



**AGH**

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

---

## **Praca inżynierska**

**Jarosław Cierpich**  
**Arkadiusz Kasprzak**

kierunek studiów: **informatyka stosowana**

# **Rozbudowa i uaktualnienie oprogramowania systemu GGSS detektora ATLAS TRT**

Opiekun: **dr hab. inż. Bartosz Mindur**

**Kraków, styczeń 2020**

### Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelnia przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....  
(czytelny podpis)

## Spis treści

<b>1. Wstęp</b>	5
1.1. Wprowadzenie do systemu GGSS	5
1.2. Cel pracy	5
<b>2. Zastosowane technologie</b>	7
2.1. Język C++	7
2.2. Biblioteki	9
2.2.1. Rodzaje bibliotek	10
2.2.2. Biblioteki statyczne	10
2.2.3. Biblioteki współdzielone	13
2.2.4. Porównanie rodzajów bibliotek	14
2.3. Narzędzie CMake	14
2.4. Język Python	15
2.5. Powłoka systemu operacyjnego - Bash	17
2.6. System kontroli wersji Git i portal Gitlab	18
2.7. Manager pakietów - RPM	21
2.8. Technologie wirtualizacji i konteneryzacji	23
<b>3. Stan początkowy projektu</b>	27
3.1. Architektura	27
3.2. Budowanie	27
3.3. Dostarczanie i uruchamianie	29
3.4. Kontrola wersji	29
<b>4. Stan docelowy projektu</b>	31
<b>5. Ograniczenia dostępnej infrastruktury</b>	33
5.1. Ograniczone uprawnienia w środowisku docelowym	33
5.2. Wersje kompilatorów i interpreterów	33
5.3. Wersja narzędzia budującego CMake	34
5.4. Związek projektu z wersją jądra systemu	34

<b>6. Wykonane prace .....</b>	<b>35</b>
6.1. Wykorzystanie funkcjonalności portalu Gitlab wspierających zarządzanie projektem .....	35
6.2. Migracja projektu do systemu kontroli wersji Git i zmiany w architekturze .....	35
6.3. Zastosowanie podejścia CI/CD .....	35
6.4. Zmiana sposobu budowania aplikacji.....	35
6.5. Budowanie i dystrybucja sterownika oraz aplikacji testującej .....	35
6.6. Maszyna wirtualna oraz konteneryzacja - Docker .....	35
6.7. Pomniejsze prace.....	35
6.7.1. Integracja bibliotek napisanych w języku C z aplikacją w C++ .....	35
6.7.2. Integracja zewnętrznej biblioteki dynamicznej z użyciem narzędzia CMake .	35
6.8. Dokumentacja projektu.....	35
<b>7. Dalsza ścieżka rozwoju projektu .....</b>	<b>37</b>
7.1. Wprowadzenie zautomatyzowanego systemu testowania projektu .....	37
7.2. Migracja do nowego standardu języka C++ .....	37
7.3. Automatyzacja procesu publikowania produktu .....	37
<b>8. Podsumowanie oraz wnioski .....</b>	<b>39</b>
8.1. Statystyki projektu .....	39
<b>A. Dodatki/Appendixes .....</b>	<b>41</b>
A.1. Adding new modules to the project using existing CMake templates.....	41
A.2. Preparing virtual machine to work as a runner .....	41

# 1. Wstęp

## 1.1. Wprowadzenie do systemu GGSS

Detektor ATLAS (*A Toroidal LHC ApparatuS*), znajdujący się w Europejskim Ośrodku Badań Jądrowych *CERN*, jest jednym z detektorów pracujących przy Wielkim Zderzaczu Hadronów (*LHC - Large Hadron Collider*). Pełni on kluczową rolę w rozwoju fizyki cząstek elementarnych, w szczególności badania przy nim prowadzone doprowadziły do potwierdzenia istnienia tzw. *bozonu Higgsa* w roku 2012 [1].

Detektor ATLAS charakteryzuje się budową warstwową - składa się z kilku subdetektorów [2]. Jednym z nich jest Detektor Wewnętrzny (*Inner Detector*) składający się z trzech głównych elementów zbudowanych za pomocą różnych technologii. Elementy te, w kolejności od położonego najbliżej punktu zderzeń cząstek, to: detektor pikselowy (*Pixel Detector*), krzemowy detektor śladów (*SCT - Semiconductor Tracker*) oraz detektor promieniowania przejścia (*TRT - Transition Radiation Tracker*). Dokładny opis zasad działania całego detektora oraz poszczególnych jego komponentów wykracza poza zakres niniejszego manuskryptu.

W kontekście niniejszej pracy kluczowym jest System Stabilizacji Wzmocnienia Gazowego (*GGSS - Gas Gain Stabilisation System*) dla detektora TRT. Jego oprogramowanie jest zintegrowane [2] z systemem kontroli detektora ATLAS (*DCS - Detector Control System*). W skład systemu GGSS wchodzi zarówno urządzenia takie jak multiplexer i zasilacz wysokiego napięcia, jak i rozbudowana warstwa oprogramowania. Autorzy pracy zaprezentują opis zmian, jakie do tej pory wprowadzili w projekcie GGSS. Zmiany te obejmują m.in. sposób budowania aplikacji wchodzących w skład systemu, ale również automatyzacja prac związanych z jego utrzymaniem i użytkowaniem.

## 1.2. Cel pracy

Przed autorami postawiony został szereg celów do zrealizowania, związanych zarówno ze zdobyciem wymaganej wiedzy domenowej, jak i przeprowadzeniem modyfikacji oprogramowania systemu GGSS.

Jednym z nich było zapoznanie się z infrastrukturą informatyczną CERN-u. Praca z oprogramowaniem oparta jest tam o unikalny ekosystem, mający zapewnić bezpieczeństwo i stabilność

całej infrastruktury, co wiąże się z wieloma ograniczeniami dotyczącymi m.in. dostępu do komputerów produkcyjnych. Konieczne było więc uzyskanie odpowiednich uprawnień i zdobycie doświadczenia w pracy z tą infrastrukturą. Ze względu na domenę działania systemu GGSS celem było również zdobycie wiedzy na temat sposobu pracy przy dużych eksperymentach, na przykładzie eksperymentu ATLAS. Ponadto uczestnictwo w rozwoju projektu tego typu miało na celu nabycie przez autorów doświadczenia w pracy w międzynarodowym środowisku, jakim jest CERN. Kluczowym dla poprawnego przeprowadzenia prac było również zapoznanie się autorów z zastosowaniem i podstawami sposobu działania systemu GGSS.

Oprócz wyżej wymienionych czynności związanych ze zdobyciem podstawowej wiedzy domenowej, celem niniejszej pracy było przeprowadzenie modyfikacji w warstwie oprogramowania projektu GGSS. Do postawionych przed autorami zadań należało zaplanowanie prac i utworzenie wygodnego, nowoczesnego środowiska do zarządzania projektem informatycznym oraz utworzenie prostego w rozwoju, intuicyjnego systemu budowania oprogramowania opartego o narzędzie CMake. Miało to na celu umożliwienie modularyzacji projektu tak, by każdy z komponentów mógł być niezależnie budowany. Ponadto zadaniem autorów była migracja projektu do systemu kontroli wersji *Git*, stanowiącego ogólnoprzyjęty standard we współczesnych projektach informatycznych. W celu uproszczenia procedury wdrażania projektu w środowisku produkcyjnym celem autorów było również zautomatyzowanie procesu budowania i dystrybucji projektu. Na koniec, by umożliwić innym uczestnikom projektu sprawne korzystanie z nowych rozwiązań, przygotowana miała zostać dokumentacja projektu w formie krótkich instrukcji czy zestawów komend. Dokumentacja, z uwagi na międzynarodowy charakter środowiska w CERN, miała zostać napisana w języku angielskim.

Niniejszy manuskrypt opisuje przede wszystkim prace związane z rozwojem oprogramowania przeprowadzone przez autorów. Praca opisuje stan początkowy projektu, założenia dotyczące stanu docelowego oraz wybrane, zdaniem autorów najważniejsze, zadania zrealizowane w ramach pracy z oprogramowaniem systemu GGSS.

## 2. Zastosowane technologie

Niniejszy rozdział zawiera krótki opis najważniejszych technologii i narzędzi używanych przez autorów podczas pracy z oprogramowaniem systemu GGSS. Przedstawione tu opisy zawierają podstawową wiedzę o sposobie działania i użytkowania tych technologii - szczegółowe przykłady przedstawione zostały w dalszej części pracy, w kontekście konkretnych rozwiązań zrealizowanych przez autorów w projekcie.

### 2.1. Język C++

C++ jest kompilowanym językiem programowania ogólnego przeznaczenia [3] opartym o statyczne typowanie. Został stworzony jako obiektowe rozszerzenie języka C (z którym jest w dużej mierze wstecznie kompatybilny), lecz wraz z rozwojem pojawiło się w nim wsparcie dla innych paradygmatów, w tym generycznego i funkcyjnego. Sprawilo to, że język ten stał się bardzo wszechstronny - pozwala zarówno na szybkie wykonywanie operacji niskopoziomowych, jak i na tworzenie wysokopoziomowych abstrakcji [3]. Dodatkową cechą wyróżniającą C++ wśród innych języków umożliwiającą programowanie obiektowe jest jego wysoka wydajność.

#### Standardy języka

W ciągu ostatnich kilku lat C++ przechodzi proces intensywnego rozwoju - od 2011 roku pojawiły się trzy nowe standardy tego języka, a kolejny przewidziany jest na rok 2020. Wspomniane nowe standardy to:

- C++11 - wprowadza funkcjonalności takie jak: wsparcie dla wielowątkowości, wyrażenia lambda, referencje do *r-wartości*, biblioteka do obsługi wyrażeń regularnych, dedukcja typów za pomocą słowa kluczowego *auto* czy pętla zakresowa. Standard ten uważany jest za przełom w rozwoju języka.
- C++14 - rozszerza zmiany wprowadzone w C++11. Nie zawiera tak wielu przełomowych zmian jak poprzedni standard - twórcy skupili się na poprawie istniejących błędów oraz rozwoju istniejących rozwiązań [4], np. dedukcji typu zwracanego z funkcji za pomocą słowa kluczowego *auto*.

- C++17 - wprowadza m.in. nowe typy danych (np. `std::variant`, `std::byte` i `std::optional`), algorytmy współbieżne, biblioteka *filesystem* przeznaczona do obsługi systemu plików oraz rozszerzenie mechanizmu dedukcji typów w szablonach na szablony klas [5]. Standard ten usuwa również pewne elementy uznane za przestarzałe, np. inteligentny wskaźnik `std::auto_ptr`, zastąpiony w standardzie C++11 przez inne rozwiązanie.

Zmiany wprowadzane w nowych standardach pozwalają na tworzenie czytelniejszego kodu, który łatwiej utrzymywać i rozwijać. Ma to znaczenie zarówno na poziomie pojedynczych instrukcji czy typów danych, jak i na poziomie architektury projektu. Listingi 2.1 oraz 2.2 przedstawiają przykład zmiany, jaka zaszła między starym standardem C++03, a C++11. Zaprezentowany kod realizuje w obu przypadkach iterację po zawartości kontenera typu `std::vector<int>` mającą na celu wypisanie na standardowe wyjście jego zawartości. Przykład ten, pomimo że bardzo prosty, dobrze obrazuje wzrost jakości i czytelności kodu w nowym standardzie.

**Listing 2.1.** Przykład kodu w języku C++ napisany z wykorzystaniem standardu C++03

```
// kontener zawierający 6 elementów typu int - inicjalizacja
// za pomocą tymczasowej tablicy, możliwa w statym standardzie
// języka
int tmp_arr[] = {1, 2, 3, 4, 5, 6};
std::vector<int> a (tmp_arr, tmp_arr + 6);

// iteracja po zawartości kontenera w standardzie C++03
for (std::vector<int>::const_iterator it = a.begin(); it != a.end(); ++it) {
    std::cout << *it << " ";
}
```

**Listing 2.2.** Przykład kodu w języku C++ napisany z wykorzystaniem funkcjonalności ze standardu C++11 (zakresowa pętla for)

```
// kontener zawierający 6 elementów typu int - nowy
// sposób inicjalizacji
std::vector<int> a{1, 2, 3, 4, 5, 6};

// iteracja po zawartości kontenera w standardzie C++11 -
// przykład zastosowania zakresowej pętli for
for (const auto& elem: a) {
    std::cout << elem << " ";
}
```

## Boost

*Boost* jest zestawem bibliotek dla języka C++, poszerzających w znacznym stopniu wachlarz narzędzi programistycznych dostarczanych przez język. Biblioteki wchodzące w skład Boost dostarczają funkcjonalności takich, jak: wygodne przetwarzanie tekstu, zapewnienie interfejsu



między C++ a językiem Python czy programowanie sieciowe [6]. Boost to projekt aktywnie rozwijany, bardzo popularny. Niektóre z bibliotek wchodzących w jego skład zostały przeniesione (nie zawsze w postaci identycznej względem oryginału) do standardu C++.

## 2.2. Biblioteki

Biblioteki są jednym z podstawowych narzędzi wprowadzających podział programu na niezależne komponenty oraz dających możliwość wielokrotnego użycia tego samego kodu. Stanowią więc zbiór funkcji, struktur itp. udostępnionych do użycia przez inne programy. Niniejsza część pracy skupia się na bibliotekach opisywanych z perspektywy języków C oraz C++. Opis dotyczył będzie rozwiązań związanych z systemami typu UNIX (nie zostanie poruszony sposób działania bibliotek na systemach Windows). Autorzy zdecydowali się opisać to zagadnienie szczegółowo z uwagi na fakt, iż architektura projektu GGSS w dużej mierze opiera się o mechanizm bibliotek. Rozważania teoretyczne wzbogacone zostaną więc prostym przykładem.

### Opis przykładu

Przykład prezentujący działanie bibliotek został napisany w języku C i składa się z dwóch katalogów: *app*, zawierającego kod źródłowy programu, oraz *complex*, zawierającego kod źródłowy, który zostanie wykorzystany do stworzenia prostej biblioteki pozwalającej na wykonywanie podstawowych operacji na liczbach zespolonych. Listing 2.3 zawiera wynik polecenia *tree*, które wypisuje na standardowe wyjście strukturę katalogu z projektem. Autorzy zdecydowali się pokazać proces budowania bibliotek bez wykorzystania narzędzi automatyzujących ten proces (takich jak *CMake*), gdyż znajomość zasad działania tego mechanizmu okazała się dla nich bardzo pomocna podczas rozwiązywania problemów związanych z wykorzystaniem bibliotek w systemie GGSS, gdzie wspomniane narzędzia były już używane.

**Listing 2.3.** Struktura katalogów projektu stanowiącego bazę przykładu dotyczącego bibliotek.

```
user@host:~$ tree --charset=ascii
.
|-- app
|   '-- app.c
'-- complex
    |-- complex_number.h
    |-- complex_ops.c
    '-- complex_ops.h

2 directories, 4 files
```

### 2.2.1. Rodzaje bibliotek

Na systemach z rodziny UNIX wyróżniamy dwa podstawowy typy bibliotek: **statyczne** oraz **współdzielone (shared)**, nazywane również **dynamicznymi**. Podejścia te znacznie różnią się od siebie. Każde z nich oferuje pewne zalety względem drugiego, przez co oba pozostają dziś w użyciu.

### 2.2.2. Biblioteki statyczne

Koncepcja stojąca za bibliotekami statycznymi jest bardzo prosta - są to archiwa zawierające w sobie kolekcję plików obiektowych (\*.o). Do tego typu bibliotek dołączone muszą być pliki nagłówkowe zawierające m.in. deklaracje funkcji stanowiących interfejs pomiędzy biblioteką, a używającym ją programem. Cechą bibliotek statycznych odróżniającą je od bibliotek dynamicznych jest fakt, że są one dołączane do plików obiektowych głównego programu w czasie linkowania - stanowią więc część wynikowego pliku wykonywalnego.

#### Tworzenie biblioteki statycznej

Rysunek 2.1 przedstawia schematycznie proces tworzenia bibliotek statycznych. Składa się on z dwóch etapów:

- kompilacja plików źródłowych biblioteki do postaci obiektowej za pomocą *gcc* (listing 2.4). Wynikiem powinny być pliki o rozszerzeniu \*.o odpowiadające wykorzystanym plikom źródłowym.

**Listing 2.4.** Kompilacja plików źródłowych biblioteki do postaci obiektowej - polecenie oraz jego wynik

```
user@host:~/complex$ gcc -c *.c
user@host:~/complex$ ls
complex_number.h  complex_ops.c  complex_ops.h  complex_ops.o
```

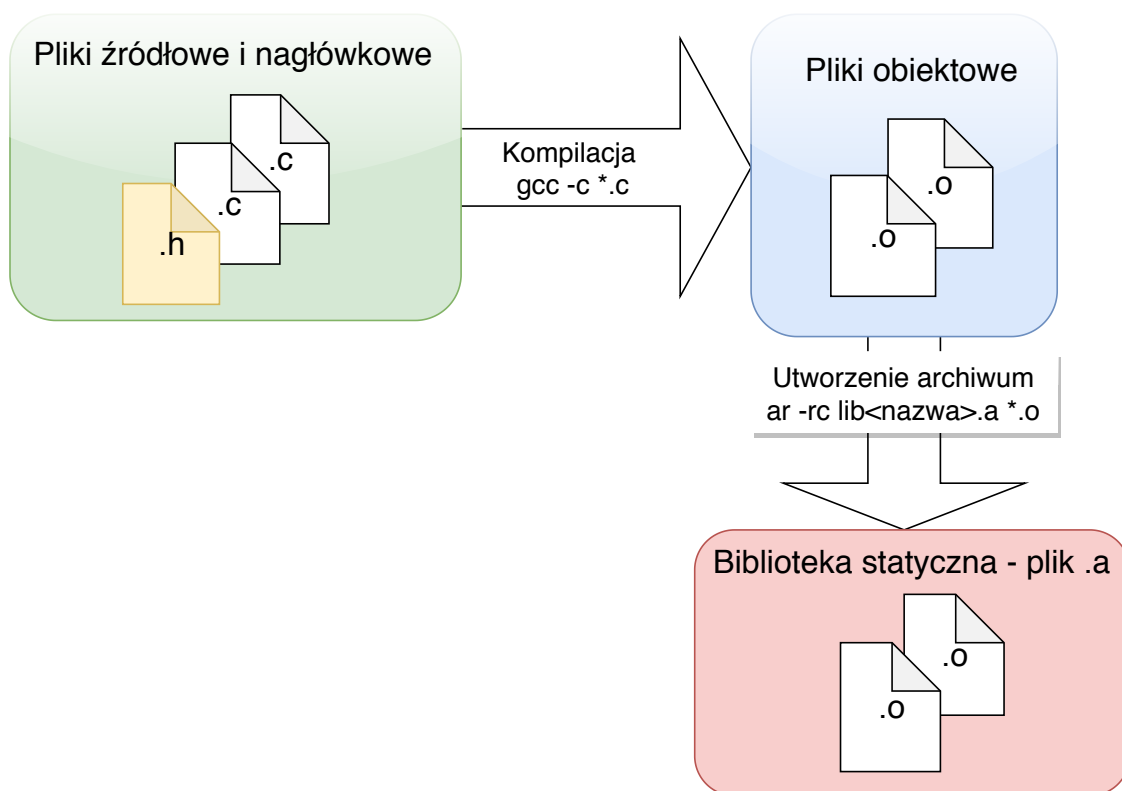
- utworzenie archiwum zawierającego pliki obiektowe za pomocą programu *archiver* (listing 2.5). Wynikiem powinien być plik o rozszerzeniu \*.a. Podczas tworzenia biblioteki należy nadać jej odpowiednią nazwę, zgodną z formatem *lib<nazwa>.a*.

**Listing 2.5.** Utworzenie biblioteki statycznej z plików obiektowych - polecenie oraz jego wynik

```
user@host:~/complex$ ar -rc libcoml.a *.o
user@host:~/complex$ ls
complex_number.h  complex_ops.c  complex_ops.h  complex_ops.o  libcoml.a
```

Zawartość powstałego archiwum można zbadać za pomocą komendy *ar -t* - wyświetla ona wszystkie pliki obiektowe wchodzące w skład danej biblioteki. Istnieje również możliwość wylistowania symboli - służy do tego narzędzie *nm*. Użycie tych narzędzi na wykonanym przez autorów

**Rys. 2.1.** Proces tworzenia biblioteki statycznej z uwzględnieniem komend koniecznych do wykonania poszczególnych etapów



przykładzie ilustruje listing 2.6. Wynikiem polecenia `nm` są tam dwa symbole, oznaczające dwie udostępnione dla użytkowników biblioteki funkcje (dodawanie i odejmowanie liczb zespolonych).

**Listing 2.6.** Użycie poleceń `ar -t` oraz `nm` na bibliotece statycznej.

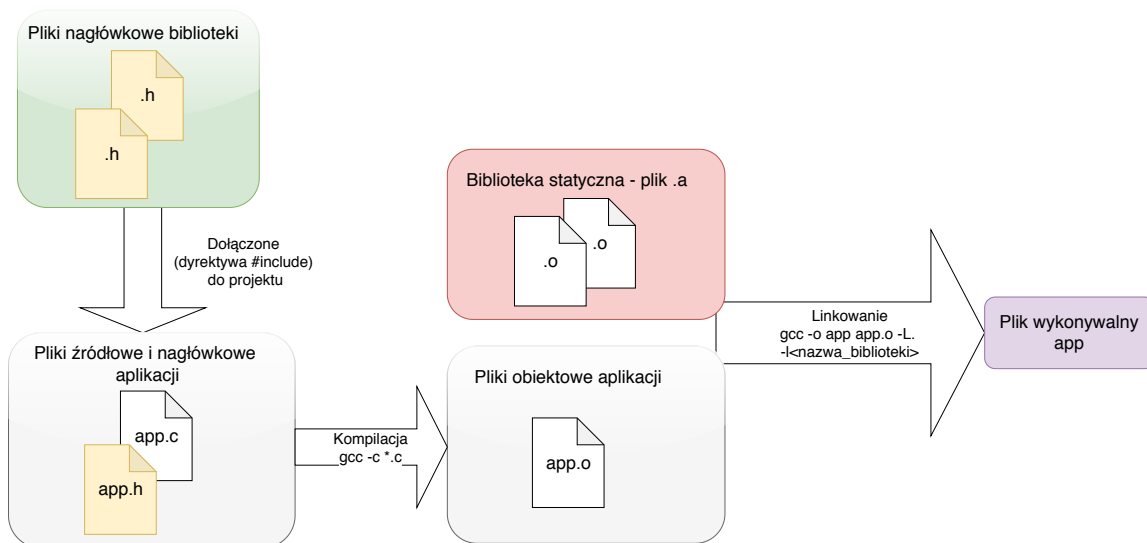
```
user@host:~/complex$ ar -t libcoml.a
complex_ops.o
user@host:~/complex$ nm libcoml.a

complex_ops.o:
0000000000000000 T add_complex_numbers
0000000000000091 T subtract_complex_numbers
```

### Dołączanie utworzonej biblioteki do programu

Rysunek 2.2 przedstawia schematycznie proces dołączania utworzonej biblioteki statycznej do programu. Proces ten składa się z następujących etapów:

**Rys. 2.2.** Proces dołączania biblioteki statycznej do programu z uwzględnieniem komend koniecznych do wykonania poszczególnych etapów



- dołączenie do źródeł programu plików nagłówkowych zawierających deklaracje stanowiące interfejs między programem a biblioteką (dyrektywa `include` - listing 2.7).

**Listing 2.7.** Plik `app.c` zawierający dyrektywę `include` dołączającą plik nagłówkowy zawierający deklaracje funkcji z biblioteki statycznej

```
#include "../complex/complex_ops.h"
#include <stdio.h>
```

```

int main(void) {

    complex_number a = {2.5, 3.7};
    complex_number b = {3.5, 0};

    complex_number c = add_complex_numbers(a, b);
    complex_number d = subtract_complex_numbers(a, b);

    printf("%lf %lf\n", c.real, c.imaginary);
    printf("%lf %lf\n", d.real, d.imaginary);

    return 0;
}

```

- kompilacja plików źródłowych programu do postaci plików obiektowych za pomocą *gcc* (listing 2.8). Wynikiem powinien być zestaw plików obiektowych odpowiadający wykorzystanym plikom źródłowym.

**Listing 2.8.** Kompilacja plików źródłowych głównego programu do postaci obiektowej.

```

user@host:~/app$ gcc -c *.c
user@host:~/app$ ls
app.c  app.o

```

- połączenie plików obiektowych i biblioteki w plik wykonywalny za pomocą *gcc* (listing 2.9). Opcja *-L* pozwala określić ścieżkę do dołączanej biblioteki, natomiast *-l* - nazwę biblioteki (bez przedrostka *lib* i rozszerzenie *.a*).

**Listing 2.9.** Linkowanie plików obiektowych programu z biblioteką statyczną, wynik uruchomienia programu obrazujący poprawne działanie przykładu.

```

user@host:~/app$ gcc -o app app.o -L../complex -lcoml
user@host:~/app$ ls
app  app.c  app.o
user@host:~/app$ ./app
6.000000 3.700000
-1.000000 3.700000

```

### 2.2.3. Biblioteki współdzielone

#### Tworzenie biblioteki współdzielonej

#### Dołączanie utworzonej biblioteki do programu

## Wersjonowanie bibliotek współdzielonych

### 2.2.4. Porównanie rodzajów bibliotek

## 2.3. Narzędzie CMake

*CMake* (*Cross-platform Make*) to narzędzie pozwalające na konfigurację procesu budowania oprogramowania (aplikacji oraz bibliotek) w sposób niezależny od platformy. Jego działanie opiera się na generowaniu pliku budującego natywnego dla określonej platformy [7] (dla systemów z rodziny UNIX jest nim *Makefile*) na podstawie przygotowanego przez użytkownika pliku *CMakeLists.txt*. Takie podejście w znacznym stopniu ułatwia tworzenie aplikacji multiplatformowych oraz pozwala na intuicyjne zarządzanie zależnościami w projekcie. Domyślnie CMake pracuje z językami C i C++, natomiast nowe wersje narzędzia wpierają ponadto m.in. język C# czy technologię CUDA [8]. Narzędzie to jest rozwijane i wspierane przez firmę *Kitware*.

### Plik CMakeLists.txt

Jak wspomniano wyżej działanie narzędzia CMake opiera się na przygotowanym przez użytkownika pliku (lub zestawie plików rozmieszczonych w strukturze katalogów projektu) *CMakeLists.txt*. Plik ten zawiera polecenia napisane w specjalnie do tego celu przygotowanym języku skryptowym. Użytkownik może za jego pomocą m.in. określać jakie pliki wykonywalne mają zostać wygenerowane podczas procesu budowania, wskazać lokalizację plików źródłowych czy określić zależności między komponentami projektu oraz bibliotekami zewnętrznymi.

### Prosty przykład

Listing 2.10 zawiera przykład prostego pliku *CMakeLists.txt*, pozwalającego na zbudowanie napisanej w języku C++ obiektowej wersji klasycznego programu *Hello world*. Przykład ilustruje zastosowanie podstawowych poleceń CMake do określenia minimalnej wersji narzędzia, standardu języka C++, wynikowego pliku wykonywalnego oraz potrzebnych plików nagłówkowych.

**Listing 2.10.** Przykład prostego pliku CMakeLists.txt przeznaczonego do budowania programu napisanego w C++

```
# Określenie minimalnej wersji CMake
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)

# Określenie standardu języka C++
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Nazwa oraz wersja projektu
project(Hello VERSION 1.0)
```

```
# Dodanie pliku wykonywalnego, który powinien powstać
# wskutek procesu budowania
add_executable(Hello Main.cpp)

# Dodanie do projektu katalogu include wraz ze znajdującym się
# wewnątrz niego plikiem nagłówkowym
target_include_directories(Hello PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}/include")
```

### Wersje CMake

CMake jest narzędziem, który w ciągu ostatnich kilku lat przechodzi gruntowne zmiany. Starsze wersje (np. 2.8) oparte są o prosty system zmiennych [9], co wprowadza szereg trudności w zarządzaniu dużymi projektami z wielopoziomowymi drzewami zależności. Dodatkowym problemem tych wersji jest również brak dobrze zdefiniowanych tzw. *dobrych praktyk* oraz nieprzystępna dla początkujących dokumentacja. Współczesne wersje narzędzie CMake (zwykle za takie uznaje się nowsze od wersji 3.0) opierają się na innym, bardziej ustrukturyzowanym [9] podejściu, co było przyczyną pojawienia się dla nich wyżej wspomnianych *dobrych praktyk*. Zalecane jest więc, by nowe projekty prowadzone były właśnie z użyciem nowszych wersji narzędzia.

### Narzędzia CTest i CPack

CMake oferuje również możliwość konfiguracji sposobu testowania projektu. Służy do tego narzędzie *CTest*, dystrybuowane razem z podstawowym narzędziem CMake. Innym przydatnym modulem jest *CPack* - narzędzie to służy przygotowywaniu pakietów instalacyjnych z oprogramowaniem. Użycie obu wymienionych narzędzi polega na umieszczeniu w pliku *CMakeLists.txt* kilku przeznaczonych do tego komend.

## 2.4. Język Python

*Python* jest nowoczesnym, wysokopoziomowym językiem programowania, wspierającym takie paradygmaty jak programowanie obiektowe czy imperatywne. Działanie Pythona opiera się na dynamicznym systemie typów. Z założenia Python jest językiem przyjemnym w użytkowaniu, co przyczyniło się do jego dużej popularności [10]. Python jest szeroko stosowany jako język skryptowy - takie też zastosowanie znalazł w projekcie GGSS.

### Prosty przykład

Listing 2.11 przedstawia prosty przykład skryptu napisanego w języku Python w wersji 3. Kod ten stanowi uproszczoną wersję skryptu zaprezentowanego w dalszej części pracy, którego zadaniem jest zbudowanie aplikacji w zależności od przekazanych przez użytkownika argumentów. Przykład prezentuje prosty skrypt przyjmujący jeden z dwóch możliwych argumentów i wypisujący informację na temat otrzymanego argumentu na standardowe wyjście.

**Listing 2.11.** Przykład prostego skryptu napisanego w języku Python 3 - przetwarzanie argumentów podanych przez użytkownika do skryptu

```
import argparse

## Prostý skrypt przetwarzający argumenty podane
## przy jego uruchomieniu przez użytkownika

## Definicja funkcji w języku Python
def parse_command_line_arguments():

    ## Obiekt przetwarzający argumenty (parser)
    parser = argparse.ArgumentParser()

    ## Argumenty wzajemnie się wykluczające
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument("-s", "--staticboost",
                      help="Use static Boost linking.", action="store_true")
    group.add_argument("-d", "--dynamicboost",
                      help="Use dynamic Boost linking.", action="store_true")

    return parser.parse_args()

if __name__=="__main__":
    ## Wywołanie funkcji
    arguments = parse_command_line_arguments()
    print(arguments)
```

## Wersje języka Python

Python funkcjonuje w dwóch wersjach: Python 2 oraz 3. Wersje te nie są ze sobą w pełni kompatybilne, tzn. pewne funkcjonalności Pythona 2 nie są dostępne w Pythonie 3 i odwrotnie. Różnice znaleźć można również np. w domyślnym sposobie kodowania łańcuchów znakowych (ASCII w Pythonie 2, Unicode w Pythonie 3) oraz w wyniku dzielenia (za pomocą operatora /) dwóch liczb całkowitych (w Pythonie 2 wynikiem jest liczba całkowita, w Pythonie 3 liczba zmiennoprzecinkowa typu *float*) [11]. Ponadto zakończenie oficjalnego wsparcia Pythona w wersji 2 przewidziane jest na styczeń 2020 roku [12] - co w momencie pisania niniejszej pracy (grudzień 2019) jest terminem niedalekim i miało kluczowe znaczenie w czasie podejmowania pewnych decyzji projektowych.

## Zewnętrzne biblioteki

Jedną z największych zalet Pythona jest bardzo duża liczba bibliotek zewnętrznych tworzonych przez społeczność Pythona. Rozbudowują one język o wiele nowych funkcjonalności, np. przetwarzanie plików HTML czy wykonywanie obliczeń numerycznych. W niniejszej pracy zastosowanych zostało kilka tego typu bibliotek, m.in. *Beautiful Soup* do wspomnianego wyżej



przetwarzania dokumentów w formacie HTML. Omówienie ich działania na przykładach znaleźć można w dalszej części pracy - przy opisie konkretnego ich zastosowania.

## 2.5. Powłoka systemu operacyjnego - Bash

Powłoka systemu jest programem, którego głównym zadaniem jest udostępnienie interfejsu umożliwiającego łatwy dostęp do funkcji systemu operacyjnego. Nazwę *powłoka* zawdzięcza temu, że jest warstwą okalającą system operacyjny. Najczęściej spotykanym rodzajem powłoki są tzw. interfejsy z wierszem poleceń (ang. command-line interface). Polecenia wprowadzane są do nich w modzie interaktywnym, tj. wykonywane są one w momencie wprowadzenia końca linii.

**Listing 2.12.** Komenda wypisująca tekst na standardowe wyjście wykonana z linii poleceń

```
user@host:~$ echo "interfejs z linią poleceń"
interfejs z linią poleceń
user@host:~$
```

Bash, czyli **Bourne Again Shell** jest powłoką systemu początkowo napisaną dla systemu operacyjnego GNU. Obecnie Bash jest kompatybilny z większością systemów Unixowych, gdzie zwykle jest powłoką domyślną oraz posiada kilka portów na inne platformy, tj.: MS-DOS, OS/2, Windows [13]. Oprócz pełnienia wyżej wymienionej funkcji, Bash jest również językiem programowania pozwalającym na tworzenie skryptów, które są kolejną metodą wprowadzania poleceń do powłoki systemu.

Korzystając z języka skryptowego powłoki Bash jesteśmy w stanie zawrzeć dodatkową logikę podczas wykonywania komend. Wspiera on takie struktury jak: instrukcje warunkowe, pętle, operacje logiczne oraz arytmetyczne. Aby wykorzystać Bash w skrypcie należy na początku pliku zamieścić zapis `#!/bin/bash`, gdzie `/bin/bash` to ścieżka do pliku interpretera Bash. Zachowanie skryptu jesteśmy w stanie uzależnić od argumentów wykonania. Ich obsługa odbywa się za pomocą zapisu `$?`, gdzie `?` jest to numer porządkowy argumentu liczony od 0.

**Listing 2.13.** Skrypt wykorzystujący argumenty wejściowe, instrukcję warunkową oraz polecenie echo

```
#!/bin/bash
if [ $1 == "argumenty" ]; then
    echo "Argument 0.: $0"
    echo "Argument 1.: $1"
else
    echo "Nieznane polecenie"
fi
```

**Listing 2.14.** Przykład działania Skryptu z Listingu 2.13

```
user@host:~$ /home/user/prostySkrypt.sh argumenty
```

```
Argument 0.: /home/user/prostySkrypt.sh
Argument 1.: argumenty
```

Bash posiada wiele poleceń, które pozwalają na wykonywanie zarówno podstawowych, jak i bardziej zaawansowanych czynności, np.: obsługa plików, obsługa systemu katalogów, zarządzanie kontami, uprawnieniami, itd.

Bash posiada również wiele zaawansowanych funkcjonalności, które pozwalają na kontrolowanie przepływu informacji w trakcie wykonywania poleceń. Przykładem jest wpisywanie tekstu do pliku ukazane na Listingu 2.15.

**Listing 2.15.** Przykład zapisu tekstu do pliku

```
user@host:~$ echo "Ten napis zostanie zapisany do pliku plik.txt" > plik.txt
user@host:~$ cat plik.txt
Ten napis zostanie zapisany do pliku plik.txt
```

W celu zapisania tekstu do pliku należy na standardowe wyjście przekazać napis za pomocą komendy **echo**, a następnie przekierować za pomocą zapisu **>**, który poprzedza nazwę pliku docelowego. W wyniku działania zawartość pliku **plik.txt** zostanie nadpisana, a w przypadku gdy takiego pliku nie ma, to zostanie on utworzony i uzupełniony o napis.

## 2.6. System kontroli wersji Git i portal Gitlab

System kontroli wersji Git jest oprogramowaniem służącym do śledzenia i zarządzania zmianami w plikach projektowych. W przypadku Git'a, aby zarejestrować pliki projektowe w celu ich śledzenia należy wykonać kilka czynności. Po pierwsze wymagane jest utworzenie repozytorium. Sprowadza się ono do wykonania odpowiedniej komendy Git'a wewnątrz folderu projektu, tj. **git init**. Podczas działania komendy wewnątrz folderu, w którym wywołaliśmy ww. polecenie, inicjowany jest ukryty folder **.git**. Jest on odpowiedzialny za przechowywanie konfiguracji dla tego repozytorium oraz zapisywanie informacji o wszystkich zmianach dokonanych w projekcie.

**Listing 2.16.** Inicjalizacja repozytorium git

```
user@host:/ścieżka/do/projektu$ git init
Initialized empty Git repository in /ścieżka/do/projektu
user@host:/ścieżka/do/projektu$ ls .git
branches  config  description  HEAD  hooks  info  objects  refs
```

Taka inicjalizacja nie spowoduje żadnego dodatkowego działania oprócz utworzenia repozytorium. Żadne pliki nie są jeszcze poddawane rewizji. W celu rejestracji plików należy wykonać jeszcze kilka kroków. Pierwszym z nich jest wykonanie komendy **git add**, która poprzedza nazwę plików lub folderów, które chcemy poddać wersjonowaniu. Elementy te zostają dodane do tzw. poczekalni, czyli są one kandydatami do utworzenia kolejnej rewizji. Przydatną komendą w tym

przypadku jest również **git status** pozwalająca na sprawdzenie obecnego stanu repozytorium. Wyświetla ono krótkie podsumowanie nt. nowych plików, usuniętych plików oraz plików zmodyfikowanych. Informuje nas również o tym, które pliki są brane pod uwagę do utworzenia kolejnej rewizji.

**Listing 2.17.** Dodawanie elementów do poczekalni

```
user@host:/ściezka/do/projektu# git add plik1 folder1
user@host:/ściezka/do/projektu# git status
On branch master

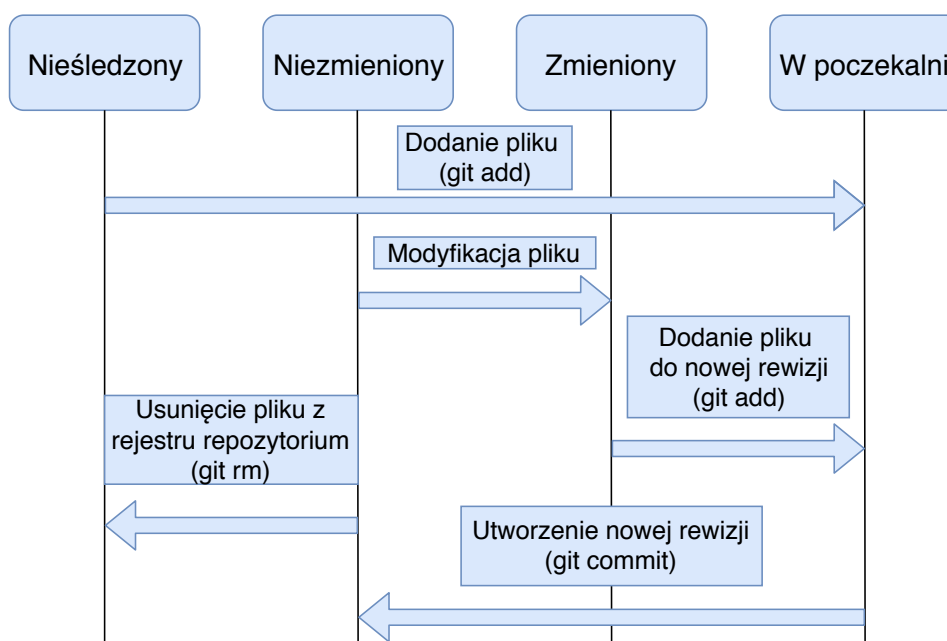
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   folder1/plik3
        new file:   folder1/plik4
        new file:   plik1

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        plik2
```



**Rys. 2.3.** Możliwe stany pliku w repozytorium GIT WSTAWIĆ REFERENCJE DO TEGO

Tworzenie nowej wersji w ramach repozytorium odbywa się za pomocą komendy **git commit**. Sprowadza się do 'zamrożenia' obecnych wersji plików zarejestrowanych do rewizji oraz przypisanie im wspólnego, unikalnego dla każdej z nich, identyfikatora. Git udostępnia komendy pozwalające na przeglądanie oraz przywracanie plików do wcześniej utworzonych wersji. Listing 2.18 przedstawia utworzenie nowej wersji oraz wyświetlenie podsumowania o utworzonych do tej pory rewizjach.

**Listing 2.18.** Utworzenie nowej rewizji

```
user@host:/sciezka/do/projektu# git commit -m "Pierwsza rewizja"
user@host:/sciezka/do/projektu# git log
commit 1d2445e961beb25940dffa9d73963f887ee553ad
Author: user <user@host.localdomain>
Date:    Wed Dec 4 17:29:55 2019 +0100
```

Pierwsza rewizja

Przejścia między rewizjami nie powodują utraty danych, gdyż zachowywana jest informacja o stanie plików dla każdej z nich, co ukazuje Listing 2.19. Tworzona jest nowa rewizja zawierająca dodatkowo **plik2**, natomiast po powrocie do poprzedniej wersji plik ten nie występuje. Gdy powrócimy do nowszej wersji ponownie pojawi się **plik2**.

**Listing 2.19.** Podsumowanie rewizji, powrót do starszej wersji

```
user@host:/sciezka/do/projektu# git add plik2
user@host:/sciezka/do/projektu# git commit -m "Druga rewizja"
user@host:/sciezka/do/projektu# git log
commit b58836df55fc2a8eb2a43aa96273853776924807
Author: user <user@host.localdomain>
Date:    Wed Dec 4 17:30:10 2019 +0100
```

Druga rewizja

```
commit 1d2445e961beb25940dffa9d73963f887ee553ad
Author: user <user@host.localdomain>
Date:    Wed Dec 4 17:29:55 2019 +0100
```

Pierwsza rewizja

```
user@host:/sciezka/do/projektu# ls
folder1 plik1 plik2
user@host:/sciezka/do/projektu# git checkout ↵
1d2445e961beb25940dffa9d73963f887ee553ad
user@host:/sciezka/do/projektu# ls
folder1 plik1
```

```
user@host:/sciezka/do/projektu# git checkout ↵  
b58836df55fc2a8eb2a43aa96273853776924807  
user@host:/sciezka/do/projektu# ls  
folder1 plik1 plik2
```

Głównym celem portalu **GitLab** jest udostępnienie środowiska do przechowywania repozytoriów Gitowych na zdalnych serwerach. Pozwala to na uniezależnienie się od maszyny na której pracujemy, zwiększa bezpieczeństwo plików źródłowych poprzez umieszczenie kopii na zdalnym serwerze oraz wspiera zespołową pracę nad kodem.

Ze względu na to, że portale typu **GitLab** są traktowane jako podstawowe narzędzie do wspólnej pracy nad kodem, to rozwinęły one wiele narzędzi wspomagających organizację oraz śledzenie pracy. Oprócz ww. funkcji **Gitlab** dostarcza wiele narzędzi do wspomagania procesu zapewniania jakości, jak i automatyzacji dostarczania kodu.

**Git** posiada specjalne komendy pozwalające na przekazywanie oraz pobieranie repozytoriów z ww. portali, czyli:

- **git clone**, która pozwala na pobranie repozytorium z portalu oraz zainicjalizowanie go lokalnie
- **git pull** - za pomocą tej komendy możemy zaktualizować repozytorium lokalne do najnowszej rewizji, która znajduje się na portalu
- oraz komenda **git push** aktualizująca zdalne repozytorium do naszej wersji

Jest to tylko podstawowy opis technologii jaką jest **Git**, przykłady bardziej zaawansowanych zastosowań pojawią się w części manuskryptu przeznaczony na prezentację wykonanych prac w ramach projektu.

## 2.7. Manager pakietów - RPM

Menadżer pakietów jest zbiorem oprogramowania, które w sposób automatyczny zarządza instalacją, aktualizacją, konfiguracją oraz usuwaniem programów komputerowych [14]. Ze względu na to, że procesy te różnią się w zależności od systemów operacyjnych oraz ich dystrybucji istnieje wiele menadżerów pakietów.

Zastosowanie technologii zarządzania pakietami pozwala znacząco zmniejszyć próg wejścia wynikający z użycia wcześniej niewykorzystywanego oprogramowania. Pozwala on odejść od żmudnego procesu ręcznej instalacji zależności oraz konfiguracji środowiska. Dzięki menadżerom wszystko jest wykonywane automatycznie. Jeżeli w trakcie procedur nie wystąpi żaden problem, to pakiet, którego zleciliśmy instalację, powinien być od razu gotowy do działania. Jeżeli domyślna konfiguracja, jaka zostanie nam zapewniona w podczas działania menadżera pakietów, nie będzie dla nas odpowiednia możemy dokonać jej modyfikacji po procesie instalacji.

RPM, czyli RedHat Package Manager jest darmowym, open-source’owym menadżerem pakietów dla systemów z rodziny RedHat oraz SUSE, czyli m.in.:

- RedHat Linux
- CentOS
- Fedora
- openSUSE

RPM jest domyślnym manadżerem pakietów dla ww. dystrybucji. Obsługuje on pakiety w ramach formatu **.rpm**. Pakiety **.rpm** zawierają w sobie wiele ważnych elementów. Po pierwsze wewnątrz nich przechowywane są dane aplikacji, czyli: pliki wykonywalne, dokumentacja, testy, konfiguracja.

Kolejnym ważnym elementem są informacje o zależnościach, czyli innych wymaganych pakietach, które pozwalają na automatyzację procesu instalacji. W momencie, gdy któraś z zależności jest niespełniona menadżer pakietów stara się odnaleźć, w bazie danych pakietów, odpowiedni wpis, aby **pobrać** oraz **zainstalować** brakujące oprogramowanie. Trzecim ważnym elementem jest logika pakietu, która jest podstawą do realizacji akcji wykonywanych przez menadżer pakietów, dostarczona w postaci skryptów powłoki, np. **bash**, zaszytych wewnątrz pliku \*.rpm.

Listing 2.20 pokazuje przykładową zawartość pakietu, czyli moduł kernela **modułAplikacji**, plik w formacie **JSON** pozwalający na konfigurację aplikacji oraz plik wykonywalny **aplikacji**, natomiast listing 2.21 pokazuje przykładową logikę pakietu RPM w postaci skryptów shell. Skrypty te nie zawierają ani kopiowania, ani usuwania plików zawartych w pakiecie, proces ten odbywa się automatycznie na podstawie ścieżek ukazanych na Listingu ?? w trakcie instalacji/deinstalacji.

**Listing 2.20.** Przykładowa zawartość pakietu RPM

```
user@host:~# rpm -qpl pakiet.rpm
/ścieżka
/ścieżka/do
/ścieżka/do/konfiguracjaAplikacji.json
/ścieżka/do/aplikacji
/usr
/usr/lib
/usr/lib/modules
/usr/lib/modules/3.10.0-862
/usr/lib/modules/3.10.0-862/extra
/usr/lib/modules/3.10.0-862/extra/modułAplikacji.ko
```

**Listing 2.21.** Skrypty pakietu RPM

```
user@host:~# rpm -qp --scripts pakiet.rpm
preinstall program: /bin/sh
postinstall scriptlet (using /bin/sh):

#!/bin/sh
echo "Post-instajacja przykładowego pakietu"
echo "Przypisywanie uprawnień"
chmod 777 /ścieżka/do/konfiguracjaAplikacji.json
echo "Ładowanie modułu aplikacji"
/sbin/modrpobe modułAplikacji

preuninstall scriptlet (using /bin/sh):
#!/bin/sh
echo "Odinstalowywanie przykładowego pakietu"
echo "Usuwanie modułu aplikacji"
/sbin/rmmmod modułAplikacji

postuninstall program: /bin/sh
```

## 2.8. Technologie wirtualizacji i konteneryzacji

**Wirtualizacja**, czyli proces uruchamiania instancji wirtualnego systemu komputerowego odseparowanego od rzeczywistego systemu komputerowego oraz jego sprzętu (ang. hardware). Pozwala na uruchomienie **wielu różnych** systemów operacyjnych na jednym komputerze **jednocześnie**. Wykorzystywany przede wszystkim do separacji środowisk dla aplikacji, czy też całych systemów. Pozwala na uruchomienie oprogramowania nieprzystosowanego do naszego systemu operacyjnego, wystarczy utworzyć instancję maszyny wirtualnej z odpowiednim systemem operacyjnym. Aplikacje uruchamiane w takiej instancji zachowują się tak, jakby znajdowały się na **odseparowanym komputerze** z własnym, dedykowanym systemem operacyjnym, bibliotekami oraz innym oprogramowaniem. Dużym plusem jest pełna separacja instancji uruchomionych na systemie gospodarza. Procesy jednej instancji **nie mają wpływu** na drugą instancję [15].

Proces wirtualizacji odbywa się za pomocą oprogramowania, które nazywa się hipernadzorcą (ang. hypervisor). Odpowiada on za zapewnienie środowiska, które pozwoli na uruchomienie maszyny wirtualnej. Wyróżniane są dwa rodzaje hipernadzorców. Pierwsze z nich bazują na wspomaganiu procesu przez fizyczny sprzęt, co pozwala na częściowe ominięcie systemu operacyjnego gospodarza, dzięki czemu narzut na wydajność jest mniejszy, natomiast drugie bazują na rozwiązaniach aplikacyjnych, dzięki czemu można je uruchamiać bez wsparcia sprzętowego, natomiast są znacznie mniej wydajne.

Konteneryzacja jest procesem utworzenia odseparowanego kontenera, czyli ustandaryzowanej jednostki, która zawiera w sobie oprogramowanie oraz zależności wymagane do uruchomienia aplikacji, w celu której została utworzona [referencja edureka]. Kontenery są tworzone na podstawie obrazu, czyli wzorcowego środowiska, które zostało zamrożone w celu późniejszego odwzorzenia. W przypadku **Dockera** obrazy te są tworzone na podstawie tzw. **Dockerfile**. Wewnątrz takiego pliku zapisywane są informacje o krokach podejmowanych w celu utworzenia obrazu, np.:

- informacje o bazowym systemie operacyjnym
- informacje o zmiennych środowiskowych
- komendy menadżera pakietów w celu instalacji zależności

Informacje te są poprzedzone odpowiednimi słowami kluczowymi, np.: **ENV**, czy **RUN**. Listing 2.22 ukazuje przykładowy Dockerfile, którego użycie, za pomocą odpowiedniej komendy Dockera, spowoduje utworzenie obrazu bazującego na dystrybucji centos z zainstalowanym kompilatorem języka c++.

**Listing 2.22.** Przykładowy Dockerfile

```
FROM cern/cc7-base:latest

# requirements.txt from ↵
    https://gitlab.cern.ch/atlas-trt-dcs-ggss/external-dim-lib/blob/master/requirements.txt
COPY PYTHON/requirements.txt requirements.txt
# BOOST
# if you want to insert your own BOOST version to Docker image
# COPY <local_path_to_boost> /usr/local/boost
# if you want to install default BOOST from repositories (1.53.0)
# RUN yum -y install boost

ENV BOOST="/usr/local/boost"

RUN yum -y install cmake \
make \
unzip \
gcc \
gcc-c++ \
python3-pip \
git \
kernel-3.10.0-862.14.4.el7 \
kernel-devel-3.10.0-862.14.4.el7 \
rpm-build \
gsl \
gsl-devel \
```



```
libicu
```

```
RUN ln -s /usr/bin/pip3 /usr/bin/pip
```

```
RUN pip install -r requirements.txt
```



## 3. Stan początkowy projektu

### 3.1. Architektura

### 3.2. Budowanie

Niniejsza część pracy zawiera opis pierwotnego sposobu budowania aplikacji, wraz z zastosowanymi rozwiązaniami technologicznymi (struktura i zawartość plików CMake) oraz listą potencjalnych ograniczeń wynikających z dotychczasowego podejścia do budowania.

#### Struktura plików CMake

Projekt w swojej oryginalnej postaci budowany był za pomocą narzędzia CMake w wersji **2.8**. Wyróżnić można było jeden nadrzędny plik *CMakeLists.txt* znajdujący się w katalogu głównym projektu oraz pomniejsze pliki dla każdego z modułów. Rysunek przedstawia w uproszczeniu pierwotną strukturę projektu, z wyszczególnieniem plików odpowiedzialnych za jego budowanie.

**TODO:** tutaj dac jakis fajny rysunek

#### Obsługa bibliotek zewnętrznych

#### Ograniczenia pierwotnego systemu budowania

Pierwotna wersja projektu narzuca daleko idące ograniczenia na sposób jego budowania. Najważniejszym z nich jest brak bezpośredniej możliwości zbudowania pojedynczych komponentów projektu. Listing 3.1 przedstawia fragment oryginalnego pliku *CMakeLists.txt* znajdującego się w katalogu głównym projektu. **TUTAJ REF DO PRACY PLUTECKIEGO** Plik ten pozwala na zbudowanie trzech aplikacji wchodzących w skład oprogramowania projektu GGSS: *ggssrunner*, *dimCS* oraz opcjonalnie *ggsspector*. Jest to jedyny plik w całym projekcie zawierający wszystkie informacje konieczne do zbudowania wymienionych aplikacji - tzn. posiadający listę bibliotek, od których aplikacje te są zależne. Oznacza to, że niemożliwe jest zbudowanie aplikacji *ggssrunner* jedynie za pomocą dedykowanego jej pliku *CMakeLists.txt*. Zatem pomimo, iż struktura projektu jest zmodularyzowana jeśli chodzi o architekturę (oprogramowanie zostało

podzielone na biblioteki), to niemożliwe jest (w prosty sposób, za pomocą dostarczonych plików *CMakeLists.txt*) zbudowanie pojedynczych modułów projektu.

**Listing 3.1.** Fragment oryginalnego pliku *CMakeLists.txt* znajdującego się w katalogu głównym pierwotnej wersji projektu

```
# array with used libraries
set (PROJECTS
    logLib
    xmlLib
    utilsLib
    handleLib
    ThreadLib
    fifoLib
    FitLib
    OrtecMcbLib
    CaenHVLib
    ggssLib
    usbrmLib
    CaenN1470Lib
    mcaLib
    daemonLib
)

foreach (singleproject ${PROJECTS})
    parse_directory(${singleproject})
endforeach(singleproject)

# executables
add_subdirectory (_ggss) # ggssrunner binary
add_subdirectory (_dimCS) #dimCS binary
if (BUILD_GGSSPECTOR)
    add_subdirectory (_ggsspector) #ggsspector binary
endif()
```

Budowanie projektu za pomocą pliku, którego fragment przedstawia listing 3.1 opiera się na liście zależności przechowywanej w zmiennej *PROJECTS*. Umożliwia to stosunkowo łatwe rozszerzanie projektu o nowe biblioteki - wystarczy dopisać nazwę katalogu z biblioteką do listy zależności. Wadą tego rozwiązania jest natomiast brak możliwości wywnioskowania zależności zachodzących w projekcie. Listing 3.2 przedstawia plik *CMakeLists.txt* służący do budowania aplikacji *ggssrunner*. Na podstawie tych dwóch plików można jedynie wywnioskować, że aplikacja *ggssrunner* jest zależna od wszystkich bibliotek, których nazwy znaleźć można w zmiennej *PROJECTS*. Nie ma natomiast możliwości identyfikacji zależności między samymi bibliotekami. Takie podejście utrudnia zrozumienie struktury projektu, co bezpośrednio prowadzi do problemów z jego rozwojem.

**Listing 3.2.** Oryginalny plik CMakeLists.txt służący budowania aplikacji ggssrunner.

```
project (_ggss)
add_executable (ggssrunner main)
target_link_libraries (ggssrunner ${PROJECTS})
install(TARGETS ggssrunner RUNTIME DESTINATION bin)
```

### 3.3. Dostarczanie i uruchamianie

### 3.4. Kontrola wersji



#### **4. Stan docelowy projektu**





## 5. Ograniczenia dostępnej infrastruktury

Z uwagi na silny związek oprogramowania GGSS z infrastrukturą CERN oraz wymóg zapewnienia możliwości budowania projektu na należących do niej maszynach, przed autorami postawiony został szereg ograniczeń związanych z możliwymi do użycia technologiami oraz sposobem wykonywania pewnych operacji. Niniejszy rozdział stanowi opis najważniejszych z tych ograniczeń z uwzględnieniem ich wpływu na obronę przez autorów pracy ścieżkę rozwoju projektu.

### 5.1. Ograniczone uprawnienia w środowisku docelowym

### 5.2. Wersje kompilatorów i interpreterów

Dostępne wersje kompilatorów i interpreterów stanowią jeden z kluczowych czynników, który należy uwzględnić podczas wprowadzania zmian w istniejącym systemie, ponieważ definiują one możliwy do wykorzystania podzbiór technologii. W kontekście systemu GGSS ograniczenia te dotyczą przede wszystkim kompilatora języka C++ oraz interpretera języka Python.

#### Wersja kompilatora języka C++

Dostępna w ramach infrastruktury projektu wersja kompilatora języka C++ to **g++ (GCC) 4.8.5**. Wspiera ona w pełni standard C++11, czyli funkcjonalności takie, jak referencje do wartości, wyrażenia lambda czy zakresowa pętla `for` [16]. Wersja ta nie wspiera niestety nowszych wydań języka (C++14/17).

#### Wersja interpretera języka Python

Domyślną wersją Pythona jest **Python 2.7.5**, jednak dostępny jest również Python 3 (w wersji **Python 3.6.8**). Z uwagi na wspomniany wcześniej koniec oficjalnego wsparcia dla Pythona 2, który ma nadejść wraz z początkiem 2020 roku, naturalnym jest więc wybór wersji 3. Infrastruktura projektu posiada jednak znaczące braki jeśli chodzi o dostępne dla wersji 3 biblioteki zewnętrzne - domyślnie nie jest np. dostępna biblioteka *Beautiful Soup*, służąca do przetwarzania dokumentów w formacie HTML. Niektóre popularne biblioteki i frameworki (np. *PyTest* - wykorzystywany do przeprowadzania testów oprogramowania) nie są dostępne dla obu wersji

Pythona. Taka sytuacja wymusza więc wykorzystanie narzędzia *virtualenv* w celu ich instalacji w odizolowanym środowisku, nie mającym wpływu na infrastrukturę CERN-u.

### 5.3. Wersja narzędzia budującego CMake

Dostępna wersja narzędzia CMake stanowiła zdaniem autorów największe ograniczenie w czasie prac nad projektem. Na maszynach docelowych dostępna jest jedynie stara wersja **2.8.12.2**. Nowsza wersja (**3.14.6**) dostępna jest na niektórych z komputerów, jednak z uwagi na konieczność zachowania kompatybilności ze wspomnianymi maszynami docelowymi, nie było możliwe jej użycie. Stosowanie wersji o numerze niższym od **3.0** skutkuje szeregiem ograniczeń - brakuje w niej wielu funkcjonalności pozwalających na stosowanie ogólnoprzyjętych dziś praktyk, jak np. określenie zakresu wersji narzędzia CMake, w którym powinna mieścić się używana wersja, by projekt można było bez problemu zbudować, czy wsparcie dla instrukcji *target\_link\_directories* [17].

### 5.4. Związek projektu z wersją jądra systemu

## 6. Wykonane prace

6.1. Wykorzystanie funkcjonalności portalu Gitlab wspierających zarządzanie projektem

6.2. Migracja projektu do systemu kontroli wersji Git i zmiany w architekturze

6.3. Zastosowanie podejścia CI/CD

6.4. Zmiana sposobu budowania aplikacji

6.5. Budowanie i dystrybucja sterownika oraz aplikacji testującej

6.6. Maszyna wirtualna oraz konteneryzacja - Docker

6.7. Pomniejsze prace

6.7.1. Integracja bibliotek napisanych w języku C z aplikacją w C++

6.7.2. Integracja zewnętrznej biblioteki dynamicznej z użyciem narzędzia CMake

6.8. Dokumentacja projektu



## 7. Dalsza ścieżka rozwoju projektu

7.1. Wprowadzenie zautomatyzowanego systemu testowania projektu

7.2. Migracja do nowego standardu języka C++

7.3. Automatyzacja procesu publikowania produktu



## 8. Podsumowanie oraz wnioski

### 8.1. Statystyki projektu





## A. Dodatki/Appendixes

A.1. Adding new modules to the project using existing CMake templates

A.2. Preparing virtual machine to work as a runner



## Bibliografia

- [1] Wikipedia The Free Encyclopedia. *ATLAS experiment*. URL: [https://en.wikipedia.org/wiki/ATLAS\\_experiment](https://en.wikipedia.org/wiki/ATLAS_experiment) (term. wiz. 2019-12-07).
- [2] Bochenek M. „Budowa aparatury detekcyjnej i przygotowanie programu fizycznego przyszłych eksperymentów fizyki cząstek (ATLAS i LHCb na akceleratorze LHC i Super LHC oraz eksperymentu na akceleratorze liniowym ILC).” W: *Akademia Górniczo-Hutnicza im. S. Staszica. Wydział Fizyki i Informatyki Stosowanej. Raport Roczny 2008*. (2008).
- [3] Bjarne Stroustrup. *Język C++, Kompendium Wiedzy, Wydanie IV (The C++ Programming Language, 4th Edition)*. ul. Kościuszki 1c, 44-100 Gliwice: HELION S.A., 2014, s. 35–53.
- [4] Wikipedia The Free Encyclopedia. *C++14*. URL: <https://en.wikipedia.org/wiki/C%2B%2B14> (term. wiz. 2019-12-07).
- [5] Bartłomiej Filipek. *C++ 17 Features*. URL: <https://www.bfilipek.com/2017/01/cpp17features.html> (term. wiz. 2019-12-07).
- [6] *Dokumentacja biblioteki Boost w wersji 1.57.0*. URL: [https://www.boost.org/doc/libs/1\\_57\\_0/?view=categorized](https://www.boost.org/doc/libs/1_57_0/?view=categorized) (term. wiz. 2019-12-07).
- [7] Kitware. *About CMake*. URL: <https://cmake.org/overview/> (term. wiz. 2019-12-07).
- [8] Sandy McKenzie (KitwareBlog). *CMake Ups Support for Popular Programming Languages in Version 3.8*. URL: <https://blog.kitware.com/cmake-ups-support-for-popular-programming-languages-in-version-3-8/> (term. wiz. 2019-12-07).
- [9] Pablo Arias. *It's Time To Do CMake Right*. URL: <https://pabloariasal.github.io/2018/02/19/its-time-to-do-cmake-right/> (term. wiz. 2019-12-07).
- [10] Mark Lutz. *Python. Wprowadzenie. Wydanie IV (Learning Python, Fourth Edition by Mark Lutz)*. ul. Kościuszki 1c, 44-100 Gliwice: HELION S.A., 2011, s. 49–65.
- [11] Laurence Bradford. *What Should I Learn As A Beginner: Python 2 Or Python 3?* URL: <https://learntocodewith.me/programming/python/python-2-vs-python-3/> (term. wiz. 2019-12-07).

- 
- [12] Team Anaconda. *End of Life (EOL) for Python 2.7 is coming. Are you ready?* URL: <https://www.anaconda.com/end-of-life-eol-for-python-2-7-is-coming-are-you-ready/> (term. wiz. 2019-12-07).
- [13] *Bash Reference Manual*. URL: [http://www.gnu.org/software/bash/manual/bash.html#What-is-Bash\\_003f](http://www.gnu.org/software/bash/manual/bash.html#What-is-Bash_003f) (term. wiz. 2019-12-08).
- [14] Wikipedia The Free Encyclopedia. *Package manager*. URL: [https://en.wikipedia.org/wiki/Package\\_manager](https://en.wikipedia.org/wiki/Package_manager) (term. wiz. 2019-12-08).
- [15] opensource.com. *What is virtualization?* URL: <https://opensource.com/resources/virtualization> (term. wiz. 2019-12-08).
- [16] GCC Team. *Status of Experimental C++11 Support in GCC 4.8*. URL: [https://gcc.gnu.org/gcc-4.8/cxx0x\\_status.html](https://gcc.gnu.org/gcc-4.8/cxx0x_status.html) (term. wiz. 2019-12-07).
- [17] *What's new in in CMake*. URL: <https://cliutils.gitlab.io/modern-cmake/chapters/intro/newcmake.html> (term. wiz. 2019-12-07).