kaatumista tulostetaan ensimmäinen löydös normaalilla tyylillä ja merkkijono *"root" Source locations* alleviivattuna. Tämän jälkeen ohjelma kaatuu. Virheilmoituksen perusteella tiedoston *llvm-project/clang/include/clang/AST/DeclCXX.h* metodissa *clang::CXXRecordDecl::getLambdaData* oleva assertio *DD && DD->IsLambda && "queried lambda property of non-lambda class"* epäonnistuu. Myös virheeseen johtanut kutsupino tulostetaan.

Toivottu käyttäytyminen

Ohjelma tulostaa hakua vastaavat tulokset. Tulosteessa korostetaan AST-solmujen kiinnostavia kohtia ja millä metodeilla näihin päästään käsiksi.

Seuraava haku toimii halutulla tavalla release-koonnilla:

1. *enable output srcloc*
2. *match returnStmt()*