

**POLITECHNIKA ŚLĄSKA**  
**WYDZIAŁ INŻYNIERII MATERIAŁOWEJ**

**Kierunek: Informatyka Przemysłowa**  
**Profil praktyczny**  
**Rodzaj studiów: stacjonarne I stopnia**

**Projekt inżynierski**  
**Krzysztof MOCHOCKI**

**PROJEKT I IMPLEMENTACJA SYSTEMU  
RAPORTUJĄCEGO BRAKI POMIĘDZY  
SERWISAMI AGREGUJĄCYMI PUBLIKACJE  
NAUKOWE OPARTY O JĘZYK C++**

Design and implementation of  
reporting system for missing articles between science  
publications aggregation sites in C++

Kierujący pracą

Dr inż. Adrian SMAGÓR

Recenzent

Dr inż. Łukasz MALIŃSKI

**Katowice, czerwiec 2021 r.**



*Dziadkowi*

# 1. Spis treści

<b>1.</b>	<b>Spis treści .....</b>	<b>4</b>
<b>2.</b>	<b>Wstęp.....</b>	<b>6</b>
2.1.	<i>Inspiracja .....</i>	6
2.2.	<i>Cel .....</i>	6
<b>3.</b>	<b>Zakres pracy.....</b>	<b>7</b>
3.1.	<i>Przegląd literaturowo - technologiczny .....</i>	7
3.1.1.	<i>Podobne rozwiązania .....</i>	7
3.1.2.	<i>Aktualny model synchronizacji.....</i>	7
3.1.3.	<i>Dostępne źródła wiedzy w alternatywnych serwisach .....</i>	9
3.2.	<i>Przegląd wybranych technologii .....</i>	12
3.2.1.	<i>System operacyjny .....</i>	12
3.2.2.	<i>Język programowania .....</i>	12
3.2.3.	<i>Styl.....</i>	13
3.2.4.	<i>System kontroli wersji .....</i>	13
3.2.5.	<i>Dokumentacja.....</i>	13
3.2.6.	<i>Statyczna analiza kodu .....</i>	15
3.2.7.	<i>System budowy .....</i>	15
3.2.8.	<i>Środowisko deweloperskie .....</i>	16
3.2.9.	<i>Wykorzystane biblioteki .....</i>	16
3.3.	<i>Część projektowa.....</i>	19
3.3.1.	<i>Założenia projektowe .....</i>	19
3.4.	<i>Interfejs użytkownika.....</i>	21
3.5.	<i>Pobór danych.....</i>	28
3.5.1.	<i>Baza DOROBK.....</i>	28
3.5.2.	<i>Bazy ORCID oraz SCOPUS .....</i>	33
3.6.	<i>Unifikacja .....</i>	35
3.6.1.	<i>Reprezentacja obiektowa .....</i>	36
3.6.2.	<i>Generowanie obiektów wyższej abstrakcji.....</i>	38
3.7.	<i>Dopasowywanie.....</i>	42
3.8.	<i>Nadzorowanie przebiegu .....</i>	44
3.8.1.	<i>Zrównoleglenie przetwarzania danych.....</i>	45
3.8.2.	<i>Generowanie raportu .....</i>	47
3.9.	<i>Omówienie serializacji .....</i>	48
3.9.1.	<i>Zasada działania.....</i>	48
3.9.2.	<i>Problematyka.....</i>	51

3.9.3.	<i>Wady i zalezy</i>	52
3.10.	<i>Logowanie</i>	53
3.11.	<i>Testy</i>	55
3.12.	<i>Narzędzia i produkty uboczne</i>	56
3.12.1.	<i>Generowanie bibliotek</i>	56
3.12.2.	<i>Asysta z systemu kontroli budowy</i>	57
3.12.3.	<i>Udział w innych projektach</i>	60
3.13.	<i>Konfiguracja stanowiska</i>	61
<b>4.</b>	<b>Podsumowanie</b>	<b>63</b>
4.1.	<i>Perspektywy rozwoju</i>	63
4.2.	<i>Wnioski</i>	65
<b>5.</b>	<b>Bibliografia</b>	<b>66</b>
<b>6.</b>	<b>Spis ilustracji</b>	<b>69</b>

## 2. Wstęp

### 2.1. *Inspiracja*

Inspiracją dla niniejszego projektu inżynierskiego było dostrzeżenie problemu dotyczącego pracowników naukowych Politechniki Śląskiej, którzy, aby móc dzielić się wynikami swoich prac z resztą świata nauki, zmuszeni są eksponować swój dorobek w różnych agregatach publikacji naukowych. Niestety w związku z różną datą pojawienia się agregatorów oraz natłokiem innych zadań, ciężko jest dopilnować, aby wszystkie serwisy posiadały wszystkie publikacje.

### 2.2. *Cel*

Celem projektu jest dostarczenie narzędzia pozwalającego zautomatyzować cały proces wyszukiwania brakujących publikacji wśród znanych i obsługiwanych serwisów.

Wśród obsługiwanych portali, powinny znaleźć się te, które posiadają najwięcej zbiorów, oraz te, które cieszą się sporą popularnością. Dodatkowym kryterium jest możliwość dodania lub edycji danych w przeszukiwanym przez aplikację portalu. Przy założonych kryteriach wyłaniają się cztery portale:

- *DOROBEK* [1]
- *ORCID* [2]
- *SCOPUS* [3]
- *Web Of Science* [4]

Docelowo program, ma generować prosty w interpretacji raport w powszechnie znanym formacie, pozwalający na jednoznaczne wskazanie, zbiorów publikacji, które należy uzupełnić w poszczególnym portalu internetowym. Formatem spełniającym te warunki oraz dodatkowo zapewniający przenaszalność i wiele narzędzi do jego obsługi, jest otwarty format opracowany przez firmę *Microsoft* dla arkuszy kalkulacyjnych: *xlsx* [5]. Biorąc pod uwagę możliwość pojawienia się nowych podobnych serwisów internetowych, projekt aplikacji powinien również zapewniać łatwą rozszerzalność o nowe źródła danych.

## 3. Zakres pracy

### 3.1. *Przegląd literaturowo - technologiczny*

#### 3.1.1. Podobne rozwiązania

W domenie publicznej niestety nie udało się znaleźć podobnych rozwiązań, co może być zrozumiałe z uwagi na charakter korzystania z tego typu serwisów. Wnioskując na podstawie dokumentacji, dostarczonej przez wyżej wymienione portale, programowalny interfejs jest udostępniany do integracji z wewnętrznymi zasobami ośrodków naukowych.

Zasoby tego typu, jeżeli będą otwarte, to nie będą skoncentrowane na twórczości pracowników Politechniki Śląskiej, lecz na swoich naukowcach. Fakt ten uniemożliwia korzystanie z potencjalnych portali jako narzędzia do diagnozowania luk we wspieranych portalach, dla obcych badaczy.

#### 3.1.2. Aktualny model synchronizacji

W momencie rozpoczęcia prac brakowało narzędzi pozwalających na sprawną synchronizację publikacji naukowych pracowników Politechniki Śląskiej w internetowych bazach publikacji, takich jak: *DOROBK*, *SCOPUS*, czy *ORCID*, ponieważ narzędzia, które miały im to umożliwić nie zapewniały wystarczających możliwości.

Dla pracowników Politechniki Śląskiej, punktem wyjścia ręcznego porównania spisów swoich publikacji w internetowych bazach, jest baza *DOROBK*. Baza ta powinna zawierać wszystkie publikacje, pracowników Politechniki Śląskiej. Aby możliwe było w prosty i systematyczny sposób porównywać dane pomiędzy portalami potrzebna jest funkcjonalność prostego zbierania danych, czego baza *DOROBK* nie zapewnia.

Naturalnym podejściem osoby niepotrafiącej programować, byłoby ręczne kopiowanie tytułów i wklejanie ich do innych wyszukiwarek, celem weryfikacji ich istnienia. Temu podejściu nie sprzyja jednak mocno dynamiczny wygląd strony, który uniemożliwia sprawne zaznaczanie interesującego ciągu znaków kursorem.

Alternatywą dla pierwszego sposobu mogłoby być skorzystanie z generacji pliku *csv* [6], który jednak jest mocno niekompletny, między innymi przez brakujące numery *ORCID*. Identyfikatory te ułatwiają wyszukiwanie, ale dane zarówno w wspomnianym pliku, jak i na stronie są nierzadko zapisane w nieprzystępnej formie.

Często widywaną praktyką jest wpisywanie tytułu w języku polskim i nowożytnym po przecinku, kropce, średniku lub spacji, co generuje kolejny problem, a mianowicie niejednoznaczność w dalszych poszukiwaniach. Z logicznego punktu widzenia pojawiają się dwie różne wartości dla jednego nie tablicowego pola. Strona, oprócz pliku w formacie *csv* oferuje również generowanie pliku w formacie *rtf*, który względem poprzedniego pliku ma przewagę w postaci kompletnych danych.

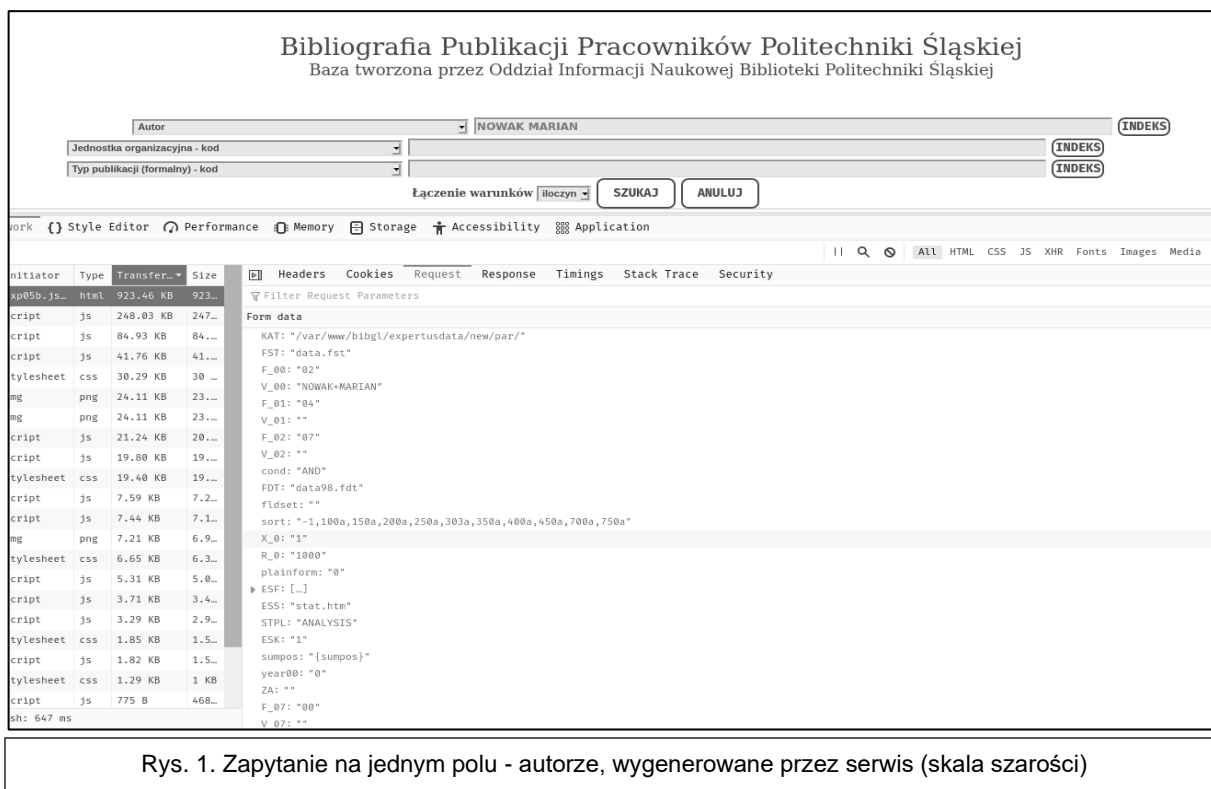
Największe trudności związane z automatycznym poborem danych, z bazy *DOROBK* jest brak możliwości zdalnego generowania plików *csv* i *rtf*, ponieważ są one generowane przez skrypty klienckie, co wymusza przetwarzanie gołego kodu strony w języku *HTML* wraz ze wszystkimi stylami i definicjami skryptów klienckich.

Domyślną postawą podczas komunikacji strony internetowej wraz z bazą danych, jest korzystanie z programowalnego interfejsu aplikacji (ang. *API – Application Programmable Interface*), za pomocą protokołu *REST*, czy *GRPC*. Można przypuszczać, że w przypadku tego serwisu, również zachodzi taka wymiana, co potwierdza szybka analiza ruchu sieciowego strony. Skorzystanie jednak z wyłoniętego zapytania, poprzez wyszukanie największego typu metody *POST* (Rys. 1), również nie napawa optymizmem, ponieważ zapytanie to, zwraca kawałek strony internetowej wygenerowanej po stronie serwera.

Ponadto zapytanie zbudowane przez stronę, wygląda na pozór na zaszyfrowane (Rys. 1), ponieważ w rzeczywistości, są to trudno identyfikowalne nazwy wprowadzonych pól. Stosowanie takiej praktyki sprawia, że wykorzystanie tego zapytania przez zewnętrzny serwis wymaga zarówno wcześniejszego zgadywania przez programistę za pomocą prób i błędów znaczenia poszczególnych pól formularza, a także żmudnego i wymagającego sporych zasobów przetwarzania dostarczonych danych w formacie *HTML*.

Wielokrotne próby komunikacji z administratorem strony zakończyły się kategoryczną odmową dostępu do jakiegokolwiek punktu dostępowego, dla osób





Rys. 1. Zapytanie na jednym polu - autorze, wygenerowane przez serwis (skala szarości)

trzech. Wymusza to na osobie zainteresowanej automatyzacją procesu weryfikacji spójności danych, w innych serwisach, na skorzystanie z przetwarzania danych uzyskanych ze źródła przeznaczonego dla użytkownika przeglądarki internetowej.

Kolejne serwisy, między innymi *ORCID* oraz *SCOPUS* zapewniają bardzo obszerne dokumentacje, z przykładami oraz dokładnymi objaśnieniami, w przeciwieństwie do bazy *DOROBK* są przyjaźnie nastawione do użytkownika i oferują zwięzły panel użytkownika.

### 3.1.3. Dostępne źródła wiedzy w alternatywnych serwisach

#### 3.1.3.1. ORCID

Serwis *ORCID* zapewnia programiście szerokie wsparcie w zakresie obsługi dostarczonych przez serwis narzędzi. Przekonać można się o tym, logując się na stronie, a następnie przechodząc do zakładki narzędzi deweloperskich (ang. *Developer Tools*), która znajduje się w poręcznym miejscu przy ikonie użytkownika w prawym górnym rogu. Po przejściu na witrynę deweloperską, strona od razu przedstawia przydatne hiperłącza, kierujące zainteresowanego do obszernej dokumentacji wystawionego interfejsu sieciowego. Istnieje możliwość, zarejestrowania swojej aplikacji. Daje to możliwość wykorzystania z jednego

z dostępnych protokołów podwyższających bezpieczeństwo komunikacji: *OAuth* lub *OAuth 2.0* [7]. Skorzystanie z bezpieczniejszego połączenia jest wymagane przez serwis, aby móc przeprowadzić proces logowanie i wykonywać operacje na koncie poza stroną internetową.



Rys. 2. Podgląd specyfikacji technicznej dostarczonej przez serwis ORCID (skala szarości)

Dzięki wykorzystaniu otwartego i sprawdzonego standardu tworzenia dokumentacji *Swagger* [8], (Rys. 2) dokumentację interfejsu sieciowego czyta się szybko z uwagi na znany format. Dodatkowo twórcy umożliwili przetestowanie punktów końcowych interfejsu, co umożliwia programiście zaplanowanie sposobu przetwarzania przychodzących danych. Kolejną istotną rzeczą jest wersjonowanie punktów końcowych. Daje to programiście spore poczucie bezpieczeństwa, że model przychodzących danych lub wygląd zapytania, nie zostanie nagle zmieniony. Polityka serwisu polega na dodaniu kolejnej wersji interfejsu sieciowego w przypadku zmiany czegokolwiek w budowie zapytań lub w modelu przychodzących danych.

### 3.1.3.2. SCOPUS

Serwis *SCOPUS* z wyjątkiem wersjonowania dostarcza te same równie przyjemną interaktywną dokumentację (Rys. 3), co serwis *ORCID*. Jednakże w porównaniu do poprzedniego zapewnia konkretne przypadki użycia swojego interfejsu z wykorzystaniem różnych języków, między innymi *Python*, czy *JavaScript*. Ponadto interaktywna dokumentacja generuje od razu polecenia *cURL*, które można wkleić do linii poleceń w systemie *GNU/Linux* i szybko sprawdzić sposób działania dostępnych punktów końcowych.

Elsevier Search APIs - interactive documentation

Detailed interface documentation is located here.

Affiliation\_Search : Affiliation Search API

Show/Hide | List Operations | Expand Operations

Author\_Search : Author Search API

Show/Hide | List Operations | Expand Operations

GET /search/author

Author Search API

Implementation Notes

Author search exposes interfaces associated with Scopus-based author profiles.  
API key in this example was setup with authorized CORS domains.

Response Class (Status 200)

No response was specified

Model | Model Schema

Response Content Type

application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
query	authlast(Einstein) and authfirst(Albert) and affi	Search query string	query	string
apiKey		Your API key	query	string
httpAccept		Requested content type, overrides HTTP header value	query	string
insttoken		Specification for authorization, institution authtoken	query	string
access_token		Specification for active session, secured authtoken	query	string

Try it out!

Engineering\_Village\_Search : Engineering Village Search API

Show/Hide | List Operations | Expand Operations

Rys. 3. Podgląd specyfikacji technicznej dostarczonej przez serwis SCOPUS (skala szarości)

## 3.2. Przegląd wybranych technologii

### 3.2.1. System operacyjny

Manjaro [9] – dystrybucja systemu operacyjnego *GNU/Linux* bazująca na dystrybucji *Arch Linux* [10], który słynie ze ścisłego trzymania się reguły KISS (ang. *Keep It Simple, Stupid*). Pośrednio wynika z tej zasady, jak wygląda zarządzanie pakietami w tej dystrybucji. Mimo istnienia menadżera pakietów *pacman*, spora część oprogramowania wymaga ręcznego budowania przez użytkownika na swoim urządzeniu. Wymóg ten miał spory wpływ na wybór zależności do samego programu, ponieważ wybór bibliotek dedykowanych pod aktualny system jest bardzo mały. Wymusza to na programiście wybór bibliotek i technologii, które są możliwe do zastosowania również na nietypowych systemach oraz cechujące się wysoką przenaszalnością. Implikuje to poprawienie przenaszalności całego projektu na różne platformy.

### 3.2.2. Język programowania

C++ [11] – elastyczny, szybki i wydajny język programowania, dający programiście dużo możliwości oraz stawiający sporo wyzwań podczas pisania w nim. Sam, spadkobierca języka C, jest dość stary i posiada wiele wersji, które, szczególnie te współczesne, mocno się od siebie różnią, stąd konieczność na wstępie jej sprecyzowania. Niniejszy projekt używa najnowszej wersji – 20, oznaczanej jako C++20, lub C++2a w przypadku wydań eksperymentalnych

Jednakże wersja języka nie jest jedyną rzeczą konieczną do sprecyzowania przy jego omawianiu. Konieczny jest jeszcze wybór kompilatora, który będzie zamieniał go na język maszynowy. Na wybranej platformie, dostępne są dwa popularne kompilatory: *gcc* oraz *clang*. Ostatecznym czynnikiem mającym wpływ na wybór kompilatora było wcześniejsze doświadczenia twórcy w pracy z *gcc*.

Jednym z wielu argumentów za wyborem tego języka jest jego wyżej wspomniana elastyczność. Jest to jeden z niewielu języków dający możliwość korzystania z wielu paradygmatów. Program tutaj opisywany, nie skupia się na kurczowym trzymaniu się jednego z nich, lecz oscyluje wokół trzech: obiektowego, funkcyjnego oraz [szablonowego] meta-programowania.

### 3.2.3. Styl

Notacja węzowa (ang. *sneak case*) [12] – zaraz obok wielbłądziej [13] jeden z najbardziej rozpoznawalnych stylów używanych w kodzie. Wyborem stojącym za skorzystaniem z tej notacji jest chęć dostosowania się do tej, dostarczonej z biblioteki standardowej. Jedynym odstępstwem od jej ścisłego przestrzegania, są nazwy argumentów w szablonach, gdzie również w bibliotece standardowej można spotkać się z notacją wielbłądzą.

Oprócz samej notacji warto jest zadbać o jednolite formatowanie w całym projekcie, co poprawia czytelność i zwiększa wrażenie spójności wszystkich jego elementów. Celem zapewnienia zgodnego formatowania, należało zdecydować się na dedykowane, oprogramowanie. Najpopularniejszym, otwartym, rozwiązaniem jest program z pakietu *clang* o nazwie *clang-tidy* [14]. Konfiguracja jest bardzo prosta, oparta na pojedynczym pliku tekstowym. Po uruchomieniu, program rekurencyjnie wyszukuje pliki według zadanego wyrażenia regularnego, a następnie podmienia na sformatowane.

### 3.2.4. System kontroli wersji

git [15] – popularny program, służący do kontroli wersji. Przy bardziej złożonych projektach jest to narzędzie obowiązkowe, z uwagi na łatwą możliwość powrotu do poprzednich wersji, możliwość łatwej współpracy z wieloma osobami, a także łatwą kontrolę zależności w postaci podmodułów. W połączeniu z serwisem *GitHub*, służącym jako zdalne repozytorium, zapewnia bezpieczny ekosystem.

### 3.2.5. Dokumentacja

Doxygen [16] [17] – narzędzie do automatycznego generowania dokumentacji w formie strony internetowej (Rys. 5), plików w formacie *man* lub *rtf*. Korzyścią tego rozwiązania jest brak konieczności ręcznego tworzenia dokumentacji w osobnych plikach. Cała treść dokumentacji jest zapisana w formie komentarzy w całym kodzie źródłowym (Rys. 4). Dodatkowo, wbudowane programy, do wspomaganie programisty, takie jak *IntelliSense* korzystają z tych komentarzy, aby móc wyświetlić pomocne informacje, podczas tworzenia nowego kodu.

```

/**
 * @brief alternative to asertion
 *
 * @tparam _ExceptionType exception to throw if check failed
 * @tparam __log_pass if set to true, communications about positive check are displayed
 */
You, seconds ago | 1 author (You)
template<template<typename Msg> typename _ExceptionType = exception, bool __log_pass = false>
requires supported_exception<_ExceptionType> struct require :
    Log<require<_ExceptionType, __log_pass>>
{
    using Log<require<_ExceptionType, __log_pass>>::get_logger;

    /**
     * @brief constructor is used as operator() for handy usage
     *
     * @tparam MsgType deduced type, of message
     * @tparam ExceptionArgs variadic type of additional argument that are passed to exception constructor
     * @param _check value to be checked
     * @param msg message to pass to exception
     * @param argv optional exception arguments
     */
    template<typename MsgType, typename ... ExceptionArgs>
    explicit require(const bool _check, const MsgType& msg = "no message provided",
        ExceptionArgs&& ... argv)

```

Rys. 4. Fragment kodu z komentarzami do wygenerowania dokumentacji

## Constructor & Destructor Documentation

### ◆ require()

```
template<template< typename Msg > typename _ExceptionType = exception, bool __log_pass = false>
```

```
template<typename MsgType, typename... ExceptionArgs>
```

```
core::exceptions::require< _ExceptionType, __log_pass >::require ( const bool      _check,
                                                                    const MsgType &   msg = "no message provided",
                                                                    ExceptionArgs &&... argv
                                                                    )
```

constructor is used as operator() for handy usage

#### Template Parameters

**MsgType** deduced type, of message

**ExceptionArgs** variadic type of additional argument that are passed to exception constructor

#### Parameters

**\_check** value to be checked

**msg** message to pass to exception

**argv** optional exception arguments

Rys. 5. Fragment wygenerowanej strony internetowej

### 3.2.6. Statyczna analiza kodu

GRADE ^	FILENAME ^	ISSUES v	DUPLICATION ^
A	libraries/demangler/.../libraries/html_scalpel/html_scalpel.h	0	0
A	libraries/network/.../libraries/orcid_adapter/orcid_adapter.h	0	0
A	tests/include/antybiurokrata/tests/demangler.test.h	0	0
A	libraries/demangler/include/.../libraries/demangler/demangler.h	0	0
A	libraries/network/src/network.cpp	0	0

Rys. 6. Raport statycznej analizy kodu (skala szarości)

codacy [17] – serwis internetowy świadczący usługi oceny kodu pod kątem użytych wzorców, bezpieczeństwa, redundancji kodu czy stosowania niedozwolonych praktyk programistycznych. Automatycznie wykrywa użyte języki, dokonuje ich statycznej analizy, a następnie generuje raport (Rys. 6), informujący o wykrytych błędach.

Dodatkowo serwis daje możliwość skorzystania z generowanej etykiety, możliwej do umieszczenia w pliku *README.md*, która zachęca innych użytkowników serwisu *GitHub* do uczestnictwa w dobrze napisanym kodzie.

### 3.2.7. System budowy

cmake [18] – otwarte, wieloplatformowe narzędzie do kontroli przebiegu budowy projektu oraz planowania testów. Oprogramowanie wykonuje instrukcje na podstawie przygotowanych plików, napisanych w dedykowanym języku skryptowym, za pomocą, którego można definiować cele do zbudowania w postaci bibliotek oraz plików wykonywalnych.

Dostarczony język daje również możliwość tworzenia niestandardowych celów budowy, co daje spore możliwości w zakresie tworzenia narzędzi wspomagających proces budowy czy proces wytwarzania kodu.

W trakcie tworzenia niniejszej aplikacji została wydana nowsza wersja programu *cmake* – 3.20. Aplikacja natomiast używa wersji 3.19.

### 3.2.8. Środowisko deweloperskie

Visual Studio Code [19] – otwarty, wieloplatformowy edytor tekstowy zawierający zestaw narzędzi wspierających programistę. Jego mocnymi stronami jest ilość wtyczek dostępnych bezpośrednio z edytora, tworzonych przez społeczność. Dzięki nim edytor zapewnia wsparcie nawet najbardziej egzotycznym językom, zarówno w zakresie podpowiadania składni, debugowania, kontroli wersji czy budowania i uruchamiania.

### 3.2.9. Wykorzystane biblioteki

#### 3.2.9.1. Wspierająco-narzędziowa

boost [20] – otwarta, wieloplatformowa biblioteka narzędziowa, lustrzana do biblioteki standardowej, jednocześnie rozszerzająca ją o wiele funkcjonalności. W przypadku większych projektów jest to, niemalże obowiązkowa pozycja na liście bibliotek.

Argumentem stojącym za wykorzystaniem tej biblioteki jest również powszechność jej użycia w różnych projektach, co sprawia, że najczęściej znajduje się już zainstalowana w systemie. Wadą tej biblioteki, jest wymagający próg wejścia, zadany przez autorów. Przy domyślnych ustawieniach biblioteka jest kompilowana w całości, co przy słabszych maszynach, może okazać dość długim procesem.

#### 3.2.9.2. Testowa

boost-ext/ut [21] – eksperymentalna część wcześniej wspomnianej biblioteki *boost*, zawierająca pakiet narzędzi do tworzenia testów, wraz z raportowaniem. Biblioteka jest stworzona w zgodzie ze standardem C++17, dając możliwość pisania testów korzystając z dobrodziejstw najnowszych rozwiązań biblioteki standardowej.

Biblioteka posiada możliwość organizacji kodu, dzięki zaimplementowanym zestawom (ang. *suites*). Wewnątrz jednego zestawu można umieścić różne konfiguracje aplikacji, a następnie napisać różne przypadki testowe (ang. *test cases*), celem przetestowania założonych ustawień. Ciekawe jest podejście autorów, do sposobu deklaracji testów, które polega na używaniu literałów ciągów znakowych.



W samych testach jako sprawdzeń, używa się bibliotecznych funkcji. Są one statycznie rozwiązywalne, z uwagi na oparcie o szablony, przeciwieństwie do biblioteki matki, która to używa makr z języka C. Takie rozwiązanie powoduje wykrycie części błędów, jeszcze na etapie kompilacji.

### 3.2.9.3. Sieciowa

drogoncpp [22] – otwarta, sieciowa biblioteka napisana w nowoczesnym standardzie języka C++, dedykowana do pisania aplikacji serwerowych, jednakże równie dobrze sprawdza się jako biblioteka kliencka.

Posiada pełne wsparcie dla rozwiązań wielowątkowych, a także posiada wsparcie dla korutyn. Bliskie trzymanie się biblioteki standardowej sprawia, że integracja z dostępnym interfejsem aplikacji jest bezproblemowa, a użytek intuicyjny. Użytkowanie biblioteki poprawia również dość szeroka dokumentacja, a także dostarczone przez społeczność, seria przykładów użycia

Wybierając tą bibliotekę konieczne jest również pobranie jej zależności, w postaci biblioteki o nazwie *tantor*. Biblioteka ta zawiera szereg kolekcji zsynchronizowanych oraz wiele narzędzi ułatwiających programowanie wielowątkowe. Instalacja jednak nie jest problematyczna, ponieważ występuje jako pod moduł biblioteki *drogoncpp* i jest dociągana automatycznie.

Dodatkowo decydując się na zainstalowanie w systemie biblioteki *drogoncpp*, zostaje dodane narzędzie umożliwiające szybkie tworzenie projektów aplikacji serwerowych. Narzędzie nazywa się *drogon-cli*. Okazało się ono bardzo pomocne w momencie tworzenia plików skryptowych dla systemu budowy *cmake*. Brak wpisów w dokumentacji biblioteki, jak należy podejść do tego zagadnienia, był zaskakujący. Rozwiązaniem okazało się wygenerowanie szablonowego projektu, a następnie użycie gotowych fragmentów plików wsadowych programu *cmake*.

#### 3.2.9.4. Wspierająca proces logowania

rang [23] – otwarta, nagłówkowa biblioteka oferująca międzyplatformowe, strumieniowe wsparcie dla kolorów w konsoli. Jest to dość niedoceniony aspekt logowania, jednakże zważając na ludzką wrażliwość na kolory, jest to pomocna funkcjonalność, redukująca szansę pominięcia ważnego błędu, który pojawił się w logach. Dodatkowo biblioteka jest zaprojektowana, by ściśle współpracować ze strumieniami z biblioteki standardowej, co dodatkowo zwiększa komfort pracy.

#### 3.2.9.5. Graficzna

Qt [24] – rozbudowana, platforma programistyczna (ang. *framework* [25]) graficzna, dedykowana dla języków C++ oraz *Python*. Zawiera dwojaki sposób opisu interfejsu użytkownika. Pierwszy, wykorzystany w tym projekcie, z wykorzystaniem formatu *xml*, który jest generowany za pomocą dołączonego edytora graficznego. Drugi, bardziej zaawansowany, wykorzystuje autorski język *QML*, który dobrze sprawdza się w przypadku rozwiązań wbudowanych, oraz mobilnych.

Do wersji 5.4 biblioteka oferowała również wsparcie dla wewnętrznego systemu kontroli budowy – *qmake*, jednakże wraz z najnowszą wersją 6.0 wsparcie to zostało zarzucone. Aktualnie rekomendowanym podejściem jest korzystanie z obecnego w tym projekcie nadzorcy budowy – *cmake*.

#### 3.2.9.6. Obsługująca rozszerzenie XLSX

QtXlsxWriter [26] – otwarta, rozbudowana biblioteka, oferująca obsługę plików w formacie *xlsx*. Niestety została stworzona w momencie pełnego wsparcia dla systemu budowy *qmake*, który aktualnie jest zarzucany kosztem *cmake* w najnowszej wersji platformy graficznej Qt. Rozwiązanie dostarczył jeden z członków społeczności. Użytkownik *dand-oss* stworzył, jeszcze nierozstrzygniętą, prośbę dołączenia jego zmian. Są one znaczące dla tego projektu, ponieważ dodają pełne wsparcie dla systemu budowy *cmake*. Jest to również powód, dla którego ta biblioteka, jako pod moduł, nie korzysta z oficjalnego repozytorium autora biblioteki.

### **3.3. Część projektowa**

#### **3.3.1. Założenia projektowe**

##### **3.3.1.1. Skalowalność**

Jednym z głównych problemów, ujednolicenia stanu wszystkich baz jest ich mnogość. Ilość aktualnych źródeł jest spora, z uwagi na bardzo dynamiczny rozwój nauki w ostatnich dekadach. Można bezpiecznie założyć, że będą powstawać nowe źródła danych, zarówno masowe (na przykład *ORCID*), czy mocno specjalistyczne (na przykład *International Journal of Minerals, Metallurgy and Materials*).

Prawidłowość ta, wymusza na aplikacjach chcących utrzymać wszystkie źródła w takim samym stanie, konieczność ciągłej rozbudowy. Stworzenie aplikacji, która będzie odpowiadać na takie wyzwanie, wymaga zaprojektowania oraz zaimplementowania odpowiedniej architektury.

Cechą aplikacji, która definiuje zdolność do rozszerzania, nazywa się skalowalnością.

##### **3.3.1.2. Przyjazna rozwojowi**

Rozwiązanie architektoniczne problemu skalowalności to podstawa pod przyszły rozwój aplikacji, bardzo ważny, lecz wciąż to tylko baza. Żeby rozwój przyszłych funkcjonalności aplikacji był sprawny i nie wymagał pełnej wtórnej analizy całego kodu, potrzebne jest stworzenie odpowiedniego środowiska w samym kodzie.

Czynnikiem, który ma zdecydowany wpływ na atmosferę w kodzie źródłowym jest utrzymana dokumentacja. Raport z Norweskiej Politechniki w Trondheim [27], wskazuje, że osoby posiadające wyłącznie kod źródłowy potrzebowały o 21,5% więcej czasu, aby zrozumieć analizowany program, niż ich koledzy, którzy dodatkowo posiadali dokumentację. Ponadto, jak wskazuje raport, zrozumienie źródeł w grupie z dostępną dokumentacją było statystycznie znaczące.

Kolejne przytoczone badanie [28], tym razem z osiemnastej konferencji na temat odwrotnej inżynierii z 2011 roku (ang. *2011 18th Working Conference on Reverse Engineering*) potwierdzają również pozytywny wpływ diagramów UML na efektywność podczas pracy z wzorcami projektowymi.

### 3.3.1.3. Współczesny

Język C++ w wersji C++20 dodał sporo nowych rozwiązań językowych, które usprawniają proces wytwarzania kodu, oraz wpływają na ilość kodu generowanego podczas kompilacji. Dzieje się tak dzięki prężnemu rozwojowi meta programowania, w ramach biblioteki standardowej jako bardzo popularnej metody wytwarzania oprogramowania, w ramach języka C++ w ciągu ostatnich paru lat. Jest to ważny krok w rozwoju języka, ponieważ uproszczenie programowania w paradygmacie meta programowania potrafi o wiele zwiększyć czytelność kodu oraz znacząco go uprościć.

Przed zmianami dokonanymi w standardach od C++11 do C++20, podejście do tego paradygmatu było dość mocno sceptyczne, szczególnie z uwagi na spore trudności w tworzeniu i rozwijaniu kodu. Dobrze określił to Herb Sutter, członek komisji standaryzacyjnej języka C++, w wywiadzie z 2011 roku dla kanału 9 [29], na prośbę o skomentowanie wyjścia, czterostronicowego błędu z procesu kompilacji, odrzekł, że jest ono „barokowe”. Na domiar złego, współczesne narzędzia wspomagające programistów, często miewają problemy z pełnieniem swojej roli w podpowiadaniu składni, przy sporym zagnieżdżeniu szablonów, co dodatkowo utrudnia rozwój oprogramowania opartego o ten sposób pisania kodu.

Standard C++20 jest pod tym kątem rewolucyjny, ponieważ razem z nim w języku pojawiła się konstrukcja konceptów [30], które w wypadku błędu jasno wskazują na miejsce i powód błędu inscenizacji szablonu. W związku z między innymi, powyższymi zmianami, które są kluczowe w optymalizacji kodu, istnieje konieczność trzymania się najnowszych wersji języka C++.

Trend ten dzielają również autorzy bibliotek, co sprawia, że znalezienie biblioteki opartej o najnowsze rozwiązania zawarte w języku nie jest, aż tak wymagające. Spore zaangażowanie, ze strony obszernej społeczności sprawia, że niejednokrotnie była konieczność aktualizacji wersji biblioteki z uwagi na nowe wydanie.

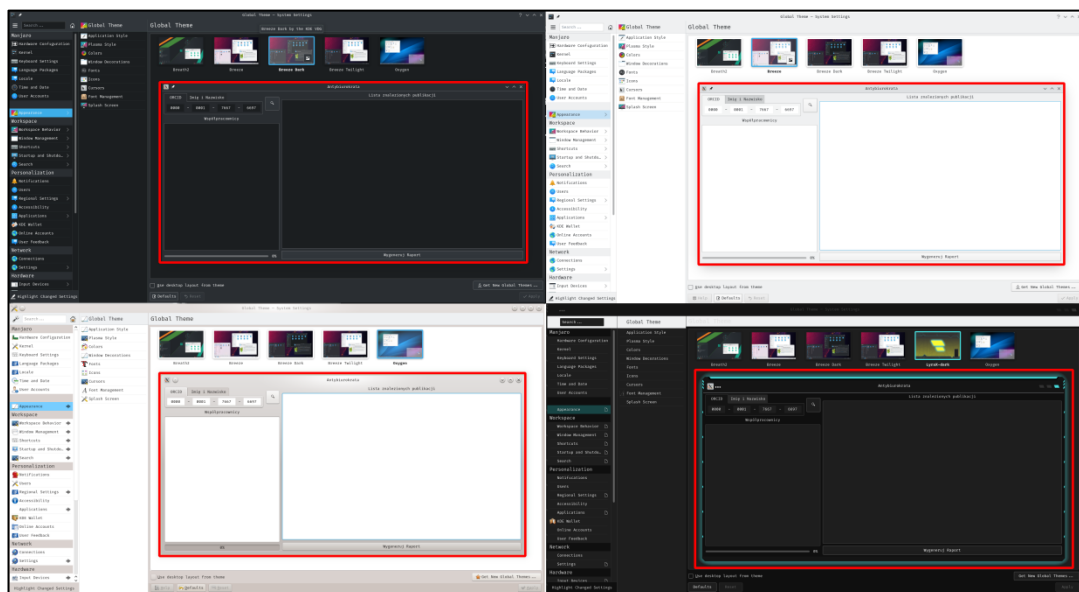
Dodatkowo, z uwagi na wyżej wymienione możliwości, kod został pozbawiony wszelkich makr, uznanych w obliczu nowych możliwości języka C++, jako zaszłości języka C, utrudniające utrzymanie kodu oraz zmniejszające jego czytelność.

### 3.3.1.4. Przystępne wyjście

Zrozumiałe i przejrzyste dla użytkownika wyjście programu jest jedną z kluczowych kwestii. Szczególnie, że jednym z celów tej aplikacji, jest redukcja czasu potrzebnego na zrównanie wielu baz do tego samego stanu. Wyjście nie powinno być w formacie nieprzyjaznym użytkownikowi lub formacie wymagającym płatnych programów do jego obsługi. Zmniejszyłoby to grono chętnych do skorzystania z aplikacji. Dodatkową cechą formatu powinna być jego przejrzystość, wynikająca z celu, jaki ma on zaspokajać. W tym przypadku, będzie to możliwość filtrowania, pod względem obecności danej publikacji w wybranej bazie. Możliwość posortowania danych, będzie miała niemały wpływ na przejrzystość raportu.

## 3.4. Interfejs użytkownika

Graficzna część aplikacji to zespół dwóch okien. Pierwsze główne oraz drugie służące, jako dialogowe, do wyświetlania szczegółów. Dzięki wykorzystaniu środowiska Qt, interfejs graficzny, w przypadku uruchomieniu na systemie operacyjnym *GNU/Linux* z nakładką graficzną *KDE*, dostosowuje się stylem do wybranego przez użytkownika motywu systemu. Daje to wrażenie integracji z systemem oraz w przypadku wybrania motywu ciemnego nie jest to zagrożeniem dla oczu użytkownika. Działa również z zewnętrznymi motywami (Rys. 7).



Rys. 7. Porównanie wyglądu w zależności od wybranego motywu systemu

Standardem dzisiejszych czasów jest responsywność interfejsu użytkownika, wymusza to zrezygnowanie ze sztywnych rozmiarów poszczególnych elementów interfejsu, na rzecz dozwolonych przedziałów. Celem zachowania spójności w organizacji elementów graficznych, konieczne jest zastosowanie kontenerów oraz dystansów ograniczających względne położenie poszczególnych części interfejsu. Środowisko Qt zapewnia sporo różnych kontenerów, usprawniający cały proces, jednakże działanie części z dostarczonych nie jest deterministyczne, co wymusza projektowanie interfejsu w oparciu o serię prób i błędów.

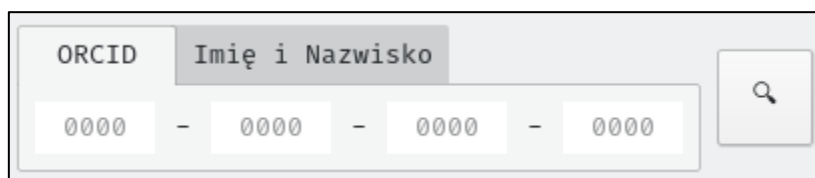
Interfejs użytkownika został zaprojektowany z zachodnio-europejskim modelem postrzegania kierunku ciągłości. Przepływ (ang. *flow*) obsługi aplikacji jest ułożony w kolejności z lewej do prawej, oraz z góry na dół. Model taki, został wybrany po zapoznaniu się ze wskazówkami firmy *Microsoft* na temat tworzenia interfejsu użytkownika [31]. Stosując się również do tych zaleceń, została ułożona kolejność tabulacji, która umożliwia obsługę niemalże całego interfejsu wyłącznie za pomocą klawiatury. Wynikiem zastosowania się do wskazówek jest uformowanie się dwupodziału w aplikacji oraz czterech stref, gdzie każda posiada konkretną funkcję i jest naturalnym przedłużeniem poprzedniej.

#### **3.4.1.1. Panel wyszukiwania**

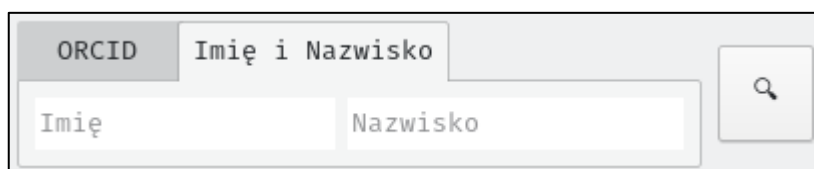
Pierwsza ćwiartka umiejscowiona w lewej górnej części głównego okna programu. Oprócz przycisku umiejscowionego po prawej stronie, posiada dwa tryby widoku, oba z polami na wejście od użytkownika. Domyślny widok (Rys. 8), oczekuje od użytkownika zadanie numeru *ORCID*, który ma zostać poddany komparacji przez algorytm wewnętrzny aplikacji.

Wejście składa się z czterech pól, których uzupełnianie powoduje przechodzenie do kolejnych komórek. Mechanizm smukłego przechodzenia, nie jest funkcjonalnością oferowaną przez środowisko Qt i wymagała ręcznego zaimplementowania. Ta sama kwestia dotyczy walidacji wprowadzonych danych. Mimo istnienia w graficznym edytorze interfejsu użytkownika opcji, wymuszających na użytkownika podanie wyłącznie liczb, pole mimo to przyjmuje dowolne znaki. Funkcjonalność automatycznego przechodzenia do kolejnych pól wejściowych identyfikatora *ORCID*, jest sprzężona z modelem weryfikacji i automatycznej korekty danych wejściowych. Implementacja ciągłości czterech pól była również konieczna,

aby użytkownik miał możliwość wklejenia numeru *ORCID*. Brak takiej funkcji, spotkałoby się najprawdopodobniej z irytacją i koniecznością ręcznego przepisania danych ze strony wprowadzającego dane.

Panel wyszukiwania z dwiema zakładkami: "ORCID" i "Imię i Nazwisko". Zakładka "ORCID" jest aktywna. W polu tekstowym widoczny jest numer "0000 - 0000 - 0000 - 0000". Po prawej stronie znajduje się przycisk z ikoną lupy.

Rys. 8. Domyślny widok panelu wyszukiwania

Panel wyszukiwania z dwiema zakładkami: "ORCID" i "Imię i Nazwisko". Zakładka "Imię i Nazwisko" jest aktywna. W polu tekstowym widoczne są etykiety "Imię" i "Nazwisko". Po prawej stronie znajduje się przycisk z ikoną lupy.

Rys. 9. Alternatywny widok panelu wyszukiwania

Drugi widok (Rys. 9) umożliwia wyszukanie osoby za pomocą imienia i nazwiska. Nie posiada tak zaawansowanej walidacji, jak poprzedni widok, ponieważ w tym wypadku wystarczyło wymuszenie polskiej lokalizacji w polach wejściowych. Jedyną restrykcją w przypadku tych pól jest ograniczenie długości imienia, do nieco dłuższego niż najdłuższe trzynastoliterowe polskie imię: Wierchosława.

Ustawienie domyślnej formy wyszukiwania jako identyfikator *ORCID*, nie jest przypadkowe i jest związane z tworzeniem zapytań do wspieranych baz. W przypadku podania imienia i nazwiska, wątki odpowiedzialne za odpytanie zewnętrznych serwisów muszą poczekać na wątek odpytujący bazę *DOROBK*, który to podane imię i nazwisko tłumaczy na numer *ORCID*. Po jego wyłuskaniu kolejne wątki rozpoczynają pracę. W przypadku podania identyfikatora, taki problem nie występuje.

Ostatnim elementem tej części interfejsu jest przycisk do rozpoczęcia procesu szukania. Zawiera on pojedynczy symbol lewostronnej lupy, oznaczony numerem kodowym *U+1F50D*. Zastosowanie ikony jako opisu przycisku ma dwie funkcjonalności. Pierwszą z nich jest łatwość w utrzymaniu proporcji przycisku, w zamierzonej formie kwadratu. Kolejnym aspektem jest ułatwienie przyszłego,

potencjalnego procesu lokalizacji, poprzez zmniejszenie pól posiadających napisy. Dodatkowo przycisk, posiada skrót klawiszowy – *SHIFT* + *ENTER*.

#### 3.4.1.2. Lista współpracowników



Rys. 10. Panel współpracowników z przykładowymi danymi (skala szarości)

Imiona i nazwiska wraz identyfikatorami *ORCID*, są wyświetlane w trakcie procesu wyszukiwania (Rys. 10), jako efekt uboczny procesu przetwarzania danych z bazy *DOROBK*. Pierwotne założenia zakresu prac nie zakładały istnienia tej części interfejsu, jednakże pomysł okazał się na tyle przyjazny dla użytkownika, że znalazł się w ostatecznej wersji aplikacji.



Zakończenie wyszukiwania odbywa się na dwa sposoby. Pierwszym z nich jest pasek postępu, który wskazuje postęp procesu poboru i przetwarzania danych. Drugim indykatorem zakończenia przetwarzania jest zmiana barw w interfejsie spowodowana ponownym aktywowaniem kluczowych elementów interfejsu po zakończeniu obróbki danych. Pasek postępu jest używany przez dwie kluczowe funkcjonalności programu, czyli przez część związaną z przetwarzaniem oraz raportowaniem. Ciężko dostrzec aktualizację paska postępu w przypadku używania go przez część raportową i jest to spowodowane bardzo szybkim procesem generowania raportu, nawet na sporej ilości danych.

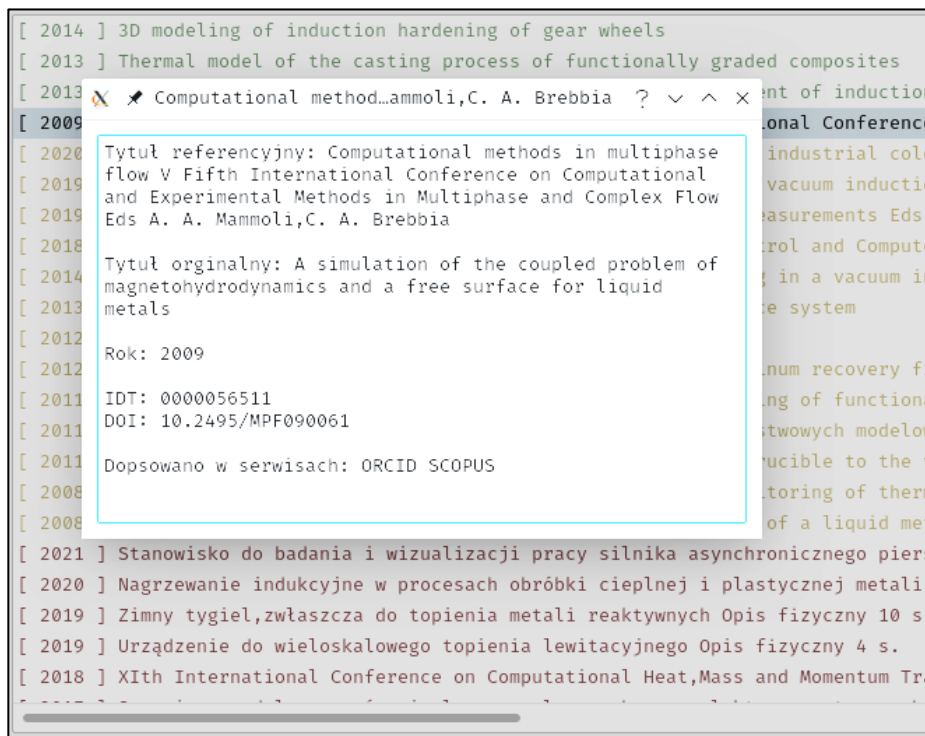
Pozycje na liście, oznaczonej jako „Współpracownicy” są posortowane według ilości wspólnych publikacji z wyszukiwanym podmiotem, znajdującym się zawsze na górze listy. Wybieranie dowolnej pozycji spowoduje wyświetlenie, na prawym panelu (opisanym w kolejnym podrozdziale) listy wspólnych publikacji. Tego typu rozwiązanie umożliwia dodatkową weryfikację poprawności danych ze strony użytkownika.

Podwójne kliknięcie w pozycję na tej liście sprawi, że identyfikator wybranej osoby, zostanie przepisany do panelu szukania. Ważnym jest, aby zaznaczyć, że proces wyszukiwania nie nastąpi, do momentu ponownego wyboru przycisku szukaj. Natychmiastowe rozpoczęcie wyszukiwania, w przypadku przypadkowego podwójnego kliknięcia, mogłoby wywołać konsternację u obsługującego, co nie jest pożądanym odczuciem przy korzystaniu z aplikacji.

#### **3.4.1.3. Lista publikacji**

Największym panelem aplikacji (Rys. 12) jest pole zawierające wynik działania aplikacji. Są to kolejne publikacje posortowane za pomocą trzech zaznaczonych kluczy. Najważniejszy czynnik wpływający na kolejność to ilość serwisów, w jakich udało się dopasować daną pozycję. Informację tę uwidacznia kolor pozycji. Zielony wskazuje, że publikacja została odnaleziona we wszystkich wspieranych bazach. Pomarańczowy informuje o częściowym dopasowaniu, natomiast czerwony ma za zadanie wyróżnić pozycje bez żadnego dopasowania. Kolejne dwa czynniki wpływające na kolejność to rok zapisany w kwadratowych nawiasach oraz tytuł publikacji. Przedstawione informacje nie są jednak kompletne, ponieważ wciąż brakuje informacji, w których serwisach prace nie zostały odnalezione. Jeżeli

użytkownik potrzebuje takiej informacji, ma dwie możliwości. Pierwszą z nich jest podwójne kliknięcie, które wyświetli okienko ze szczegółami (Rys. 11), lub może wygenerować raport.



Rys. 11. Okienko dialogowe, ze szczegółami przeszukiwania (rozjaśnione)

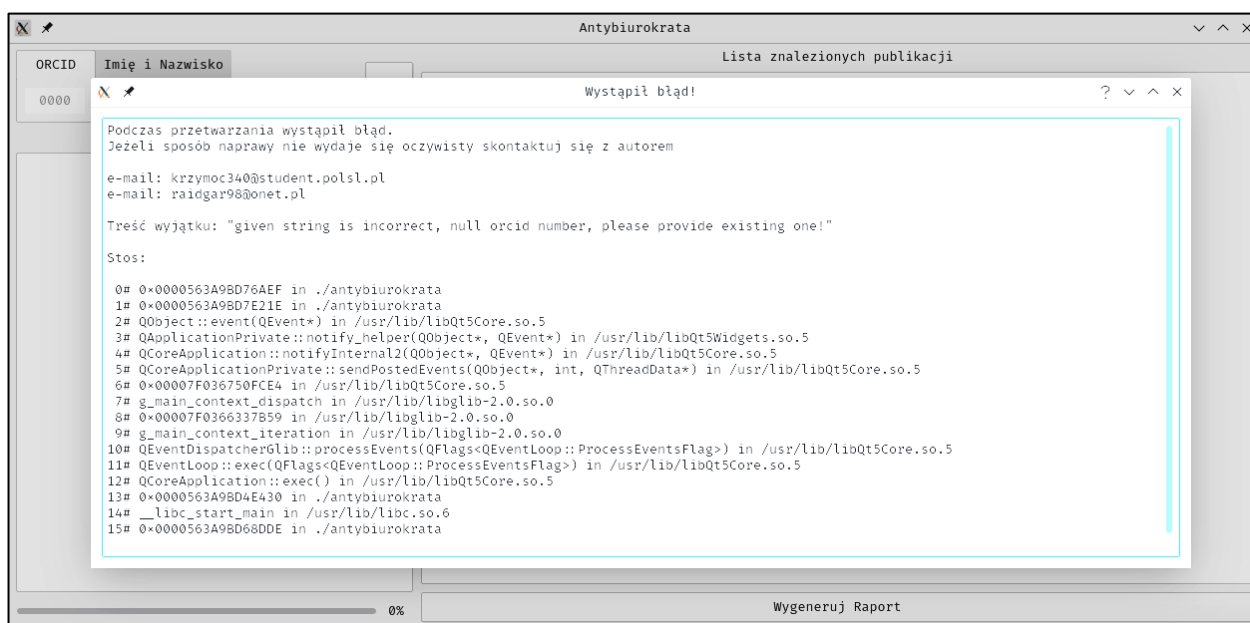


Rys. 12. Panel publikacji z przykładowymi danymi

### 3.4.1.4. Generowanie raportu

Ostatni element interfejsu użytkownika, znajduje się pod listą publikacji oraz składa się wyłącznie z przycisku o długości równej szerokości elementu powyżej. Jedynym jego celem jest wyzwolenie akcji generowania raportu. Podobnie jak przycisk wyszukiwania, również posiada przypisany skrót klawiszowy: **CTRL + ENTER**

### 3.4.1.5. Komunikat błędu



Rys. 13. Okno dialogowe w momencie detekcji błędu (rozjaśnione)

Komunikat o błędzie (Rys. 13) zawiera stałą, krótką notkę na temat postępowania, która wskazuje użytkownikowi możliwe drogi szukania rozwiązania problemu. Pierwszą sugerowaną jest zapoznanie się z treścią komunikatu, w języku angielskim. Jeżeli użytkownik zna język angielski lub jest zaznajomiony już z podobnym błędem, solucja okaże się prosta. W tym wypadku nie został podany identyfikator ORCID.

Alternatywnym rozwiązaniem problemu jest kontakt z autorem projektu. Kontakt jest możliwy mailowo, za pomocą jednego z dwóch wymienionych adresów e-mail. Celem usprawnienia procesu identyfikacji błędu, dołączany jest stos wywołań, który jest bardzo pomocny podczas analizy błędu.

Całość jest wypisana w obszarze tekstowym tylko do odczytu. Wybór takiego obiektu do reprezentacji tekstowej, sprawia, że nie ma problemu ze skopiowaniem i wklejeniem do wiadomości mailowej całości tekstu.

Po zamknięciu okna dialogowego, cały interfejs jest przywracany do ustawień umożliwiających ponowne wyszukiwanie.

### **3.5.     *Pobór danych***

#### **3.5.1. Baza *DOROBK***

Problematyka poboru danych z bazy DOROBK, została po części wyjaśniona w rozdziale poświęconym przeglądowi literatury w podrozdziale omawiającym aktualny model synchronizacji (3.1.2). W tym rozdziale nacisk zostanie położony na stworzone rozwiązania, umożliwiające wydobywanie interesujących danych.

##### **3.5.1.1.   Przygotowanie zapytania**

Pierwszym krokiem umożliwiającym stworzenie zapytania jest zapoznanie się z dostępną dokumentacją bądź jak w tym wypadku analiza ruchu, w kontekście wprowadzonych w formularzu danych (Rys. 1). Konstrukcja aplikacji daje dwa możliwe kryteria wyszukiwania. Pierwszym jest identyfikator *ORCID*, drugim zaś, para ciągów znakowych imię i nazwisko. Przeglądając dostępne pola, umożliwiające filtrowanie na stronie bazy DOROBK, okazuje się, że dostępne są obie możliwości, jednakże wyniki zwracane przez stronę w przypadku filtrowania po identyfikatorze *ORCID* są skromniejsze niż przy wyszukiwaniu tej samej osoby, za pomocą imienia i nazwiska. Potwierdza to próba wyszukiwania za pomocą klucza: *ADRIAN SMAGÓR*, zwracająca 33 wyniki oraz próba wyszukania za pomocą odpowiadającego identyfikatora *ORCID: 0000-0002-1994-3266*. Druga próba zamiast spodziewanych 33 wyników, zwraca zaledwie 27. Kierując się większą ilością danych, para imię i nazwisko zostaje wybrana, jako klucz w przeszukiwaniu bazy DOROBK.

Analiza ruchu sieciowego strony, wskazuje, że polem w żądaniu, pobierającym rekordy, odpowiedzialnym za przechowywanie imienia i nazwiska jest to, nazwane *V\_00*. Dodatkowo zostaje uwidoczniła różnica, pomiędzy wprowadzonym ciągiem znaków: *ADRIAN SMAGÓR*, a odczytanym z żądania: *SMAG%C3%93R+ADRIAN*.

Wynika to z faktu zastępowania znaków, nieznajdujących się w zakresie tablicy *ASCII* [32], odpowiednimi znacznikami, aby zawartość ciała żądania, została poprawnie zinterpretowana, przez wszystkie pośredniczące programy, na serwerze kończąc. W tym przypadku jest to zamiana litery „Ó” na „%C3%93”, co jest liczbą kodową litery, w systemie *UTF-8* zapisaną systemem szesnastkowym [33]. Dodatkowo podmiana znaków specjalnych zabezpiecza system, przed wstrzykiwaniem fragmentów kodu języka SQL (ang. *SQL Injection*) [34], oraz wstrzykiwaniem skryptów klienckich (ang. *Cross Site Scripting*) [35], które mogą narazić serwer, lub klientów na wykonanie niepożądanego kodu.

Przed przejściem do kolejnego etapu, należy dokonać sprawdzenia, jak serwer będzie reagował, gdy zapytanie nie nadchodzi za pośrednictwem aplikacji klienckiej, jaką jest aplikacja webowa bazy *DOROBK*. Jednym z możliwych scenariuszy, jest odmowa dostępu, na przykład, przez brak ważnego uwierzytelnienia za pomocą protokołu *OAuth*. Wysłanie żądania za pomocą narzędzia *cURL*, odrzuca tą wątpliwość, poprzez wyświetlenie odpowiedzi z serwera, zawierającą poprawną odpowiedź.

Public Member Functions	
template<typename... U>	<b>demangler</b> (U &&... u)
template<conv_t type>	<b>demangler &amp; process</b> () performs processing, and guards to not do it twice More...
<b>demangler &amp;</b>	<b>operator</b> () () proxy to process<> More...
string_view_type	<b>get</b> () const gets view on processed data More...
string_type	<b>get_copy</b> () const gets copy of processed data More...
template<conv_t type>	<b>requires</b> (type==conv_t::HTML  type==conv_t::URL) static void <b>mangle</b> (u16str &out) do oposite job to demangle, works only for HTML More...
Static Public Member Functions	
template<conv_t type>	static void <b>mangle</b> (str &out) proxy to mangle(u16str&) More...
	static void <b>mangle_html</b> (u16str &out) implementation of mangle for HTML conversion tag More...
	static void <b>mangle_url</b> (u16str &out) implementation of mangle for URL conversion type More...
	static bool <b>is_polish</b> (u16str_v view) trivially way of gussing is string is in polish lang? It demangles to english, and returns comprasion between
	static void <b>sanitize</b> (u16str &out) unifies given string to comparable format accross whole project More...
template<conv_t type>	static void <b>demangle</b> (u16str &out) do all job, by replacing polish chars to selected one More...
template<conv_t type>	static void <b>demangle</b> (str &out) proxy to <b>demangle</b> (u16str&) More...

Rys. 14. Interfejs klasy *demangler*

Generowanie zapytania po stronie aplikacji wymaga jednak stworzenia mechanizmu, który będzie odpowiednio preparował wprowadzone imiona i nazwiska, aby spełniały normy bezpieczeństwa. Zadanie zamiany łańcuchów na różne formaty, należy do klasy *demangler* (Rys. 14). Zawiera ona szereg metod, umożliwiających unifikację, konwersję i rewersję wejściowych łańcuchów znaków. Klasa ta jest wspierana, przez klasę *depolonizator*, która zawiera jako pole statyczne, definicję mapy translacji, pomiędzy różnymi zapisami.

### 3.5.1.2. Przetwarzanie danych

Przetwarzanie źródła strony internetowej, można wykonać na jeden z dwóch sposobów. Pierwszy, bardziej elegancki, to zastosowanie odpowiedniego parsera, umożliwiającego semantyczne przetworzenie tekstu, co pozwala do pewnego stopnia uniezależnić się od zmian wprowadzanych na stronie. Zaletą, jest również możliwość tworzenia złożonej logiki oraz delegację walidacji danych do zewnętrznego kodu parsującego.

Alternatywą jest wyszukiwanie, przez dopasowywanie wzorów, na przykład poprzez wykorzystanie wyrażeń regularnych lub szukając charakterystycznych fragmentów w tekście. Zdecydowaną zaletą tego rozwiązania, jest brak wymogu, ze strony źródła, aby było poprawnie zapisane, co mogłoby doprowadzić do błędu w analizatorze formatu. Dodatkowo, brak konieczności analizy oraz tworzenia rozległych struktur drzewiastych w pamięci, czyni taki proces szybszym. Wadą tego podejścia jest duża wrażliwość mechanizmu przetwarzającego na zmiany w źródle strony.

Podczas procesu tworzenia kodu, przetestowanych zostało kilka bibliotek napisanych w języku C++, jednakże żadna nie spełniła jednocześnie dwóch wymagań: wysokiej wydajności oraz zrozumiałego interfejsu aplikacji. Dzięki pierwszej, wykonanie kodu, będzie mieścić się w rozsądnych ramach czasowych, szczególnie przy większej liczbie publikacji. Druga zapewnia łatwość w utrzymaniu kodu oraz nie utrudnia procesu diagnozowania błędów. Leksykalne przetwarzanie kodu *HTML*, *XML* lub innego opartego na znacznikach, nie jest wydajne na tle innych formatów. Dobrze oddaje to cytāt wybitnego programisty, twórcy jądra systemu *GNU/Linux*, Linusa Torvaldsa [36], jednakże z uwagi na jego wulgarność, nie zostanie tutaj przytoczona.

Konsekwencją nieudanych prób było stworzenie dedykowanego interpretera kodu strony bazy *DORORBEK*. Opiera się on na słowach kluczowych oraz składa się z trzech warstw. Każda z nich działa w oparciu o wynik poprzedniej. Pierwsza jest najbardziej prymitywna, jednakże redukuje znacząco ilość kodu do przetworzenia. Korzystając z wyrażenia regularnego wyszukuje linijki, zawierające potrzebne dane.

Jeżeli aktualnie przetwarzana linijka, pasuje do używanego wyrażenia regularnego, jest ona przekazywana do funkcji *html\_scalpel*. Jej zadaniem jest odfiltrowanie znaczników, wraz z ich zawartością, pozostawiając wyłącznie tekst pomiędzy nimi. Na tym etapie największa część przetwarzanego kodu jest odfiltrowywana. Znalezione słowa są kumulowane w wektorze, który celem uniknięcia kopiowania, jest przekazywany do funkcji wyłuskującej, jako referencja.

Na samym początku przychodząca linijka jest konwertowana do typu *u16str*, który zapewnia poprawne kodowanie polskich znaków diakrytycznych. Typ ten jest definiowany jako zmienna łańcuchowa z biblioteki standardowej, która zamiast podstawowych jednobajtowych zmiennych znakowych używa dwubajtowych. Jest to najbardziej kosztowna operacja w ciele funkcji, jednakże gwarantuje poprawne przetworzenie odczytywanych danych. Następnie w pętli, następuje iterowanie po zmiennej tekstowej. Za pomocą serii przełączników, następuje decyzja, czy dany znak powinien zostać uwzględniony. Wykrywając określone znaki, między innymi

```
33
34     bgpolsl_repr_t::bgpolsl_repr_t(const std::vector<u16str>& words)
35     {
36         const std::map<u16str, u16str*> keywords{{std::pair<u16str, u16str*>
37             {u"IDT", &idt}, {u"Rok", &year}, {u"Autorzy", &authors}, {u"Tytuł oryginału", &org_title},
38             {u"Tytuł całości", &whole_title}, {u"Czasopismo", nullptr}, {u"Szczegóły", nullptr},
39             {u"p-ISSN", &p_issn}, {u"DOI", &doi}, {u"Impact Factor", nullptr}, {u"e-ISSN", &e_issn},
40             {u"Adres", nullptr}, {u"Afiliacja", &affiliation}, {u"Punktacja", nullptr},
41             {u"Pobierz", nullptr}, {u"Dyscypliny", nullptr}, {u"Uwaga", nullptr}};
42
43         u16str* savepoint = nullptr;
44         for(const u16str& word: words)
45         {
46             u16str save_range;
47             for(const auto& kv: keywords)
48             {
49                 const size_t pos = word.find(kv.first);
50                 if(pos != u16str::npos) /* if found */
51                 {
52                     savepoint = kv.second;
53                     save_range = u16str_v{word.c_str() + pos + 1ul + kv.first.size()};
54                     break;
55                 }
56                 else save_range = word;
57             }
58
59             if(savepoint)
60             {
61                 if(savepoint->size() > 0) *savepoint += u' ';
62                 core::demangler<u16str, u16str_v>::mangle<conv_t::HTML>(save_range);
63                 *savepoint += save_range;
64             }
65         }
66     }
```

Rys. 15. Konstruktor struktury *bgpolsl\_repr\_t*

spacje i przecinki, ciąg znaków jest dzielony na słowa, które są przesuwane do wektora, korzystając z funkcji bibliotecznej *std::move*.

Przygotowany w ten sposób zestaw słów, jest przekazywany do trzeciej warstwy, reprezentowanej przez konstruktor struktury *bgpolsl\_repr\_t* (Rys. 15). Zawiera ona zbiór instrukcji w postaci tablicy asocjacyjnej. Jej kluczem jest oczekiwane słowo, natomiast wartość jest reprezentowana przez wskaźnik na zmienną, do której mają trafiać kolejne słowa. Taka organizacja ma dwie zalety oraz jedną poważną wadę.

Pierwsza zaleta dotyczy sporej elastyczności kodu. Jeżeli znajdzie potrzeba wyłuskania dodatkowych danych, zmiana będzie polegała na modyfikacji istniejących wartości w mapie lub dodaniu nowych. Definiowanie ignorowanych pól poprzez przypisanie ich punktowi zapisu wartość pustą, jest czytelne i zrozumiałe.

Kolejną zaletą jest umiejscowienie obciążających tekstowych porównań w strukturze, zoptymalizowanej pod kątem przeszukiwań. Dzięki złożoności logarytmicznej, przeszukiwany zbiór z każdym porównaniem się zmniejsza. W przeciwieństwie do wektora, nie ma konieczności przeszukiwania przy każdym słowie całej struktury.

Wadą tego rozwiązania, jest zwiększające się prawdopodobieństwo wystąpienia błędnego przetwarzania z każdym kolejnym kluczem. Jeżeli w wartości dowolnego pola, na przykład w tytule, znajdzie się wyraz obecny już jako klucz, może dojść do błędnego wyłuskania tytułu, w tym wypadku ucięcia jego części, oraz potencjalnego wpisania jego reszty, do błędnie rozpoznanego pola. Jest to poważne zagrożenie, jednakże stosunkowo proste w diagnozie z uwagi na charakterystyczne przesunięcie wszystkich danych.

Tak przygotowany obiekt, dodawany jest do kolekcji, który jako zbiór wstępnie zorganizowanych danych, jest przekazywany do warstw o wyższej abstrakcji, celem umożliwienia porównania danych z różnych źródeł. Wyższe warstwy zostaną omówione w podrozdziale 3.6 *Unifikacja*.



### 3.5.2. Bazy *ORCID* oraz *SCOPUS*

Tytułowe bazy posiadają mocno zbliżony schemat danych przychodzących, co pozwala omówić je jednocześnie nie pomijając kluczowych kwestii. Oba źródła danych są na tyle zbliżone, że korzystają z wielu tych samych klas oraz funkcji.

#### 3.5.2.1. Obsługa zapytań

Portal *ORCID*, zapewnia wysoki poziom dokumentacji oraz przykładów użycia. Skutkiem tego jest bardzo krótki kod zajmujący się generowaniem żądania, zajmujący zaledwie trzy linijki. Serwis podczas zapytania zwraca komplet danych, powodując wyczerpanie tematu obsługi sieci w ramach bazy *ORCID*.

Klasa odpowiedzialna, za obsługę połączenia z serwisem *ORCID*, posiada względem pozostałych adapterów jedną więcej metodę do poboru informacji o zadanej za pomocą argumentów osobie. Metoda *get\_name\_and\_surname* i jest używana w momencie, gdy użytkownik chce skorzystać z wyszukiwania za pomocą identyfikatora *ORCID*. Mimo, że wyszukanie za pomocą numeru wymaga dodatkowego zapytania, jest szybsze z powodu możliwości uruchomienia wszystkich wątków jednocześnie, ponieważ z wyjątkiem adaptera obsługującego bazę *DOROBK*, wszystkie adaptery korzystają z identyfikatora *ORCID*.

Przykład serwisu *SCOPUS*, jest bardziej złożony. Generowanie żądania utrudniają limity narzucone przez portal, który zwraca maksymalnie dwadzieścia pięć rekordów, co w przypadku bardziej płodnych twórczo pracowników Politechniki Śląskiej wymaga wielokrotnego odpytywania się bazy. Mechanizm stronicowania (ang. *pagination*) jest zrobiony na zasadzie podpowiedzi doklejanych do danych zwrotnych. Oznacza to, brak konieczności wyliczania zakresu kolejnej części danych, ponieważ wyliczone wartości, są obecne w ciele odpowiedzi z serwisu.

Ograniczenia czasowe nie pozwoliły na wdrożenie dedykowanego rozwiązania, które znajduje się w języku C++ od najnowszego, obowiązującego w tym projekcie, standardu – korutyn. Pozwoliłoby to na łatwiejsze uwspółbieżnienie procesu oczekiwania na dane oraz przetwarzanie już pobranych.

### 3.5.2.2. Przetwarzanie danych

Pod względem przetwarzania, oba źródła danych są identyczne i można opisać je za pomocą algorytmu:

1. Korzystając z funkcji bibliotecznej, przetwórz przychodzące dane w formacie *JSON*;
2. Korzystając z opublikowanego na stronie dokumentacji serwisu, schematu pliku *JSON*, stwórz obiekt tymczasowy i wyłuskuj kolejne pola;
  - a. Jeżeli pole jest zagnieżdżone, przed każdym wyłuskaniem sprawdź, czy pole nie jest puste (nie ma wartości *null*);
  - b. Jeżeli spodziewanym typem elementu jest tablica, sprawdź czy jej rozmiar jest większy lub równy jedności;
  - c. W momencie otrzymania typu prostego na spodziewanym poziomie, dokonaj przypisania do odpowiedniego pola w tymczasowym obiekcie
  - d. Po uzupełnieniu wszystkich wymaganych pól w obiekcie, dodaj go do kolekcji

Konsekwencją schematu pliku *JSON*, co za tym idzie również konstrukcji algorytmu, jest tworzenie przez kod, charakterystycznych wielokrotnie zagnieżdżonych instrukcji sterujących (Rys. 16)

```
114
115     const jvalue& addresses = json->get("addresses", null_value);
116     if(addresses != null_value)
117     {
118         const jvalue& address_arr = addresses.get("address", empty_array);
119         if(address_arr.isArray())
120         {
121             for(const jvalue& item: address_arr)
122             {
123                 const jvalue& source = item.get("source", null_value);
124                 if(source != null_value)
125                 {
126                     const jvalue& source_name = source.get("source-name", null_value);
127                     if(source_name != null_value)
128                     {
129                         const jvalue& value = source_name.get("value", null_value);
130                         if(value != null_value && value.isString())
131                         {
132                             const str x{value.asCString()};
133                             if(try_split(x)) return;
134                         }
135                     }
136                 }
137             }
138         }
139     }
```

Rys. 16. Przykładowy fragment wyłuskania wartości z danych w formacie *JSON*

### 3.6. Unifikacja

Wstępne przetworzenie nadchodzących danych, omówione w poprzednim rozdziale, jest dopiero początkiem w obróbce danych, a zarazem pierwszym poziomem w przepływie (ang. *flow*) aplikacji. Kolejnym etapem, koniecznym, aby móc w prosty sposób porównywać dane.

Podstawowym rozwiązaniem, zastosowanym również tutaj, jest stworzenie klasy lub struktury, która będzie obiektową reprezentacją autora oraz publikacji. Sposób ten został rozwinięty o możliwość dowolnej serializacji oraz deserializacji takiej struktury. Zabieg ma na celu uprościć proces logowania, a także umożliwić przyszły rozwój takich funkcjonalności jak obiektowe podejście do bazy danych, szerokie wsparcie różnych form raportów, czy możliwość strumieniowania obiektów. Problematyka oraz złożoność implementacji została omówiona w rozdziale 3.9.

```
368
369      /** @brief object representation of publication */
370      You, seconds ago | 1 author (You)
371      struct detail_publication_t : public serial_helper_t
372      {
373          u16ser<detail_publication_t::_>          title;
374          u16ser<detail_publication_t::title>      polish_title;
375          dser<detail_publication_t::polish_title, uint16_t> year;
376          dser<detail_publication_t::year, ids_storage_t> ids;
377
378      /**
379       * @brief compares two me with other
380       *
381       * @return int 0 = equal, 1 = greater, -1 = lesser
382       */
383       int compare(const detail_publication_t&) const;
384       inline friend bool operator==(const detail_publication_t& me, const detail_publication_t& other) ...
385       inline friend bool operator!=(const detail_publication_t& me, const detail_publication_t& other) ...
386       inline friend bool operator<(const detail_publication_t& me, const detail_publication_t& other) ...
387       inline friend bool operator>(const detail_publication_t& me, const detail_publication_t& other) ...
388
389     };
390     using publication_t = cser<detail_publication_t::ids>;
391     using shared_publication_t = pd::shared_t<publication_t>;
```

Rys. 17. Obiektowa reprezentacja publikacji

```
422
423      /** @brief object representation of person (author) */
424      You, seconds ago | 1 author (You)
425      struct detail_person_t : public serial_helper_t
426      {
427          dser<detail_person_t::_>          name;
428          dser<detail_person_t::name, polish_name_t> surname;
429          dser<detail_person_t::surname, orcid_t> orcid;
430          mutable dser<detail_person_t::orcid, publications_storage_t> publications{};
431
432       friend inline bool operator==(const detail_person_t& p1, const detail_person_t& p2) ...
433       friend inline bool operator!=(const detail_person_t& p1, const detail_person_t& p2) ...
434       friend inline bool operator<(const detail_person_t& p1, const detail_person_t& p2) ...
435
436     };
437     using person_t = cser<detail_person_t::publications>;
438     using shared_person_t = processing_details::shared_t<person_t>;
```

Rys. 18. Obiektowa reprezentacja osoby autora

### 3.6.1. Reprezentacja obiektowa

Podczas omawiania drugiego poziomu, najważniejsze są dwie klasy, obie zaprezentowane na powyższych zrzutach ekranu (Rys. 17) (Rys. 18). Jednakże faktyczny model reprezentacji obiektowej składa się z wielu mniejszych klas, których zadanie jest wysoce wyspecjalizowane.

```
158
159      /**
160      * @brief wraps string
161      *
162      * @tparam validator with this struct incoming strings will be validated
163      * @tparam unifier with this struct incoming strings will be unified
164      */
165      You, seconds ago | 1 author (You)
166      template<typename validator = default_validator, typename unifier = default_unifier>
167      struct detail_string_holder_t : public serial_helper_t
168      {
169          ser<6detail_string_holder_t::_, u16str> data;
170          u16str raw;
171
172          using custom_serialize = pd::string::serial;
173          using custom_deserialize = pd::string::deserial;
174          using custom_pretty_print = pd::string::pretty_print;
```

Rys. 19. Fragment klasy - obiektowej postaci zweryfikowanego i jednolitego tekstu

Przykładem takiej klasy jest *detail\_string\_holder\_t* (Rys. 19), która odpowiada za weryfikowanie, oraz unifikowanie przechowywanych łańcuchów znakowych. Wymagane parametry szablonu to dwie klasy.

Pierwsza z nich, o nazwie *validator*, wymaga od typu dwóch metod. Pierwsza to konstruktor jednoargumentowy, przyjmujący stałą referencję do łańcucha znaków, który ma zostać poddany weryfikacji. Druga wymagana metoda to operator konwersji na typ logiczny – *bool*. Użycie tego typu sprowadza się do stworzenia tymczasowego obiektu, na którym natychmiast jest wymuszona konwersja do zmiennej logicznej. Otrzymany wynik decyduje o wystąpieniu wyjątku podczas przetwarzania łańcucha znaków, który docelowo ma zostać przechowywany.

Kolejny parametr to *unifier*, czy unifikator. Wymogi dla tego typu to wyłącznie jednoargumentowy konstruktor, przyjmujący referencje na zweryfikowany łańcuch znaków. Klasa ma za zadanie przetworzyć łańcuch w sposób umożliwiający szybsze i pewniejsze porównanie.

Dodatkowo struktura przechowuje przetworzoną wartość w oryginale w polu o nazwie *raw*. Pozorna redundancja danych jest w rzeczywistości optymalizacją czasu przetwarzania, dokonana kosztem pamięci operacyjnej. Z uwagi na nierzadkie porównania dwóch ciągów znaków, w strukturach uporządkowanych, lub w trakcie dokonywania dopasowania różnych obiektów, lepiej jest zapisać przetworzoną formę, a w przypadku konieczności odczytu, używać wyliczonej postaci. Konieczność przechowania oryginału wiąże się z wymogiem reprezentacji przechowywanych danych użytkownikowi w interfejsie użytkownika.

Przykładem specjalizacji, tej struktury, jest typ *polish\_name\_t* (Rys. 20), który w ramach parametrów szablonu, przekazuje własne narzędzia walidacji i ujednolicenia łańcuchów znakowych

```
294
295      /** @brief provides validation for polish names */
      You, a month ago | 1 author (You)
296      struct polish_validator
297      {
298          const u16str_v& x;
299
300          operator bool() const noexcept;
301      };
302
303      /** @brief provides unification for polish names (capitalizing) */
      You, a month ago | 1 author (You)
304      struct polish_unifier
305      {
306          polish_unifier(u16str& x) noexcept;
307      };
308
309      /** @brief aliasing for handy usage */
310      using polish_name_t = string_holder_custom_t<polish_validator, polish_unifier>;
311
```

Rys. 20. Deklaracja typu *polish\_name\_t*

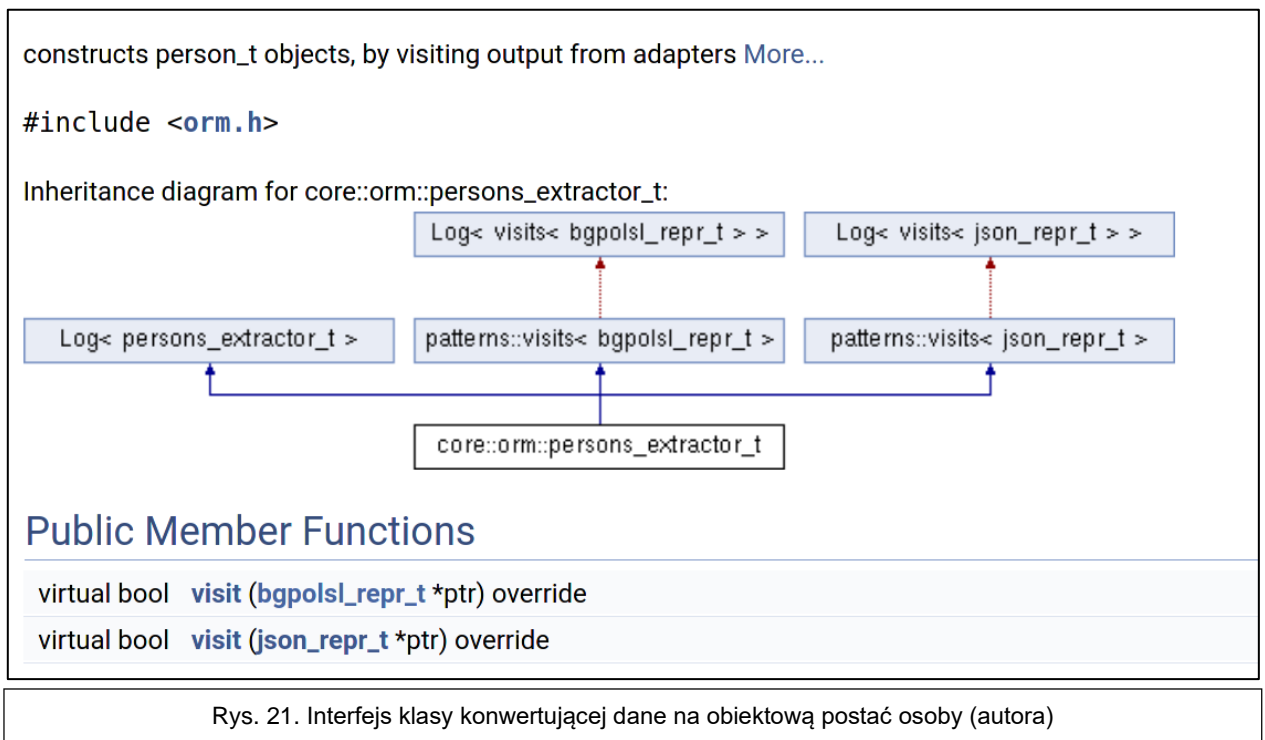
Rozwiązanie oparte o szablony, pozwala na delegację ról do pojedynczych klas lub struktur, oraz znacząco redukuje ilość pisanego kodu. Jeżeli będzie potrzeba zmiany sposobu unifikacji polskich imion lub dostosowanie poziomu weryfikacji, wystarczy tego dokonać w jednym miejscu. W tym przypadku rolę kontenera pełni *string\_holder\_t*, rolę weryfikatora *polish\_validator*, a unifikatora *polish\_unifier*.

Klasa od przechowywania łańcuchów znaków jest w projekcie wykorzystana również do znakowej reprezentacji typów enumerowanych, przechowywania tytułów czy reprezentacji identyfikatorów publikacji, takich jak *DOI*.

### 3.6.2. Generowanie obiektów wyższej abstrakcji

Przejdźmy między poziomem pierwszym, gdzie znajdują się świeżo przetworzone dane z serwisów, do poziomu umożliwiającego ich porównanie, zostało wykonane przy pomocy autorskiej modyfikacji wzorca projektowego wizytator [37] do wzorca nazwanego roboczo: zaproszony wizytator.

Różnica polega na częściowo statycznym rozwiązaniu zależności, a także jawnej deklaracji wspieranych klas, przez wizytatora. Zapewnia to przejrzystość w czytaniu kodu, daje możliwość kompilatorowi na zwiększenie poziomu optymalizacji, a także zaznaczenie dodatkowej zależności na diagramie klas w dokumentacji automatycznej (Rys. 21).



Wykorzystanie wzorca projektowego wizytator jest spowodowane, chęcią wydzielenia funkcjonalności poza klasę, w tym wypadku *person\_t*. Powoduje to, delegację funkcjonalności i odciążenie klasy z części zadań. Dodatkowo umożliwia stworzenie logicznego podziału i przejrzystości w zależnościach pomiędzy bibliotekami. Klasa odpowiedzialna za obiektową reprezentację osoby, nie ma potrzeby dostępu do szczegółów pobierania danych. Rozwój aplikacji, poprzez dodawanie kolejnych interfejsów sieciowych powodowałby znaczący rozrost klasy, której zadanie nie polega na konwersji pomiędzy warstwami.

### 3.6.2.1. Przetwarzanie danych z bazy DOROBK

Rozpoczęcie procesu przetwarzania odbywa się poprzez odwiedzenie, przez klasę *publications\_extractor\_t*, struktury wyjściowej z adaptera bazy *DOROBK* – *bgpolsl\_repr\_t*. Przed uzupełnieniem dowolnego pola, zapewniana jest podstawowa weryfikacja, sprawdzająca obecność jakichkolwiek danych oraz zakończenie przetwarzania, w przypadku braku kluczowych.

Pierwszym elementem uzupełnianym przez wizytator, są identyfikatory. W przypadku bazy *DOROBK*, obsługiwane są następujące wyróżniki: *IDT*, *DOI*, *E-ISSN* oraz *P-ISSN*. Każdy z nich jest unifikowany podczas dodawania do obiektu reprezentującej publikację. Jeżeli po wykryciu wszystkich możliwych zbior jest pusty, następuje wyjście z funkcji, ponieważ jest to podstawowy (ale nie jedyny!) czynnik determinujący dopasowanie publikacji względem innych źródeł danych.

Kolejnym przetwarzanym polem jest data, która po sprawdzeniu czy pole nie jest puste, wywołuje funkcję biblioteczną *std::stoi*. Brak wsparcia dla dwubajtowych ciągów znakowych, ze strony biblioteki standardowej było zaskoczeniem oraz wymusiło dodatkową konwersję do standardowego jednobajтового kontenera tekstu. Jest to dobre miejsce do rozpoczęcia procesu optymalizacji, zastępując koniunkcję tekstowej konwersji oraz wywołania funkcji biblitecznej, na dedykowany konwerter dwubajtowych łańcuchów znakowych do liczb całkowitych.

Pola przetwarzane, jako trzecie w kolejności, to tytuły. Ponieważ obiektowa reprezentacja przewiduje dwa rodzaje tytułów, jest konieczność dopasowania, który tytuł zostanie wpisany do odpowiedniego pola. Pierwszym polem obiekcie bardziej abstrakcyjnym, jest tytuł oryginalny, który ma priorytet podczas komparacji dwóch publikacji, oraz polski, gdy tytuł występuje w dwóch językach.

Optymistycznym przypadkiem rozróżnienia, jest zbiór publikacji, posiadających dwa osobne wpisy, na tytuł oryginalny i w języku polskim. Taki przypadek pozostawia tylko problem detekcji języka, ponieważ nie ma gwarancji, zastosowanego języka dla danego pola w obiekcie wyjściowym adapteru bazy *DOROBK*. Decyzyjność w tej sprawie jest niedopracowana i bazuje na detekcji polskich znaków w tekście, co jest mocnym nadużyciem, ale z uwagi na brak lepszego rozwiązania, częściowo spełnia

swoje zadanie. Założenie takich kryteriów sprawia, że zdania nie zawierające polskich znaków, nie zostaną poprawnie rozpoznane, na przykład: *Ala ma kota*.

Istnieje jednak niemały zbiór prac, gdzie dwa tytuły są wpisane do tego samego pola w obiekcie wynikowym z wstępnego przetwarzania. Niestety z uwagi na tę, niechlubną praktykę konkatencji tytułów w języku polskim i nowożytnym przy użyciu kropki, przecinka, średnika lub w najgorszym przypadku spacji, jako separatora, nie jest możliwe poprawne rozdzielanie dwóch tytułów, jeżeli znajdują się w tym samym polu. Lata zaniedbań w weryfikacji danych oraz trzymaniu spójności wpisów, mają swoją konsekwencję w braku możliwości reprezentacji tych publikacji w sposób prawidłowy za pomocą podejścia obiektowego. Warto zaznaczyć, że mimo niepoprawnych danych, wciąż jest możliwość wykazania zbieżności między odpowiadającymi sobie publikacjami z różnych źródeł. Presja czasowa, niestety uniemożliwiła implementację opisanego później, w niniejszej pracy, rozwiązania.

Przed zakończeniem przetwarzania publikacji następuje użycie jeszcze jednego, wcześniej wspomnianego (Rys. 21) wizytatora. Użycie polega na wpisaniu przetworzonej publikacji, do pola wizytatora oraz użyciu go na przychodzących danych.

```
9      bool persons_extractor_t::visit(bgpolsl_repr_t* ptr)
10     {
11         check_nullptr(ptr);
12         shared_person_t result;
13
14         const u16str_v affiliation(ptr->affiliation); // alias
15         auto& person = result().data(); // alias
16         const auto reset_person = [&person] { person.reset(new person_t{}); };
17         for(u16str_v part_of_affiliation: string_utils::split_words<u16str_v>(affiliation, u','))
18         {
19             if(part_of_affiliation.empty()) continue;
20             const string_utils::split_words<u16str_v> splitter{part_of_affiliation, u' '};
21             auto it = splitter.begin();
22             reset_person();
23
24             u16str_v v = (it == splitter.end() ? u"" : *it);
25 >         const auto safely_move = [&]() -> bool { ...
31
32             if(polish_name_t::value_t::validate(v)) (*person)().surname(v);
33 >         else ...
38
39             if(!safely_move()) continue;
40
41             if(polish_name_t::value_t::validate(v)) (*person)().name(v);
42 >         else ...
47
48             if(!safely_move()) continue;
49
50             if(orcids_t::value_t::is_valid_orcid_string(v))
51                 (*person)().orcid(orcids_t::value_t::from_string(v));
52 >         else ...
57
58         auto pair = this->persons->insert(result);
59         if(pair.second)
60             log.info() << "successfully added new author: "
61             << patterns::serial::pretty_print{(*pair.first)()} << logger::endl;
62         if(current_publication())
63             ((*pair.first)()).publications()->insert(current_publication().data());
64     }
65
66     return true;
67 }
68
```

Rys. 22. Metoda wyluskująca kolejnych autorów z zadanej publikacji



Klasa wyluskująca autorów, korzysta z autorskiego iteratora [38] zakresów tekstowych. Motywacją do implementacji takiej konstrukcji, było rozczarowanie nowym modulem biblioteki standardowej dla operacji na zakresach – *ranges*. Potrzeba podzielenia tekstu na słowa, innymi słowy, wskazania zakresów, pomiędzy spacjami, jest zaspokajana przez bibliotekę zakresów, jednakże oferowany dostęp do wytyczonych danych uniemożliwia wydajnościowe przetwarzanie. Dzieje się tak, ponieważ wbudowane iteratory pozwalają wyłącznie na dostęp do kolejnych liter, zamiast generować widoki, za pomocą bardzo szybkich widoków tekstowych (ang. *string view*). Powoduje to konieczność przepisania zakresu do tymczasowej zmiennej tekstowej, a następnie wykorzystanie tak stworzonego sztucznie widoku, co jest marnotrawstwem zasobów. Napisana klasa pozwala dowolnie dzielić tekst, zarówno na słowa, jak również na linie, czy dowolny inny znak, co zostało zaprezentowane w linii 17 i 20 na powyższym rzucie ekranu (Rys. 22). Klasa pozwala na dostęp do kolejnych zakresów w pętli zakresowej, co zdecydowanie poprawia czytelność kodu.

Dodanie autora do unikalnego zbioru, następuje, jeżeli zostaną poprawnie rozpoznane trzy łańcuchy znaków: imię, nazwisko oraz identyfikator *ORCID*. W każdym innym wypadku następuje przejście do przetwarzania kolejnego autora. Przy poprawnym rozpoznaniu autora, zostaje dodana jeszcze referencja do aktualnie przetwarzanej publikacji jako wspólnego wskaźnika, co omija duplikację danych oraz niweluje problem własności obiektu. Stworzenie tej zależności jest pomocą dla graficznego interfejsu użytkownika.

Zakończenie przetwarzanie w ekstraktorze publikacji, kończy dodanie obecnie przetwarzanej do zbioru już wcześniej, poprawnie przetworzonych prac.

#### **3.6.2.2. Przetwarzanie danych z pozostałych źródeł**

Obiektem wynikowym z przetwarzania danych pochodzących z baz *ORCID* oraz *SCOPUS*, są obiekty typu *json\_repr\_t*, dzięki czemu możliwe jest przetworzenie wyniku z obu źródeł za pomocą tej samej funkcji. Obniża to ilość pisanego, jak i generowanego kodu oraz ułatwia utrzymanie kodu w przyszłości.

Przetwarzanie rozpoczyna się od sprawdzenia roku, jeżeli jest niepusty następuje wcześniej wspomniana nieoptymalna konwersja do standardowego

łańcucha znaków, a następnie konwersja na liczbę. W kolejnych krokach zostają przepisane tytuły oraz numery identyfikacyjne. Jeżeli identyfikatory nie występują lub tytuł w przetwarzanym obiekcie jest pusty, metoda kończy przebieg. Aktywowanie ekstraktora autorów na tym obiekcie, jest podyktowane, wyłącznie weryfikacją danych, co zostanie wyjaśnione lepiej w kolejnych rozdziałach. Ostatecznie publikacja zostaje dodana do zbioru znalezionych publikacji.

### 3.7. Dopasowywanie

Kolejnym poziomem przetworzenia danych, jest stworzenie głębszych zależności pomiędzy przetworzonymi danymi. Zapis relacji, występujący pomiędzy publikacjami jest zdefiniowany przez trzy struktury: *detail\_publication\_with\_source\_t*, *detail\_sourced\_publication\_storage\_t*, oraz *detail\_publications\_summary\_t*.

```
/**
 * @brief object representation of match
 */
You, 6 days ago | 1 author (You)
struct detail_publication_with_source_t : public serial_helper_t
{
    dser<detail_publication_with_source_t::_, ser_match_type> source;
    dser<detail_publication_with_source_t::source, shared_publication_t> publication{};

    /** @brief redirect all comprasion operators */
    inline friend auto operator==(const detail_publication_with_source_t& pws1,
                                  const detail_publication_with_source_t& pws2)
    {
        return pws1.source().data() == pws2.source().data();
    }
};
using publication_with_source_t = cser<detail_publication_with_source_t::publication>;

/**
 * @brief container with unique values
 */
You, a week ago | 1 author (You)
struct detail_sourced_publication_storage_t : serial_helper_t
{
    using item_t = publication_with_source_t;
    using inner_t = std::set<item_t>;
    dser<detail_sourced_publication_storage_t::_, inner_t> data;

    using putter_t = pd::collection::emplacer<std::set, item_t>;
    using custom_serialize = pd::collection::serial<std::set, item_t>;
    using custom_deserialize = pd::collection::deserial<putter_t, std::set, item_t>;
    using custom_pretty_print = pd::collection::pretty_print<std::set, item_t>;
};
using sourced_publication_storage_t = cser<detail_sourced_publication_storage_t::data>;

/**
 * @brief object representation of comprasion result
 */
You, a week ago | 1 author (You)
struct detail_publications_summary_t : public serial_helper_t
{
    dser<detail_publications_summary_t::_, shared_publication_t> reference;
    dser<detail_publications_summary_t::reference, sourced_publication_storage_t> matched;
};
using publication_summary_t = cser<detail_publications_summary_t::matched>;
using shared_publication_summary_t = pd::shared_t<publication_summary_t>;
```

Rys. 23. Deklaracje struktur opisujące zależności pomiędzy publikacjami

Klasa, która zawiera definicję dopasowania publikacji, oraz generuje obiekty struktur przedstawionych na powyższym rysunku (Rys. 23) nazywa się *summary*. Metodą odpowiedzialną za przetwarzanie, która posiada wsparcie dla wielowątkowości, jest *process\_impl*.

Metoda jest zaprojektowana do wywołania w osobnym wątku oraz wymaga ustawienia zmiennych wewnętrznych, czego nie gwarantuje utworzenie klasy. Opóźnienie pełnej inicjalizacji, jest konieczne, aby umożliwić przekazanie do wielu wątków instancji tej samej klasy. Implikuje to również konieczność potwierdzenia, ustawienia wszystkich pól w klasie, za co odpowiedzialna jest zmienna logiczna o charakterze atomowym. Wykorzystane zostaje tutaj podejście oparte o rygiel pętlowy (ang. *spin lock*) [39], w kombinacji z oddaniem czasu procesora, co jest łatwiejsze w implementacji, oraz szybsze, niż tradycyjne podejście wykorzystujące muteks [40].

```
46
47 |         for(auto& item: *browser)
48 |         {
49 |             auto& report_item = (*item())();
50 |             auto& matches      = report_item.matched()().data();
51 |
52 |             if(matches.find(search) != matches.end())
53 |                 continue; // it's already matched, no sense for processing
54 |             const auto& ref = (*report_item.reference()().data())();
55 |
56 |             for(const objects::shared_publication_t& y: input)
57 |             {
58 |                 /**...
69 |                 if(ref.compare(*y()) == 0)
70 |                 {
71 |                     m_report.access([&](report_t& r) { matches.emplace(mt, y); });
72 |                     break;
73 |                 }
74 |             }
75 |         }
76 |     }
77
```

Rys. 24. Pętla tworząca zależności, pomiędzy publikacjami

Powyżej (Rys. 24), w linii 47 następuje rozpoczęcie pętli po zsynchronizowanej kolekcji, zawierającej instancje struktury *detail\_publications\_summary\_t*. Zaraz po sformułowaniu aliasów w dwóch pierwszych wersach, następuje sprawdzenie czy dopasowanie już wcześniej nie zostało dokonane. Jest to zabezpieczenie, na wypadek zmiany sposobu organizacji wątków lub sposobu przetwarzania, aktualnie można uznać ten fragment kodu za niepotrzebny.

Po zweryfikowaniu unikalności, następuje iteracja po publikacjach do przypisania, w przypadku wyznaczenia zgodności następuje utworzenie relacji, poprzez dodanie współdzielonego wskaźnika na publikację do kolekcji oraz przerwanie iteracji, z kwestii optymalizacyjnych. Linia 71 zawiera blokadę na iterowanej kolekcji *browser*, celem modyfikacji obecnego elementu.

Klasa generująca raport używa również wzorca projektowego obserwator, za pomocą którego informuje inne zainteresowane instance o zakończeniu przetwarzania. Emisja sygnału znajduje się w destruktorze klasy wraz, z którym zostaje rozpropagowany współdzielony wskaźnik do gotowego obiektu reprezentującego zależności pomiędzy publikacjami.

### 3.8. Nadzorowanie przebiegu

Konstrukcja poprzednich struktur wymusza ścisłą kontrolę życia obiektów, szczególnie podczas korzystania z klasy *summary*, która informację o zakończeniu zadania wysyła dopiero w destruktorze. Ponadto, sposób przetwarzania danych umożliwia zrównoleglenie części procesu.

```
252
253 // thread order
254 {
255     stop();
256     std::jthread th1{bgpolsl_getter};
257     stop();
258
259     if(!person())
260     {
261         std::unique_lock<std::mutex> lk{mtx_orcid};
262         cv_orcid.wait(lk, [&] {
263             stop();
264             return person();
265         });
266         dassert(person(), "person is not properly setted up!"_u8);
267     }
268
269     stop();
270     std::jthread th2{
271         core::detail::universal_getter
272         <objects::match_type::SCOPUS>
273         {person, sum, on_progress_delegate},
274
275         std::ref(mtx_report),
276         std::ref(cv_report)};
277     stop();
278
279     std::jthread th3{
280         core::detail::universal_getter
281         <objects::match_type::ORCID>
282         {person, sum, on_progress_delegate},
283
284         std::ref(mtx_report),
285         std::ref(cv_report)};
286     stop();
287 }
288 }
289
290
```

Rys. 25. Kolejność startu nowych wątków.

### 3.8.1. Zrównoleglenie przetwarzania danych

Podczas pobierania i dopasowywania, korzysta się w sumie z czterech wątków. Pierwszy, jest tworzony z uwagi na rozdzielenie obsługi graficznego interfejsu użytkownika, od czasochłonnego procesu ewaluacji danych. Kolejne trzy są tworzone w pierwszym. Każdy z nich posiada na własność jeden adapter oraz komplet wizytatorów, do ekstrakcji obiektów z przychodzących danych. Ma to dwie zalety. Pierwszą z nich jest klarowny podział własności oraz użytku obiektów. Dodatkowo dzięki ujednoliceniu interfejsów adapterów, możliwe jest stworzenie ogólnej formy przetwarzania danych dla dwóch z trzech wątków, które zostały zdefiniowane, jako funktory (Rys. 25).

```
20
21
22     /**
23     * @brief universal processor for alternative data sources
24     *
25     * @tparam mt data source
26     */
27     template<objects::match_type mt> struct universal_getter
28     {
29         using delegate_t = patterns::call_ownership_delegator<size_t>;
30         using w_summary_t = std::shared_ptr<reports::summary>;
31
32         delegate_t& on_progress;
33         w_summary_t sum;
34         std::shared_ptr<orm::persons_extractor_t> prsn_visitor;
35         std::shared_ptr<orm::publications_extractor_t> pub_visitor;
36
37         /**
38         * @brief Construct a new universal getter object
39         *
40         * @param source source of
41         * @param i_sum object to use as summary engine
42         * @param i_on_progress function to call on progress
43         */
44         universal_getter(const objects::shared_person_t& source, w_summary_t i_sum,
45                         delegate_t& i_on_progress) :
46             on_progress{i_on_progress},
47             sum{i_sum}, prsn_visitor{new orm::persons_extractor_t{}}
48         {}
49
50         void operator()(std::mutex& mtx, std::condition_variable_any& cv)...
```

Rys. 26. Deklaracja uniwersalnego funktora

Zgeneralizowanie problemu (Rys. 26), do szablonu struktury wykonawczej, umożliwia czytelne tworzenie nowych wątków, co zostało zaprezentowane w liniach 270 i 279 na wcześniejszym zrzucie ekranu (Rys. 25). Przy zachowaniu spójności w interfejsach kolejnych adapterów, podczas dodawania obsługi dla kolejnych serwisów, da to możliwość ponownego wykorzystania tego szablonu. Trzymając się zaleceń standardu, zostało dodane wsparcie tokena zatrzymania (ang. *stop token*),

co zapewnia dodatkową responsywność. Bierze się z ona z braku konieczności oczekiwania na zamknięcie programu w przypadku, gdy życzy sobie tego użytkownik. Daje to dodatkowo możliwość w przyszłości zaimplementowania anulowania aktualnego przetwarzania, co jest szczególnie przydatne, gdy użytkownik się pomylił i nie chce czekać na zakończenie zadania.

Zrównoleglenie przetwarzania wiąże się z powstaniem problemu zebrania wyników, z wszystkich wątków oraz przekazania ich do wyświetlenia na ekranie w graficznym interfejsie użytkownika. Rozwiązaniem tej kwestii okazał się wcześniej wspomniany wzorzec projektowy obserwator, który w ramach platformy programistycznej *Qt*, nazywany jest mechanizmem sygnałów i slotów. Zastosowanie go umożliwia informowanie wielu różnych instancji o zaistniałych wydarzeniach, bez konieczności tworzenia nielogicznych zależności. Klasa generująca sygnał, nie musi wiedzieć o istnieniu klasy, która zarządza wyświetlaniem tej informacji na ekranie. Użycie takiego schematu dodatkowo poprawia czytanie kodu.

```
14
15 MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new Ui::MainWindow)
16 {
17     ui->setupUi(this);
18
19     qRegisterMetaType<incoming_report_t>("incoming_report_t");
20     qRegisterMetaType<incoming_relatives_t>("incoming_relatives_t");
21     qRegisterMetaType<error_report_t>("error_report_t");
22
23     QObject::connect(this, &MainWindow::send_publications, this, &MainWindow::collect_publications);
24     QObject::connect(this, &MainWindow::send_related, this, &MainWindow::collect_related);
25     QObject::connect(this, &MainWindow::send_max_progress, this, &MainWindow::set_max_progress);
26     QObject::connect(this, &MainWindow::switch_activation, this, &MainWindow::set_activation);
27     QObject::connect(this, &MainWindow::send_error_report, this, &MainWindow::on_error_thrown);
28     QObject::connect(this, &MainWindow::send_report_generation_done, this, &MainWindow::on_report_generation_done);
29     QObject::connect(this, &MainWindow::send_progress, this, &MainWindow::set_progress, Qt::QueuedConnection);
30
31     eng.on_calculated_progress.register_slot([&](const size_t N) { emit set_max_progress(N); }); // You, a week ago • synchronized GUI with eng
32     eng.on_finish.register_slot([&](report_t ptr) { core::check_nullptr(ptr); emit send_publications(incoming_report_t{ptr}); }); // You, second
33     eng.on_collaboration_finish.register_slot([&](relatives_t ptr) { core::check_nullptr(ptr); emit send_related(incoming_relatives_t{ptr}); });
34     eng.on_progress.register_slot([&](const size_t N) { emit send_progress(N); }); // You, seconds ago • Uncommitted changes
35     eng.on_error.register_slot([&](error_report_t report) { emit send_error_report(report); emit switch_activation(true); this->clear_ui(); });
36
37 }
```

Rys. 27. Konstruktor klasy głównego okna

Wspomnianą czytelność widać na powyższym zrzucie ekranu (Rys. 27). Zaznaczone nazwy pól w liniach 31-35, są to obiekty klasy *observable*, która jest interfejsem na implementację sygnałów z biblioteki *boost*. Dokładnie widać elementy obsługiwane przez główne okno aplikacji, wśród tych oferowanych przez klasę zawierającą opisany wyżej model wielowątkowego przetwarzania – *engine*.

Wydzielenie osobnej klasy do zarządzania wątkami oraz przebiegiem synchronizacji, ma dać możliwość w przyszłości implementacji innych interfejsów użytkownika. Przykładowe implementacje, to interfejs sieciowy lub interfejs konsolowy.

### 3.8.2. Generowanie raportu

Wybrany formatem reprezentującym raport jest *xlsx*. Generowanie raportu jest obsługiwane przez klasę *generator*, która również interfejs wyjściowy bazuje na wzorcu projektowym obserwator. Dodatkowo cała klasa spełnia wymagania stawiane przez funktor, poprzez implementację operatora okrągłych nawiasów. Jest to spowodowane, chęcią ułatwienia uruchamiania procesu generowania pliku w osobnym wątku.

Przy instancjonowaniu klasy generującej plik opracowany przez firmę Microsoft, należy podać trzy parametry. Pierwszym jest struktura reprezentująca zależności między publikacjami, natomiast pozostałe dwa to wskaźniki na funkcje, które mają zostać wywołane podczas ogłaszania postępu, oraz po zakończeniu generowania pliku.

Metoda implementująca generację nazywa się *process\_impl*. Rozpoczyna się od ustawiania używanych stylów w arkuszu, co w wyniku zapewni jego stylistyczną spójność. Następnie zostaje stworzone indeksowanie, pomiędzy nagłówkami, a numerami kolumn, za pomocą dynamicznych tablic ciągłej alokacji – wektorze oraz tablic asocjacyjnych pochodzących z biblioteki standardowej.

Zapis do arkusza rozpoczyna się od nagłówków, które są przechowywane w wektorze. Na końcu tego oraz wszystkich kolejnych wierszy jest dokładana pojedyncza czarna komórka, co ma służyć, jako graficzne zakończenie wiersza. Spowodowane jest to wadą biblioteki, która posiada bardzo ograniczony zasób działających operacji na zakresach komórek, sprawiając, że kolorowane są całe wiersze, zamiast wyróżnionych zakresów.

W dalszej kolejności następuje iteracja po kolejnych relacjach pomiędzy publikacjami oraz ich zapis w generowanym arkuszu. Całość zwieńczona jest zapisem na dysk, ze sprawdzeniem powodzenia za pomocą asercji. W trakcie każdej pętli dochodzi niestety do sporej ilości konwersji, co jest spowodowane używaniem jako łańcuchów znaków klasy *QString*, przez bibliotekę obsługującą rozszerzenie *xlsx*.

W metodzie nadrzędnej *progress*, dostępnej publicznie, następuje wysłanie sygnałów o pełnym zakończeniu generacji pliku, za pomocą metody *progress\_impl*.

## 3.9. Omówienie serializacji

### 3.9.1. Zasada działania

System serializacji składa się z szeregu klas i struktur, uzupełniających się w zastosowaniu oraz funkcjonalności. Kluczowymi z punktu widzenia użytkownika klasy są dwie, *ser* oraz *cser*. Obie, pełnią funkcje prostego kontenera na dowolny typ oraz świadczą szereg operatorów, ułatwiających interakcje z posiadanym elementem.

Przekierowane są konstruktory, operatory przypisania, operatory porównania, a także zostały zaimplementowane operatory konwersji. Oprócz wymienionych występują dodatkowo operatory okrągłych nawiasów, które pełnią rolę funkcji dostępowych, ponieważ umożliwiają zarówno zapis jak i odczyt referencyjny wewnętrznego obiektu.

Kluczową, z punktu widzenia pełnionej funkcji, jest implementacja interfejsu akceptora, z wzorca projektowego wizytator. Metody *accept* w przypadku wspomnianych wcześniej dwóch klas różnią się znacząco swoją rolą, lecz nie aż tak bardzo implementacją. Obie klasy zawierają dwie implementacje metody *accept*, co jest spowodowane kwestiami optymalizacyjnymi, jako że jedna z metod jest oznaczona, jako stała, co umożliwia kompilatorowi lepsze ułożenie stosów wywołań.

Przed omówieniem metod akceptujących, należy omówić parametry szablonu, jako że to one stanowią kluczową część implementacji. Obie klasy są zadeklarowane za pomocą przeciążenia swojej pustej wersji, jest to spowodowane, przez wymogi stawiane przez język oraz chęcią czytelnego zapisu, podczas korzystania z struktur.

```
172
173     /**
174      * @brief ser implementation by specialization
175      *
176      * @tparam class_t type of owner class
177      * @tparam class_member_t type of previous member
178      * @tparam class_t::*value reference to member in class
179      * @tparam T type of current member
180     */
181
182     You, 2 weeks ago | 1 author (You)
181 |     template<typename class_t, typename class_member_t, class_member_t class_t::*value, typename T>
182 |     struct ser<value, T>
183 |     {
184 |         using is_serializable_class = std::true_type;
185 |         using value_type             = T;
186 |         /** wrapped value */
187 |         value_type val;
188 |     }
```

Rys. 28. Początek deklaracji struktury *ser*



Linia 181 na powyższym zrzucie ekranu (Rys. 28), zawiera cztery typy. Ostatni z nich to typ enkapsulowany przez omawianą klasę. Pozostałe dwa to niezbędne typy pozwalające na stworzenie trzeciego, ostatniego parametru szablonu - statycznego wskaźnika na kolejne pole klasy. Deklaracja klasy oparta na połączonych za pomocą statycznych wskaźników kolejnych pól w klasie, imituje kontener połączonej listy (ang. *linked list*).

Takie rozwiązanie umożliwia jednostronne, rekurencyjne, swobodne, statyczne iterowanie po zadeklarowanej w ten sposób klasie, za pomocą wizytatorów. Przykładem zastosowania takiego mechanizmu jest serializacja oraz deserializacja strumieniowa.

```

21 struct zwierzę_impl : public serial_helper_t
22 {
23     ser<zwierzę_impl::_, int>          id;
24     ser<zwierzę_impl::id, std::string> nazwa;
25     ser<zwierzę_impl::nazwa, int>      ilość_nóg;
26 };
27 using zwierzę = cser<zwierzę_impl::ilość_nóg>;
28
29 int main(int argc, char* argv[])
30 {
31     std::stringstream ss;           // punkt zapisu
32     zwierzę z1, z2, z3;             // zmienne na odczyt
33     zwierzę kot{1, "misza", 4};      // pierwsza zmienna testowa
34     zwierzę papuga{2, "papug", 2};   // druga zmienna testowa
35     zwierzę pająk{3, "wojtuś", 6};   // trzecia zmienna testowa
36
37     ss << kot << papuga << pająk;   // zapis zmiennych do strumienia
38     std::cout << ss.str() << std::endl; // wyświetlenie strumienia
39     ss >> z1 >> z2 >> z3;           // odczyt
40     std::cout << z1 << z2 << z3 << std::endl; // wyświetlenie za pomocą standardowego wyjścia
41     global_logger.error() << z1 << z2 << z3 << logger::endl;
42     // ^ wyświetlenie za pomocą klasy z wyłączenie
43     // przeladownym operatorem lewego przesunięcia bitowego
44 }

```

Rys. 29. Przykład użycia struktur *ser* oraz *cser*

```

1 misza 4 2 papug 2 3 wojtuś 6
1 misza 4 2 papug 2 3 wojtuś 6
[ERROR][2021-06-05T18:00:06.553468][global_logger] 1 misza 4 2 papug 2 3 wojtuś 6

```

Rys. 30. Wyjście z powyższego przykładu (Rys. 29)

Przykład z powyższego zrzutu ekranu (Rys. 29) wraz z wyjściem (Rys. 30), prezentuje użycie serializacji oraz deserializacji. Nie ma potrzeby pisania dodatkowych funkcji, które obsługują proces wkładania, oraz wyciągania danych ze strumienia. Rozwiązanie to daje sporą elastyczność w zakresie tworzenia własnych modyfikatorów. Kolejny przedstawiony przykład, zaprezentuje możliwość nadpisania domyślnej reprezentacji.

```

1 misza 4
zwierzę_impl[ 1, misza, 4 ]
2 jerzy 4
ładne_zwierzę_impl[ jerzy ]

```

Rys. 31. Wyjście z poniższego przykładu (Rys. 32)

```

29 struct ładne_zwierzę_impl : public serial_helper_t
30 {
31     ser<ładne_zwierzę_impl::_, zwierzę> z;
32
33     You, seconds ago | 1 author (You)
34     struct custom_pretty_print{ // nadpisanie domyślnej reprezentacji
35         template<typename os_t> custom_pretty_print(os_t& os, const zwierzę& x) { os << x().nazwa(); }
36     };
37     using ładne_zwierzę = cser<ładne_zwierzę_impl::z>;
38
39     int main(int argc, char* argv[])
40     {
41         zwierzę kot{1, "misza", 4}; // pierwsza zmienna testowa
42         ładne_zwierzę jeź; // druga zmienna testowa
43         jeź().z = zwierzę{2, "jerzy", 4};
44
45         std::cout << kot << std::endl;
46         std::cout << pretty_print{kot} << std::endl;
47         std::cout << jeź << std::endl;
48         std::cout << pretty_print{jeź} << std::endl;
49     }

```

Rys. 32 Przykład nadpisania domyślnego wywołania

Dopisanie struktury powyżej (Rys. 32), w linii 34, o nazwie ustalonej przez modyfikator *pretty\_print*, używanego w obecnym projekcie jako deweloperskie narzędzie do drukowania reprezentacji obiektów wraz z typami, umożliwiło zmianę wyświetlanej wartości (Rys. 31), z wszystkiego, na wyłącznie jedno pole.

Zadaniem metody *accept*, w przypadku struktury *ser*, jest przekazanie otrzymanego wizytatora do kolejnego pola, które jest wskazane za pomocą wbudowanego w typ, statycznego wskaźnika, zapisanie wyniku z wizytatora, do jego instancji, a następnie wywołanie go dla siebie. Wymagania, jakie musi spełnić wizytator są określone za pomocą konceptu, co w przypadku niespełnienia wymagań przez wizytator, prowadzi do klarownego błędu, jeszcze podczas kompilacji, co jest zaletą, oferowaną przez najnowszy standard języka.

Zadanie metody akceptującej w kontenerze *cser*, jest inne. W przeciwieństwie do metody z struktury *ser*, wpisuje do pustego wskaźnika, adres przechowywanej przez siebie klasy (reprezentuje wskaźnik *this*), a następnie przekazuje wizytator do ostatniego pola w przechowywanej klasie. Referencje do niego, podobnie jak klasa *ser*, posiada dzięki statycznemu wskaźnikowi, zawartego w szablonie.

### 3.9.2. Problematyka

Największym problemem podczas implementacji, było zapewnienie odpowiedniego przekierowania konstruktorów oraz operatorów, umożliwiających wygodne korzystanie z klasy enkapsulowanej klasy. Złożoność tkwi w możliwości przychodzących do operatorów oraz konstruktorów danych. Mogą pojawić się, bowiem dane zapakowane zarówno w strukturę *ser* jak również strukturę *cser*. Nie można dopuścić do sytuacji, gdzie zostanie podjęta próba przypisania opakowanego typu, do typu podstawowego, co doprowadzi do błędu kompilacji.

Kolejnym problemem, podczas tworzenia kodu wymienionych klas, były ciężkie w zrozumieniu błędy, często niemieszczące się na obróconym pionowo ekranie. Doprowadzało to do absurdalnych sytuacji, gdzie stos błędnego instancjonowania, przewyższał wielokrotnie długość całej implementacji. Wymusiło to opracowywanie rozwiązania w osobnym projekcie, a w drugiej kolejności wklejenie rozwiązania do niniejszego projektu.

Projekt przechodził bardzo burzliwy proces ewolucji, który początkowo nie zakładał innych funkcjonalności, niż serializacja, deserializacja oraz drukowanie wraz z typami [41]. Przejście na kolejną warstwę abstrakcji, umożliwiło tworzenie nowych zastosowań, możliwe, że nawet w pełni statycznych operacji iterowanych po polach struktury. Proces ewolucji kodu, można zaobserwować, przez nazewnictwo użyte dla struktur. Początkowo, gdy struktury te miały wyłącznie służyć do serializacji, nazwa *ser* jako skrótowiec od angielskiej nazwy serializacji, miał jak najbardziej sens. To samo tyczy się nazwy *cser*, której wytłumaczenie to serializacja klasy w języku angielskim. Aktualnie nazwa powinna odnosić się do możliwości iteracji po typach.

Ostatni problem dotyczy samych modyfikatorów. Podczas implementacji modyfikatora *pretty\_print*, zachodził częsty problem błędnego przejścia na modyfikator zwykłej serializacji, co wiązało się z nieczytelnym, wyjściem. Spowodowało to konieczność wprowadzenia oznaczania typów (ang. *type tagging*). Zostało to zaimplementowane, za pomocą prostych struktur, zawierających wyłącznie referencje na oznaczany typ, co powoduje wymuszenie określonej ścieżki podczas instancjonowania typów przez kompilator.

### 3.9.3. Wady i zalety

Definitywną zaletą skorzystania z opisanego mechanizmu jest brak konieczności dedykowanego pisania mechanizmów serializacji oraz deserializacji, dla każdego z typów. Umożliwia to ograniczenie pisanego kodu oraz z uwagi na statyczne rozwiązywanie, redukcję generowanego kodu, jeżeli kompilator nie wykryje użycia dla danych modyfikatorów. Dodatkowo z uwagi na statyczność tego kodu oraz implementacja z wykorzystaniem szablonowego meta-programowania umożliwia kompilatorowi bardzo głębokie optymalizacje.

Konstrukcja oparta o prosty kontener, uzależnia możliwość korzystania z kopii binarnej od przechowywanego typu, co w przypadku rozwiązań z sporą ilością wykonywanych kopii jest kluczowe i pozwala wysoce przyspieszyć mechanizmy kopiujące. Zrezygnowanie z mechanizmu kompilacji pozwala również na binarne kopiowanie pomiędzy typami o tych samych wartościach, co zdarza się wykorzystywać w przypadku niektórych implementacjach na niższym poziomie.

Utworzenie interfejsu umożliwia wprowadzenie naturalnych metod dostępu (ang. *accessors*). Ich użycie w projekcie daje możliwość sprawniejszego debugowania oraz może poprawić stan hermetyzacji kodu.

Pojawienie się dłuższych nazw typów, wymusza na użytkowniku biblioteki stosowanie aliasów, co ma bezpośredni wpływ na czytelność kodu oraz intencje jego wprowadzenia.

Opisane tutaj rozwiązanie, jest wolne od makr. Jest to kuszące rozwiązanie, dla projektów celujących w wysoki poziom kodu, a także unikanie przestarzałych rozwiązań.

Naczelną wadą tego podejścia jest redukcja przejrzystości kodu, poprzez utrudniony dostęp do bardziej zagnieżdżonych pól struktur i klas. Wymusza to na użytkowniku klasy stosowanie referencji, jako aliasy do bardziej położonych niżej w hierarchii danych w przechowywanych strukturach. Zostało to podkreślone na poniższym zrzucie ekranu (Rys. 33), gdzie większość inwokacji operatora okrągłych nawiasów to wyluskanie pola bardziej zagnieżdżonego.

```

void publication_widget_item::display()
{
    if(!m_publication.expired())
    {
        // aliases
        const auto& p      = (*m_publication.lock().get())();
        const auto& ref     = (*p.reference())();
        const auto& matched = p.matched().data();

        // creating string
        QString result = QString::fromStdString(std::to_string(ref.year()));
        result.insert(0, "[ ");
        result += " ] ";
        result += QString::fromStdString(ref.title().raw);
    }
}

```

Rys. 33. Przykład dostępu do głębiej umieszczonych pól w strukturach

Kolejną wadą jest wydłużony czas kompilacji. Spowodowane jest to potrzebą kompilatora na głębsze przeanalizowanie struktur. Czas ten potrafi być znacząco wydłużony, gdy zostaje wybrany wyższy stopień optymalizacji, wśród udostępnianych przez kompilator.

Ostatnią z znaczących wad, jest pojawienie się bardzo długiego błędu, przy niepoprawnym zapisie. Zastosowanie konceptów, znacznie zredukowało ilość miejsc, gdzie takie błędy mogą się pojawić. Warto zwrócić uwagę, że jest to nie jest wyłączny problem, tej biblioteki, lecz wszystkich rozwiązań bazujących na podejściu meta-programowania.

### 3.10. Logowanie

Celem zapewnienia wygodnego oraz pomocnego w analizie problemów systemu logowania, potrzebne było napisanie klasy zapewniającej szeroką elastyczność w zakresie obsługiwanych formatów.

Oprócz możliwości wyświetlania, powinna zostać obsłużona możliwość wyróżniania komunikatów różnego pochodzenia oraz ustalaniu, które komunikaty mają być pokazywane. Umożliwi to zachowanie porządku oraz zapewni wygodę podczas przeszukiwania wyjścia z aplikacji.

Wszystkie te cechy spełnia klasa *logger*, która stanowi złożony interfejs dla standardowego strumienia wyjścia. Implementacja opiera się również między innymi o strumień standardowy, umożliwiając wygodne korzystanie, niczym z obiektu *cout*.

Zapewnienie sporej ilości obsługiwanych typów, zostało zrealizowane, poprzez wykorzystanie szablonów. Jedynym wymogiem stawianym przed typem podczas specjalizowania funkcji, jest możliwość umiejscowienia go w standardowym strumieniu.

Potrzeba różnorodnego wyjścia została zrealizowana dzięki bibliotece rang, oraz serii statycznych funkcji, których wywołania w dedykowanych funkcjach buduje odpowiednią preambułę, kolor tekstu oraz tła, a następnie dodaje komunikat wybrany przez użytkownika. Dodatkowo, przy kompilacji typu *Debug* lub *Release with Debug Symbols*, przed każdą wiadomością dołączany jest fragment stosu zawierający nazwę klasy oraz nazwę funkcji lub metody wraz z numerem liniiki, z którego pochodzi wpis. Jest to bardzo pomocne, w momencie analizy błędu.

Klasa oferuje w sumie cztery tryby wyświetlania informacji: błąd, ostrzeżenie, informacja, wpis deweloperski. Każdy z tych trybów cechuje się innym kolorem, a także innym początkiem wpisu. Domyślnie wyświetlane są tylko wiadomości z ostrzeżeniami i błędami, żeby nie zaśmiecać konsoli użytkownika, ale dzięki dostarczonemu interfejsowi obsługi poziomu wpisów, możliwa jest wygodna i dynamiczna ich modyfikacja. Oprócz podstawowych funkcji dostępowych, dostarczono również strukturę pełniącą rolę strażnika, która w swoim destruktorze przywraca poprzedni poziom szczegółowości wpisów.

Dodatkowo jest możliwość tworzenia dedykowanych loggerów, dla dowolnej klasy, wystarczy skorzystać z unikalnego dla języka C++ wzorca projektowego CRTP [42], co umożliwi dodawanie wpisów, z dodatkowym przedrostkiem odnoszącym się do nazwy klasy. Wzorzec ten jest stosowany powszechnie w projekcie, a przykład użycia pokazano poniżej (Rys. 34) w lini numer 142.

```
133
134 | /**
135 |  * @brief This class is wrapper for mangling and demangling given strings
136 |  *
137 |  * @tparam string_type string to operate on (str or u16str)
138 |  * @tparam string_view_type interface to use (str_v or u16str_v)
139 |  */
    You, a week ago | 1 author (You)
140 | template<acceptable_string_types_req string_type           = str,
141 |         acceptable_string_view_types_req string_view_type = str_v>
142 | class demangler : public Log<demangler<string_type, string_view_type>>
143 | {
144 |     using Log<demangler>::log;
145 | }
```

Rys. 34. Przykład użycia

### 3.11. Testy

```
[raidg@Papug][127] ~/proj/polsl/build $ ./tests/tests
[INFO][2021-06-06T01:15:18.091520][testbase_logger] entering `orcid_tests` suite
[INFO][2021-06-06T01:15:18.091629][testbase_logger] entering `polish_name` suite
[INFO][2021-06-06T01:15:18.092656][testbase_logger] entering `person_tests` suite
[INFO][2021-06-06T01:15:18.092675][testbase_logger] entering `string_splitter_tests` suite
All tests passed (82 asserts in 24 tests)
```

Rys. 35. Uruchomienie oraz wyniki wykonanych testów

Istotną częścią tego projektu były testy. Kluczowe funkcje, takie jak klasa *demangler*, czy weryfikacja podanych łańcuchów zostały poddane testowaniu, wieloma przypadkami (Rys. 35). Niestety zdecydowana większość kluczowych funkcji, nie jest w najmniejszym stopniu sprawdzona.

Najdalej posunięty w abstrakcji test, to weryfikacja spójności w serializacji deserializacji obiektu reprezentującego autora. Brak sprawdzeń dla publikacji, systemu ich dopasowania czy weryfikacji danych w generowanych raportach to jedne z największych bolączek tego projektu. Ograniczenie czasowe spowodowały spore zaniedbanie w tym zakresie.

Obecnie występują wyłącznie dwa rodzaje testów: jednostkowy oraz jednostkowy negatywny. Pierwsze sprawdzają, czy w wyniku przetworzenia, otrzymywane są prawidłowe wartości. Drugie weryfikują pojawienie się określonego błędu (Rys. 36).

```
"case_02"_test = [] {
    const auto expect_assertion = [](const str_v& x) {
        ut::expect(
            ut::throws<core::exceptions::assert_exception<u16str>>([&] { polish_name_t{x}; }));
    };

    expect_assertion(invalid_pn_01);
    expect_assertion(invalid_pn_02);
    expect_assertion(invalid_pn_03);
    expect_assertion(invalid_pn_04);
    expect_assertion(invalid_pn_05);
    expect_assertion(invalid_pn_06);
    expect_assertion(invalid_pn_07);
    expect_assertion(invalid_pn_08);
    expect_assertion(invalid_pn_09);
    expect_assertion(invalid_pn_10);
    expect_assertion(invalid_pn_11);
    expect_assertion(invalid_pn_12);
    expect_assertion(invalid_pn_13);
};
```

Rys. 36. Przykładowy negatywny test, sprawdzający pojawienie się odpowiedniego błędu

### **3.12. Narzędzia i produkty uboczne**

Rozwijając przedstawiony projekt, powstała paleta różnych narzędzi wspomagających oraz kilka incydentów w innych projektach. Ponieważ udział tych elementów nie pozostał obojętny dla projektu, a nie mają bezpośredniego wpływu na działanie aplikacji, zostały wydzielone do osobnego rozdziału.

#### **3.12.1. Generowanie bibliotek**

Struktura projektu, określona katalogami, ma charakter drzewiasty oraz silnie przewidywalny. Konieczność tworzenia katalogów oraz trzymanie spójności konwencji wymusiło napisanie skryptów automatyzujących proces dodawania kolejnych bibliotek.

Niespełnione aspiracje niniejszego projektu do rangi statusu projektu w pełni międzyplatformowego wymusiły skorzystanie z języka, który również posiada możliwość ewaluacji na wielu platformach. Biorąc pod uwagę łatwość obsługi, skryptowy język *Python*, został wybrany jako język wsparcia.

Skrypty zostały umieszczone w dedykowanych katalogach, podkatalogu *gen*. Pierwszy znajduje się w podkatalogu o nazwie *library*, kolejny w *windows*. Nazwy skryptów jasno definiują ich przeznaczenie. Każdy z nich przyjmuje jeden parametr i jest nim odpowiednio nazwa biblioteki i nazwa nowego okna dialogowego.

Każdy z tych drobnych programów wykonany z odpowiednim argumentem, tworzy katalog o podanej nazwie w odpowiednim miejscu w projekcie. Następnie generowana jest odpowiednia struktura, a na końcu generowane są pliki, opierając się o dostarczone, wejściowe. Cały proces jest błyskawiczny, a przede wszystkim sprawdzony, ponieważ zdecydowana większość ujętych bibliotek została stworzone przy pomocy opisanych skryptów.



### 3.12.2. Asysta z systemu kontroli budowy

Wspomniany wyżej system kontroli wersji *cmake*, zawiera szereg ciekawych funkcjonalności, umożliwiających wsparcie procesu tworzenia kodu.

#### 3.12.2.1. Pilnowanie aktualności bibliotek

```
[raidg@Papug][0] ~/proj/polsl/build $ rcmake ../antybiurokrata/
-- The CXX compiler identification is GNU 10.2.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
git found, updating submodules
Cloning into '/home/raidg/proj/polsl/antybiurokrata/libraries/QtXlsxWriter'...
Submodule path 'libraries/QtXlsxWriter': checked out '627aeb82678f268432c49bec2eddddf7f9e2365ce'
-- Looking for C++ include pthread.h
-- Looking for C++ include pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - found
```

Rys. 37. Pobranie podmodułu przez system kontroli budowy

Integracja z programem git zapewnia, że przed rozpoczęciem generowania plików *MAKEFILE*, zostaną pobrane najnowsze wersje podmodułów. Celem demonstracji, usunięto jeden z modułów. Został on od razu pobrany jeszcze przed rozpoczęciem budowy, umożliwiając pozytywne jej zakończenie, co widać na powyższym przykładzie (Rys. 37).

### 3.12.2.2. Zapobieganie przypadkowej budowie

```
[raidg@Papug][0] ~/proj/polsl/antybiurokrata $ cmake .
-- The CXX compiler identification is GNU 10.2.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Error at CMakeLists.txt:8 (message):
  In-source builds are not allowed, remove CMakeFiles and CMakeCache.txt

-- Configuring incomplete, errors occurred!
See also "/home/raidg/proj/polsl/antybiurokrata/CMakeFiles/CMakeOutput.log".
```

Rys. 38. Zatrzymanie programu *cmake*, po wykryciu budowy w źródle

Podczas tworzenia oprogramowania, często zdarzają się pomyłki. Kompilacja w źródle jest jedną z bardziej bolesnych. System budowy, w czasie generowania plików budowy, tworzy nową strukturę katalogów, zbliżoną do tej w projekcie oraz generuje tam swoje pliki. Usuwanie takich plików jest bardzo męczące, a wykorzystanie kontroli wersji, może okazać się bardziej kłopotliwe niż ręczne usuwanie wszystkich plików.

Dodanie blokady (Rys. 38), która zapobiega przypadkowej kompilacji w złym miejscu, potrafi uratować sumarycznie sporo godzin cennego czasu. Niestety część zmiennych, w tym ścieżka do źródeł, jest dostępna, dopiero po wygenerowaniu dwóch plików. Potrzeba ścieżki, do sprawdzenia, czy kompilacja zachodzi w źródłach, uniemożliwia prewencję generacji tych dwóch plików, jednakże, aby wspomóc proces czyszczenia o pomyłce, wyświetlany jest komunikat z nazwami obiektów do usunięcia.


Osoby, które nie są zainteresowane tworzeniem nowego kodu, oraz mają nawyk budowania aplikacji w źródle, mają taką możliwość, poprzez dodanie odpowiedniej flagi podczas uruchomienia polecenia *cmake*.

### **3.12.2.3. Generowanie dokumentacji**

Projekt zakładający przyszły rozwój, nie powinien być pozbawiony dokumentacji. Zastosowanie automatycznej dokumentacji, podczas pisania kodu pomija konieczność tworzenia osobnego dokumentu z wrywkami kodu oraz koniecznością trzymania spójności stylu w całej dokumentacji.

Wykorzystane narzędzie *Doxygen*, wymaga uruchomienia polecenia oraz posiadania pliku wsadowego, zawierającego instrukcje o wyglądzie dokumentacji, oraz plikach, jakie mają zostać uwzględnione, lub wykluczone z procesu generowania. Dbanie o aktualność tego pliku jest czynnością, która może zostać zautomatyzowana.

Dodanie nowego celu budowy, który jest ukrytym wywołaniem polecenia programu *Doxygen*, z punktu widzenia użytkownika enkapsuluje cały proces oraz sprowadza go do regularnego wywołania polecenia *make docs* w katalogu budowy. Zamknięcie całości w wymienionym poleceniu daje możliwość podmiany generowania dokumentacji (Rys. 39), bez konieczności zmiany sposobu wywołania.

 <div> <b>antybiurokrata</b>  <small>This project aims for creating diffs between different science articles aggregation sites</small> </div>	
<b>ANTYBIUROKRATA</b>	
<a href="#">Main Page</a>	<a href="#">Related Pages</a>
<a href="#">Namespaces</a>	<a href="#">Classes</a>
<a href="#">Files</a>	<a href="#">Examples</a>
<b>Class List</b>	
Here are the classes, structs, unions and interfaces with brief descriptions:	
<div> <div>▼</div> <div>N</div> <div>__dummy</div> </div>	
<div> <div>C</div> <div>global_logger</div> </div>	
<div> <div>▼</div> <div>N</div> <div>core</div> </div>	Base namespace for whole program
<div> <div>▼</div> <div>N</div> <div>detail</div> </div>	Definition of engine internal helper types
<div> <div>C</div> <div>translation_value_t</div> </div>	Type used in internal map in letter_converter as value
<div> <div>C</div> <div>depolonizator</div> </div>	Polish translation
<div> <div>C</div> <div>universal_getter</div> </div>	Universal processor for alternative data sources
<div> <div>▼</div> <div>N</div> <div>exceptions</div> </div>	Basic definitions and tools for throwing exceptions
<div> <div>C</div> <div>exception</div> </div>	Simplest exception, use this to catch all exceptions
<div> <div>C</div> <div>exception_base</div> </div>	Basic exception,
<div> <div>C</div> <div>assert_exception</div> </div>	Default exception, that is raised on failed check
<div> <div>C</div> <div>not_found_exception</div> </div>	This exception should be thrown if something is not found
<div> <div>C</div> <div>pointer_is_null</div> </div>	This exception should be thrown if given pointer is nullptr, but shouldn't
<div> <div>C</div> <div>tee_exception</div> </div>	Same as exception, but additionally prints reason to stdout, usefull, if extended log is required
<div> <div>C</div> <div>require</div> </div>	Alternative to assertion
<div> <div>C</div> <div>require_not_nullptr</div> </div>	Struct for checking is pointer nullptr
<div> <div>C</div> <div>error_report</div> </div>	Object representation of error summary
<div> <div>▼</div> <div>N</div> <div>network</div> </div>	Implementation of network classes
<div> <div>►</div> <div>N</div> <div>detail</div> </div>	
<div> <div>C</div> <div>bgpolsl_adapter</div> </div>	Data collector for bg.polsl.pl
<div> <div>C</div> <div>connection_handler</div> </div>	Basic interface for handling http requests
<div> <div>C</div> <div>orcid_adapter</div> </div>	Data collector for orcid
<div> <div>C</div> <div>scopus_adapter</div> </div>	Data collector for scopus
<div> <div>▼</div> <div>N</div> <div>objects</div> </div>	Objects that should be used for comparing and IO operations
<div> <div>►</div> <div>N</div> <div>detail</div> </div>	
<div> <div>►</div> <div>N</div> <div>processing_details</div> </div>	Templates of actions on complex types
<div> <div>▼</div> <div>N</div> <div>orm</div> </div>	Converters from adapters output to objects
<div> <div>C</div> <div>persons_extractor_t</div> </div>	Constructs person_t objects, by visiting output from adapters
<div> <div>C</div> <div>publications_extractor_t</div> </div>	Constructs publication_t objects, by visiting output from adapters

Rys. 39. Fragment spisu wszystkich udokumentowanych klas w projekcie

### 3.12.2.4. Formatowanie

Dodanie automatycznego formatowania podlega jest umotywowane dokładnie tymi samymi pobudkami, co tworzenie automatycznej dokumentacji. Konieczność pamiętania składni polecenia *clang-format*, a także konieczność aktualizacji plików wejściowych sprawia, że przestaje być to komfortowe w dłuższej perspektywie.

Polecenie *make format*, zamyka cały proces trzymania spójnego formatowania w regularnej inwokacji, przed dodaniem kolejnych zmian. Mnogość programów pozwalających formatować kod, nie zamyka możliwości zmiany aplikacji utrzymującej porządek w kodzie. Zamiana tego typu podobnie jak w przypadku dokumentacji, będzie nie widoczna z punktu widzenia programisty wykonującego polecenie.

### **3.12.3.      Udział w innych projektach**

#### **3.12.3.1. Wikipedia**

Przeglądanie literatury, a także podobnych rozwiązań, doprowadziło do znalezienia artykułu *Template Meta-Programming* w ramach otwartej, darmowej encyklopedii – Wikipedia. Przegląd zamieszczonego spisu treści, następnie artykułu doprowadził do obserwacji braku wpisu odnośnie do opisanego meta-programowania.

Braki te zostały uzupełnione [30], wraz z przykładem rozwiązania problemu *FizzBuzz*, który jest często wykorzystywany w celach dydaktycznych. Zmiana została przypieczętowana drobnymi poprawkami, większości w zakresie gramatyki oraz użytego słownictwa, przez anonimowego wikipedystę.

Zamieszczony wpis bibliograficzny datą referuje na dodany wpis, który od tamtego momentu został wielokrotnie zmodyfikowany, przez inne różne osoby.

#### **3.12.3.2. Biblioteka matematycznej komparacji ciągów znakowych**

Porównywanie ciągów znakowych może zostać dokonane zarówno jako sztywne porównanie użytych znaków w odpowiedniej kolejności, oraz za pomocą systemu oceny, do którego używa się skomplikowanych algorytmów matematycznych.

Chęć wykorzystania jednej z publicznych bibliotek [43], spotkała się z problemem braku kompatybilności systemów budowy. Korzystając z mechanizmów społecznościowych portalu GitHub, została utworzona prośba wdrożenia zaimplementowanych zmian do głównego projektu. Mimo kontaktu z autorem zmiany są wciąż niedodane do głównego repozytorium.

Chęć upewnienia się, co do poprawności zamieszczonych zmian, przez ich akceptację, ze strony autora projektu niestety, wyszły poza zakres czasu oddania pracy podsumowującej projekt inżynierski. Zastosowanie tej biblioteki zostanie szerzej wyjaśnione w kolejnym rozdziale 4.1.

### 3.13. Konfiguracja stanowiska

Konfiguracja stanowiska, została przedstawiona w tradycyjnej dla projektów z otwartym kodem formie, czyli w pliku *README.md*. Tutaj zostanie ona rozwinięta oraz opisana przy założeniu poprawnej instalacji wybranej platformy – *Manjaro GNU/Linux*.

Pierwszym krokiem, praktycznie w pełni automatycznym jest zainstalowanie środowiska Qt, co na pewno zainstaluje wszystkie wymagane biblioteki oraz moduły. Żeby móc tego dokonać, należy pobrać instalator z oficjalnej strony projektu Qt, założyć konto (lub użyć istniejącego), pobrać instalator, a następnie przejść przez proces instalacji z domyślnymi ustawieniami. Jeżeli wystąpi problem ze znalezieniem przycisku do pobierania oprogramowania, można skorzystać z hiperłącza umieszczonego w pliku *README.md*, który odwołuje się bezpośrednio do pobierania instalatora.

Kolejnym etapem, najdłuższym, jest pobranie biblioteki *boost* w wersji 1.75, z oficjalnej strony projektu [20]. Po wypakowaniu źródeł, należy wejść do katalogu, a następnie wykonać skrypt *bootstrap*. Nastąpi kompilacja kolejnego programu – *b2*. Po uruchomieniu, rozpocznie się proces kompilacji z wykorzystaniem wszystkich dostępnych zasobów maszyny. Instalację należy dokończyć, poprzez stworzenie dowiązań symbolicznych ścieżek wypisanych na ekranie katalogów zawierających pliki nagłówkowe oraz skompilowane biblioteki, odpowiednio do */usr/include/boost* oraz */usr/lib/boost*. Dobrym pomysłem jest również stworzenie zmiennej środowiskowej z głównym katalogiem biblioteki *boost*, czyli katalogu, w którym znajdziemy wygenerowany program *b2*, o nazwie *BOOST\_ROOT*. Ułatwi to systemowi budowy znalezienie odpowiednich modułów. Rekomenduje się dodanie wyżej wymienionej zmiennej do pliku */etc/enviroment*.

Trzecią biblioteką na liście nosi nazwę *rang*. Jediną czynnością wymaganą po jej pobraniu z oficjalnego repozytorium [23], to stworzenie dowiązania symbolicznego */usr/include/rang* do katalogu *rang/include* umieszczonego w miejscu pobrania biblioteki.

Czwarta biblioteka, *drogon*, jest nieco bardziej skomplikowana. Po pobraniu za pomocą programu *git*, należy w katalogu projektu uruchomić polecenie *git submodule –init –recursive*. Pobierze to wszystkie zależności projektu, co jest niezbędne do prawidłowego działania. Następnie należy stworzyć katalog, przykładowo o nazwie *build* oraz wejść do niego. Wykonanie kolejno poleceń: *cmake ..* oraz *make –j\${nproc}*. Zbuduje całą bibliotekę, wykorzystując wszystkie dostępne zasoby maszyny, może to chwilę zająć. Ostatnim krokiem jest wykonanie polecenia *make install* z uprawnieniami administratora, co zainstaluje bibliotekę w przewidziany, przez autorów sposób. Jeżeli, podczas budowy właściwego projektu wystąpią problemy ze znalezieniem biblioteki, można spróbować przenieść instalację, znajdującą się w katalogu */usr/local/include* oraz */usr/local/lib* odpowiednio do katalogów */usr/include*, oraz */usr/lib*. Przenieść należy zarówno bibliotekę *drogon*, jak i jej zależność – bibliotekę *trantor*. Należy podkreślić przenoszenie, ponieważ stworzenie dowiązania symbolicznego, może spowodować dodatkowe problemy, natury systemowej.

Ostatnią biblioteką, pozostałą do zbudowania jest *boost-ext/ut*. Po pobraniu jej z oficjalnego repozytorium [21] wystarczy zbudowaniu i zainstalowaniu w analogiczny do poprzedniego sposób. Nie zidentyfikowano, tutaj dodatkowych problemów.

Po wykonaniu powyższej konfiguracji pozostaje wyłącznie pobrać kod programu z repozytorium, stworzyć katalog budowy, a następnie uruchomić dwie komendy: *cmake ..* oraz *make –j\${nproc}*. Pierwsza komenda, powinna znaleźć wszystkie wcześniej zainstalowane biblioteki oraz pobrać zależności. Druga komenda, dokona kompilacji. Po zakończeniu budowy, użytkownik posiada trzy możliwości. Pierwszą jest uruchomienie testów, co można zrobić za pomocą komendy *./tests/tests*. Drugą możliwością jest wygenerowanie dokumentacji, co jest możliwe, korzystając z wcześniej wymienionej komendy – *make docs*. Wygenerowana strona internetowa znajduje się w katalogu *docs/html*. Otworzenie za pomocą przeglądarki pliku *indeks.html*, pozwoli swobodnie przeglądać dokumentację. Warto wspomnieć, że dokumentację można wygenerować zaraz po poleceniu *cmake*, nie jest wymagana dość długa kompilacja. Ostatnią możliwą do wykonania akcją, jest uruchomienie aplikacji za pomocą komendy: *./antybiurokrata*.

## 4. Podsumowanie

### 4.1. *Perspektywy rozwoju*

Jednym z celów tej pracy, było zapewnienie możliwości rozwoju. Zostało to osiągnięte za pomocą różnych metod, technik oraz rozwiązań. Podczas wytwarzania kodu, powstało parę pomysłów, które głównie z uwagi na ograniczenia czasów, nie zostały zaimplementowane, lub rozwinięte. Zostaną one tutaj opisane jako punkt wyjścia przyszłych prac.

Kluczowym elementem działania aplikacji jest porównywanie. Aktualnie cały ciężar komparacji odbywa się dychotomicznie, co ma swoje zalety, jednakże nie bierze pod uwagę istotnego czynnika – błędu ludzkiego. Wielokrotnie podczas weryfikacji algorytmu zdarzała się sytuacja, braku stworzenia relacji między na pozór identycznymi tytułami. Głębsza analiza, szybko weryfikowała, istnienie pojedynczych literówek. Eliminacja tego problemu, powinna być kompleksowa oraz polegać na eliminacji podejścia jednostanowego, określanego za pomocą zmiennej logicznej, na podejście oparte o spektrum ocen. Oznacza to rozszerzenie klasy *detail\_publications\_summary\_t* (Rys. 23) o dodatkową kolekcję zawierającą możliwe dopasowania, oraz wyróżnienie ich w reprezentacji graficznej, za pomocą odpowiedniego koloru.

Celem stworzenia systemu opartego o oceny, potrzeba jest zmiana podejścia do porównania ciągów znaków. Rozwiązanie przynosi zastosowanie jednego z dwóch spośród wielu algorytmów, pomiarów ciągów tekstowych (ang. *string metric*) [44]. Pierwszym, preferowanym, jest algorytm *Damerau–Levenshtein* [45], który definiuje rozbieżność między dwoma ciągami znaków, jako liczbę minimalną liczbę niezbędnych edycji, aby uzyskać dwa identyczne teksty. Drugi algorytm *Levenshtein* [46], również tak definiuje odległości, pomiędzy tekstami, lecz zdefiniowana liczba operacji, jaką można wykonać jest mniejsza, w porównaniu do poprzedniego algorytmu. Klasyfikacja, biorąca pod uwagę nieścistości, musi wiązać się z dodaniem informacji dla użytkownika o powodzie, braku pełnego dopasowania. Znajdzona biblioteka zawierająca implementacje obu algorytmów niestety nie posiada wsparcia, dla obecnego w niniejszym projekcie systemu budowy, co zostało wyjaśnione w rozdziale 3.12.3.2.

Niezbędnym elementem, będzie również refaktoryzacja, implementacji serializacji, głównie z uwagi na gwałtowny rozwój, nazewnictwo nieoddające w pełni zastosowania oraz spore wycinki kodu znajdującego się pod komentarzem. Dodatkowo, istnienie zaledwie dwóch sprawdzeń, dla tak rozbudowanej funkcjonalności, to zdecydowanie za mało.

Potrzeba zwiększenia ilości testów, nie dotyczy wyłącznie serializacji. Dotyka ten problem w szczególności implementacji sieciowych, a także klas porównujących, weryfikujących oraz tworzących relacje pomiędzy instancjami.

Dodatkowym elementem z zakresu serializacji, jest rozwinięcie możliwych zastosowań tej biblioteki, poprzez stworzenie nowych modyfikatorów umożliwiających generowanie raportów w formie *xlsx*, za ich pomocą. Kolejnym etapem, niewątpliwie, będzie stworzenie modyfikatora, pozwalającego na generowanie zapytań w języku *SQL*. Stworzyłoby to podwaliny pod połowicznie statyczny obiektowy model bazodanowy. Przeznaczeniem tej implementacji, niewątpliwie jest eksport do modułu, który będzie możliwy do dołączenia do tego oraz wielu innych projektów.

Wcześniej wspomniany pod koniec rozdziału 3.8.1 argument przemawiający za enkapsulacją logiki do pojedynczej klasy, nie powinien zostać w tym rozdziale pominięty. Konieczność stworzenia interfejsu konsolowego, wydaje się posiadać przynajmniej równie wysoki priorytet, co stworzenie graficznego interfejsu użytkownika. Wpływają na to dwa czynniki. Pierwszym jest wybrane środowisko, jakim jest dystrybucja systemu *GNU/Linux*, gdzie przeważająca ilość oprogramowania, jest dostępna wyłącznie lub dodatkowo za pośrednictwem interfejsu konsolowego. Kolejnym czynnikiem wywierającym presję, stworzenia takiego interfejsu, to możliwość planowania zadań regularnego sprawdzania serwisów, przykładowo z wykorzystaniem narzędzia *cron* oraz przesyłanie odpowiednich raportów do na przykład listy subskrybentów. Innym zaproponowanym interfejsem, jest interfejs sieciowy. Jest to absolutna podstawa, pod stworzenie integracji, pomiędzy innymi aplikacjami, które chciałyby skorzystać z wyników ewaluacji.



## 4.2. *Wnioski*

- Stopień złożoności pozyskiwania danych ze źródła mocno od zależy od formatu oferowanych danych
- Zastosowanie interaktywnej, rozbudowanej dokumentacji wraz z przykładami ma spory wpływ na szybkość i jakość wytwarzania kodu
- Pisanie oprogramowania wraz z testami wymaga więcej czasu, ale redukuje ilość pomyłek i zwiększa stabilność oprogramowania
- Tworzenie dokumentacji, wykorzystując pozostawione specjalne komentarze, oszczędza sporo czasu oraz pomaga w czytaniu kodu
- Stosowanie wzorców projektowych, prowadzi do poprawienia czytelności
- Wykorzystanie wizytatora jako wzorca projektowego zwiększ modularność projektu
- Mechanizm sygnałów i slotów umożliwia sprawną synchronizację danych pochodzących z różnych wątków
- Środowisko Qt oferuje elastyczne techniki tworzenia graficznych interfejsów użytkownika
- Wykorzystanie zewnętrznych bibliotek, zawierających testy na oferowane przez siebie funkcjonalności zwiększa stabilność pisanego kodu
- Tworzenie kodu z wykorzystaniem szablonowego meta-programowania jest wymagające, lecz pozwala umożliwić kompilatorowi wyliczyć lub zoptymalizować spore kawałki generowanego kodu
- Możliwym jest pisanie złożonych aplikacji w języku C++, pomijając wykorzystanie makr, wymusza to na programiście stosowania wzorców projektowych oraz nieszablonowego myślenia w rozwiązaniach opartych o szablony
- Dbając o jednolite formatowanie kodu w całej aplikacji, zwiększa się spójność podczas jego czytania oraz nie ma potrzeby ciągłego dostosowywania się do różnego stylu zapisu
- Problematyka serializacji jest w C++ mocno złożona i wymaga znacznie szerszego opracowania, by mogła zostać wykorzystana profesjonalnie

## 5. Bibliografia

1. SPLENDOR; Biblioteka Główna Politechniki Śląskiej. Strona Główna Wyszukiwarki Bazy Dorobek. 29 05 2021. [Zacytowano: 30 06 2021.] <https://www.bg.polsl.pl/expertus/new/bib/>.
2. ORCID. Dokumentacja Interfejsu Sieciowego ORCID. [Zacytowano: 30 06 2021.] <https://pub.orcid.org/v3.0/>.
3. © Elsevier B.V. . Dokumentacja Interfejsu Sieciowego SCOPUS. 2021. [Zacytowano: 30 06 2021.] <https://dev.elsevier.com/search.html>.
4. Clarivate. Dokumentacja Interfejsu Sieciowego Web Of Science. 2021. [Zacytowano: 02 06 2021.] <https://developer.clarivate.com/apis/wos>.
5. Microsoft. Specyfikacja techniczna formatu XLSX. [Zacytowano: 30 06 2021.] <https://support.microsoft.com/pl-pl/office/specyfikacje-i-ograniczenia-programu-excel-1672b34d-7043-467e-8e27-269d656771c3>.
6. Specyfikacja plików CSV. [Zacytowano: 30 06 2021.] [https://en.wikipedia.org/wiki/Comma-separated\\_values](https://en.wikipedia.org/wiki/Comma-separated_values).
7. Bertocci Vittorio. Bezpieczeństwo protokołu OAuth 2.0. 08 01 2019. [Zacytowano: 30 06 2021.] <https://auth0.com/blog/oauth2-implicit-grant-and-spa/>.
8. SMARTBEAR. Strona domowa Swagger. [Zacytowano: 30 06 2021.] <https://swagger.io/>.
9. Müller Philip. Strona dystrybucji Manjaro. 2011. [Zacytowano: 30 06 2021.] <https://manjaro.org/>.
10. Vinet Judd, Griffin Aaron i Polyák. Levente. Strona dystrybucji Archlinux. 2002. [Zacytowano: 30 06 2021.] <https://archlinux.org/>.
11. Dokumentacja języka C++. *cppreference*. 11 08 2020. [Zacytowano: 30 06 2021.] <https://en.cppreference.com/w/>.
12. Specyfikacja notacji węzowej. 19 05 2021. [Zacytowano: 30 06 2021.] [https://en.wikipedia.org/wiki/Snake\\_case](https://en.wikipedia.org/wiki/Snake_case).
13. Specyfikacja notacji wielbłądziej. 24 05 2021. [Zacytowano: 30 06 2021.] [https://en.wikipedia.org/wiki/Camel\\_case](https://en.wikipedia.org/wiki/Camel_case).
14. The Clang Team. Dokumentacja programu clang-tidy. 2021. [Zacytowano: 30 06 2021.] <https://clang.llvm.org/extra/clang-tidy/>.
15. Strona główna programu git. 2021. [Zacytowano: 30 06 2021.] <https://git-scm.com/>.
16. Heesch Dimitri van. Strona główna narzędzia Doxygen. 2021. [Zacytowano: 30 06 2021.] <https://www.doxygen.nl/index.html>.
17. Codacy. Strona projektu codacy. 2021. [Zacytowano: 02 06 2021.] <https://www.codacy.com>.
18. Kitware. Strona projektu cmake. 2021. [Zacytowano: 02 06 2021.] <https://cmake.org/>.

19. Microsoft. Strona główna projektu Visual Studio Code. 2021. [Zacytowano: 30 06 2021.] <https://github.com/microsoft/vscode>.
20. Strona projektu Boost. 11 12 2020. [Zacytowano: 30 06 2021.] <https://www.boost.org/>.
21. Repozytorium projektu boost-ext/µt. 13 05 2021. [Zacytowano: 30 06 2021.] <https://github.com/boost-ext/ut>.
22. an-tao. Repozytorium projektu drogoncpp. 10 04 2021. [Zacytowano: 31 06 2021.] <https://github.com/an-tao/drogon>.
23. Gauniyal Abhinav. Repozytorium projektu rang. 24 11 2018. [Zacytowano: 31 06 2021.] <https://github.com/agauniyal/rang>.
24. Chambe-Eng Eirik i Nord Haavard. Strona projektu Qt. Qt, 2021. [Zacytowano: 02 06 2021.] <https://www.qt.io/>.
25. Community Research and Development Information Service. Strona słownika internetowego Glosbe. *Witryna zamieszczonego przykładu, tłumaczącego słowo framework*. [Zacytowano: 31 05 2021.] <https://app.glosbe.com/tmem/show?id=-2423176738232212855>.
26. Zhang Debao i dand-oss. Repozytorium projektu QtXlsxWriter. 09 12 2020. [Zacytowano: 31 06 2021.] <https://github.com/dand-oss/QtXlsxWriter>.
27. Tryggeseth Eirik. *Report from an Experiment: Impact of Documentation on Maintenance*. Department of Computer and Information Science, Norwegian University of Science and Technology. Trondheim : Empirical Software Engineering, Kluwer Academic Publishers, 1997. str. 5, Raport z Eksperymentu.
28. *Does the Documentation of Design Pattern Instances Impact on Source Code Comprehension? Results from Two Controlled Experiments*. Gravino Carmine i inni. Limerick, Ireland : IEEE, 2011. 2011 18th Working Conference on Reverse Engineering. strony 67-76. DOI: 10.1109/WCRE.2011.18.
29. Sutter Herb. Herb's last appearance on C9. [os. udzielił wyw.] Charles Channel 9. *Going Deep*. 07 06 2011. <https://channel9.msdn.com/Shows/Going+Deep/Conversation-with-Herb-Sutter-Perspectives-on-Modern-C0x11>.
30. Mochocki Krzysztof. Zastosowanie koncepcji w Szablonym Meta-Programowaniu. *Concepts*. 14 05 2021. [Zacytowano: 1 06 2021.] [https://en.wikipedia.org/wiki/Template\\_metaprogramming#Concepts](https://en.wikipedia.org/wiki/Template_metaprogramming#Concepts).
31. Microsoft. Guidelines for Keyboard User Interface Design. 04 2002. [Zacytowano: 02 06 2021.] <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/dnacc/guidelines-for-keyboard-user-interface-design>.
32. Tablica kodów ASCII. [Zacytowano: 02 06 2021.] <https://ascii.cl/>.
33. Wikipedia. *Strona litery Ó*. [Zacytowano: 02 06 2021.] <https://en.wikipedia.org/wiki/%C3%93>.
34. Hacksplaining. Hacksplaining. *Protecting Against SQL Injection*. Hacksplaining. [Zacytowano: 02 06 2021.] <https://www.hacksplaining.com/prevention/sql-injection>.

35. —. Hacksplaining. *Protecting Your Users Against Cross-site Scripting*. Hacksplaining. [Zacytowano: 02 06 2021.] <https://www.hacksplaining.com/prevention/xss-stored>.
36. Torvalds Linus. Komentarz Linusa Torvaldsa na temat formatu XML. . [Komentarz]. 07 03 2014. [https://en.wikiquote.org/wiki/Linus\\_Torvalds#2014](https://en.wikiquote.org/wiki/Linus_Torvalds#2014).
37. Wikipedia. *Strona wzorca projektowego Wizytator*. [Zacytowano: 03 06 2021.] <https://pl.wikipedia.org/wiki/Odwiedzaj%C4%85cy>.
38. Wikipedia. *Strona słowa iterator*. [Zacytowano: 03 06 2021.] <https://pl.wikipedia.org/wiki/Iterator>.
39. computerworld. computerworld. *Strona tłumaczenia słowa spin lock*. [Zacytowano: 03 06 2021.] <https://www.computerworld.pl/slownik/termin/47256/spin-lock.html>.
40. Demin Alexander. Blok osobisty Alexandra Demina. *Porównanie różnych mechanizmów synchronizacji*. 05 05 2012. [Zacytowano: 03 06 2021.] <https://demin.ws/blog/english/2012/05/05/atomic-spinlock-mutex/>.
41. Mochocki Krzysztof. GitHub. *Strona zmiany w repozytorium projektu antybiurokrata*. 10 05 2021. [Zacytowano: 05 06 2021.] <https://github.com/raidgar98/antybiurokrata/commit/d517d11e1ddb33004019cae1336f590a6d9dda90>.
42. Wikipedia. *Strona wzorca projektowego CRTP*. 23 04 2021. [Zacytowano: 06 06 2021.] [https://en.wikipedia.org/wiki/Curiously\\_recurring\\_template\\_pattern](https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern).
43. Herstine Michael. GitHub. *Repozytorium projektu damerau-levenshtein*. 20 03 2021. [Zacytowano: 05 06 2021.] <https://github.com/sp1ff/damerau-levenshtein>.
44. Wikipedia. *Strona hasła String metric*. 22 02 2021. [Zacytowano: 05 06 2021.] [https://en.wikipedia.org/wiki/String\\_metric](https://en.wikipedia.org/wiki/String_metric).
45. Wikipedia. *Strona algorytmu Damerau–Levenshtein*. 06 05 2021. [Zacytowano: 05 06 2021.] [https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein\\_distance](https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance).
46. Wikipedia. *Strona algorytmu Levenshtein*. 09 04 2021. [Zacytowano: 05 06 2021.] [https://en.wikipedia.org/wiki/Levenshtein\\_distance](https://en.wikipedia.org/wiki/Levenshtein_distance).

## 6. Spis ilustracji

Rys. 1. Zapytanie na jednym polu - autorze, wygenerowane przez serwis (skala szarości).....	9
Rys. 2. Podgląd specyfikacji technicznej dostarczonej przez serwis ORCID (skala szarości) .....	10
Rys. 3. Podgląd specyfikacji technicznej dostarczonej przez serwis SCOPUS (skala szarości) .....	11
Rys. 4. Fragment kodu z komentarzami do wygenerowania dokumentacji.....	14
Rys. 5. Fragment wygenerowanej strony internetowej .....	14
Rys. 6. Raport statycznej analizy kodu (skala szarości) .....	15
Rys. 7. Porównanie wyglądu w zależności od wybranego motywu systemu.....	21
Rys. 8. Domyślny widok panelu wyszukiwania .....	23
Rys. 9. Alternatywny widok panelu wyszukiwania .....	23
Rys. 10. Panel współpracowników z przykładowymi danymi (skala szarości) .....	24
Rys. 11. Okienko dialogowe, ze szczegółami przeszukiwania (rozjaśnione) .....	26
Rys. 12. Panel publikacji z przykładowymi danymi .....	26
Rys. 13. Okno dialogowe w momencie detekcji błędu (rozjaśnione) .....	27
Rys. 14. Interfejs klasy demangler .....	29
Rys. 15. Konstruktor struktury bgpolisl_repr_t .....	31
Rys. 16. Przykładowy fragment wyluskania wartości z danych w formacie JSON .....	34
Rys. 17. Obiektowa reprezentacja publikacji .....	35
Rys. 18. Obiektowa reprezentacja osoby autora .....	35
Rys. 19. Fragment klasy - obiektowej postaci zweryfikowanego i jednolitego tekstu .....	36
Rys. 20. Deklaracja typu polish_name_t.....	37
Rys. 21. Interfejs klasy konwertującej dane na obiektową postać osoby (autora) .....	38
Rys. 22. Metoda wyluskująca kolejnych autorów z zadanej publikacji .....	40
Rys. 23. Deklaracje struktur opisujące zależności pomiędzy publikacjami .....	42
Rys. 24. Pętla tworząca zależności, pomiędzy publikacjami .....	43
Rys. 25. Kolejność startu nowych wątków. ....	44
Rys. 26. Deklaracja uniwersalnego funktora .....	45
Rys. 27. Konstruktor klasy głównego okna .....	46
Rys. 28. Początek deklaracji struktury ser .....	48
Rys. 29. Przykład użycia struktur ser oraz cser .....	49
Rys. 30. Wyjście z powyższego przykładu (Rys. 29) .....	49
Rys. 31. Wyjście z poniższego przykładu (Rys. 32) .....	50
Rys. 32. Przykład nadpisania domyślnego wywołania.....	50
Rys. 33. Przykład dostępu do głębiej umieszczonych pól w strukturach .....	53
Rys. 34. Przykład użycia .....	54
Rys. 35. Uruchomienie oraz wyniki wykonanych testów.....	55
Rys. 36. Przykładowy negatywny test, sprawdzający pojawienie się odpowiedniego błędu .....	55
Rys. 37. Pobranie podmodułu przez system kontroli budowy .....	57
Rys. 38. Zatrzymanie programu cmake, po wykryciu budowy w źródle .....	57
Rys. 39. Fragment spisu wszystkich udokumentowanych klas w projekcie .....	59