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 NAPISANY

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

Kierujący pracą:

Dr inż. Adrian SMAGÓR

Recenzent:

dr hab. inż. Roman PRZYŁUCKI, prof. PŚ

Katowice, czerwiec 2021 r.

1. Spis treści

2. Wstęp

2.1. Inspiracja

Inspiracją dla niniejszego projektu inżynierskiego było dostrzeżenie problemu dotykającego 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 największą popularnością. Dodatkowym kryterium jest możliwość dodania lub edycji danych w przeszukiwanym przez aplikację portalu. Przy założonych kryteriach wyłaniają się 4 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 w jego obsłudze, 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 zapewniać łatwą rozszerzalność o nowe strony.

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: *DOROBEK*, *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 *DOROBEK*. 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 *DOROBEK* nie zapewnia.

Naturalnym podejściem osoby niepotrafiącej programować, byłoby ręczne kopiowanie tytułów i wklejane 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*, znacznie ułatwiające wyszukiwanie, a dane zarówno w nim, jak i na stronie są nierzadko zapisane w nieprzystępnej formie.

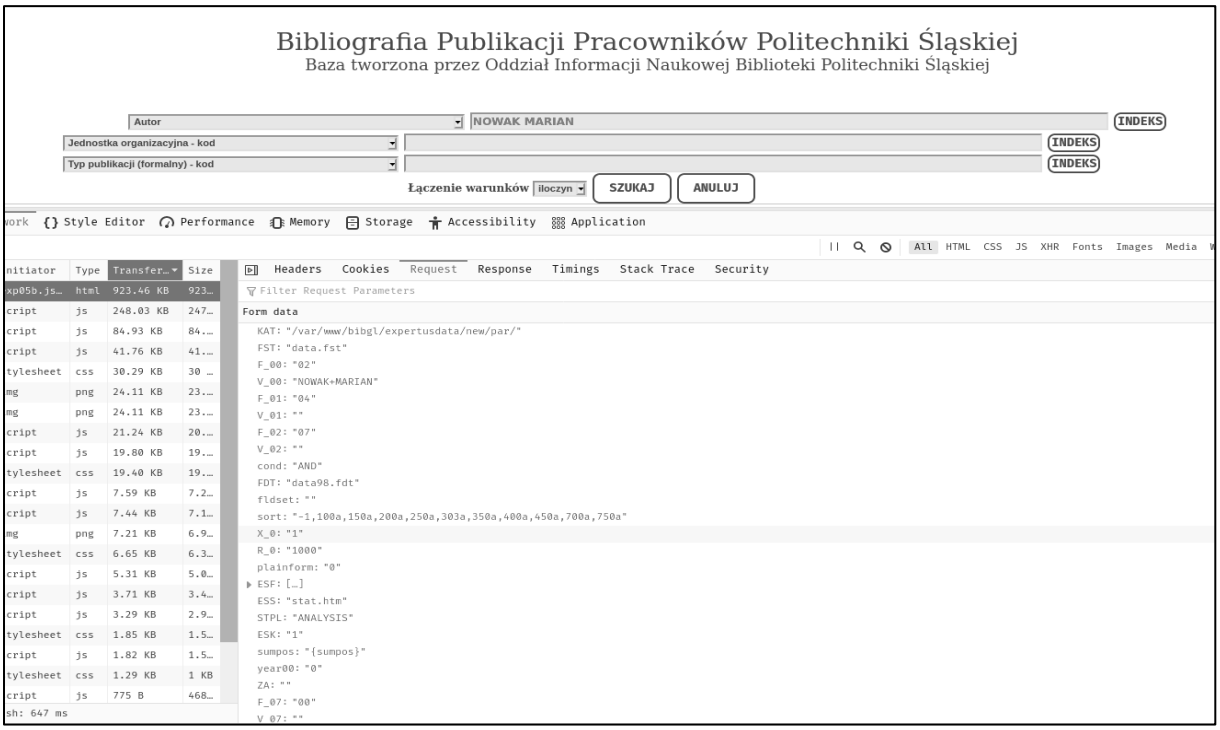
Często widywaną praktyką jest wpisywanie tytułu w języku polskim i nowożytnym po przecinku, średniku lub kropce, co generuje kolejny problem. Problem powodujący niejednoznaczność w dalszych poszukiwaniach, ponieważ pojawiają się dwie różne wartości dla jednego nie tablicowego pola. Strona, oprócz pliku 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 *DOROBEK* 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 wyłonionego zapytania, poprzez wyszukanie największego typu metody *POST* (Rysunek 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 (Rysunek 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



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

trzecich, co wymusza na osobie zainteresowanej automatyzacją procesu weryfikacji spójności danych w innych serwisach na skorzystanie z przetwarzanie danych uzyskanych ze źródła przeznaczonego dla użytkownika przeglądarki internetowej.

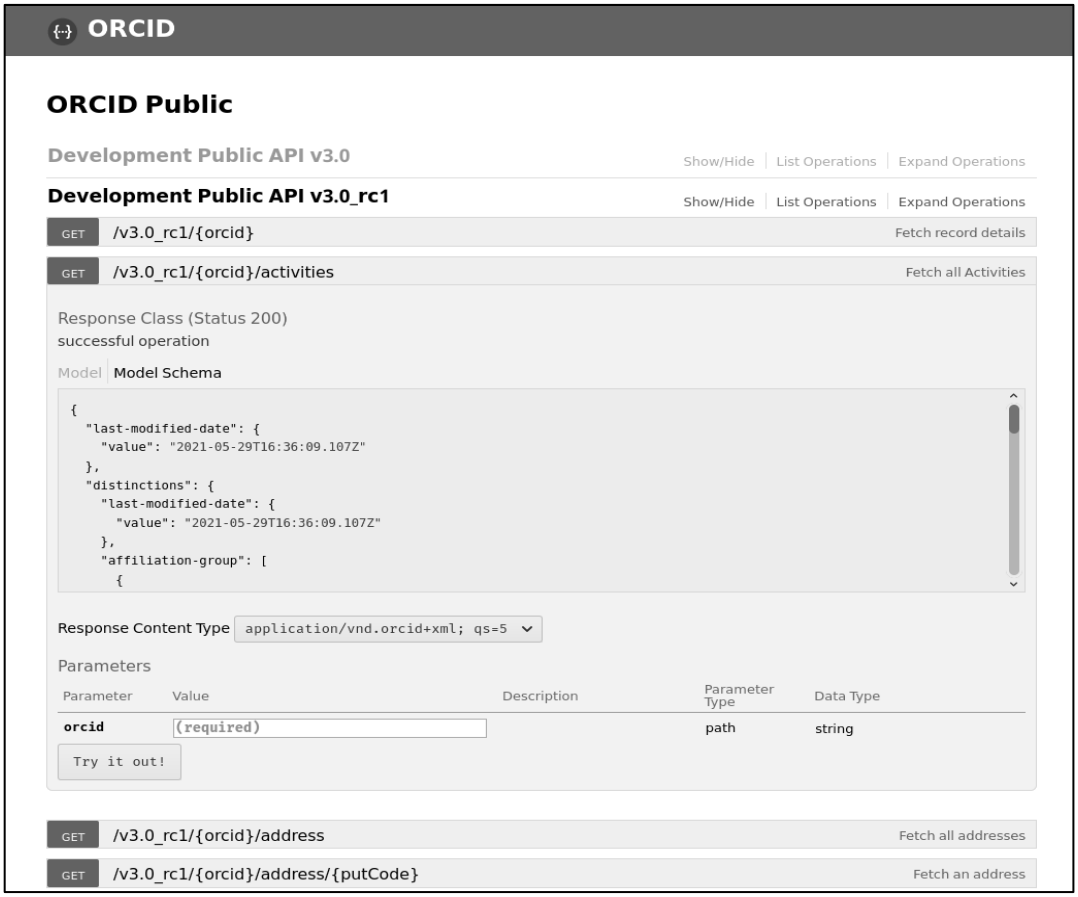
Kolejne serwisy, między innymi *ORCID*, oraz *SCOPUS* zapewniają bardzo obszerne dokumentacje, z przykładami oraz dokładnymi objaśnieniami, w przeciwieństwie do bazy *DOROBEK* są przyjaźnie nastawione do użytkownika i oferują zwięzły panel użytkownik.

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 narzędzie 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* (6). Skorzystanie z bezpieczniejszego połączenia jest wymagane przez serwis, aby móc przeprowadzić proces logowanie i wykonywać operacje na koncie poza stroną internetową.

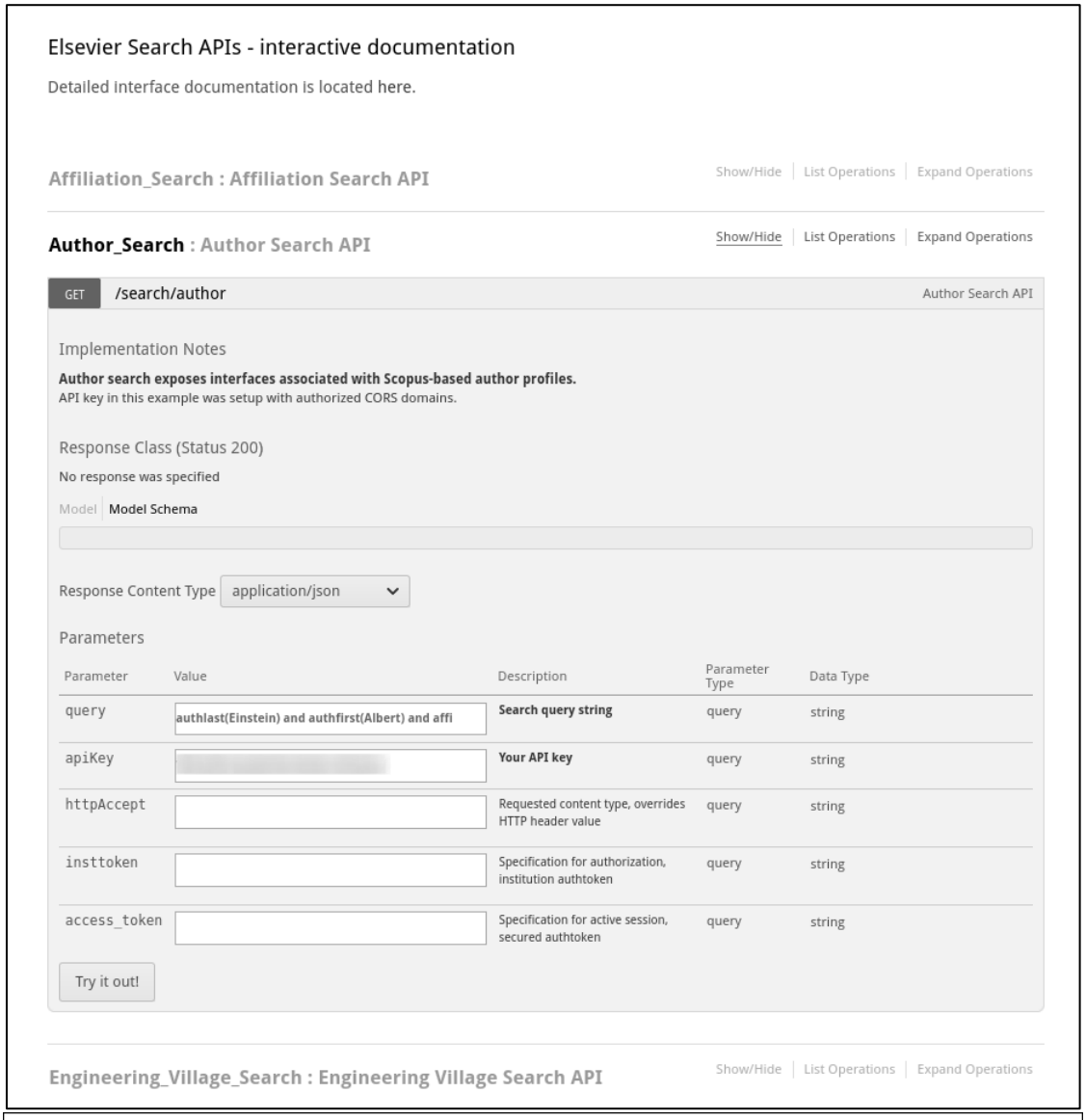


Rysunek 2 Podgląd specyfikacji technicznej dostarczonej przez serwis ORCID (skala szarości)

Dzięki wykorzystaniu otwartego i sprawdzonego standardu tworzenia dokumentacji *Swagger* (7), (Rysunek 2) dokumentacja 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* za wyjątkiem wersjonowania dostarcza te same równie przyjemną interaktywną dokumentację, 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.



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

3.2. Przegląd wybranych technologii

3.2.1. System operacyjny

Manjaro (8) – dystrybucja systemu operacyjnego *GNU/Linux* bazująca na dystrybucji *Arch Linux* (9), 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++ (10) – 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 wężowa (ang. *sneak case*) (11) – zaraz obok wielbłądziej (12) 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 wielbłądzią.

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* (13). 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 (14) – 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 (15) (16) – narzędzie do automatycznego generowania dokumentacji w formie strony internetowej, 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. Dodatkowo, wbudowane programy, do wspomagania programisty, takie jak *IntelliSense* korzystają z tych komentarzy, aby móc wyświetlić pomocne informacje, podczas tworzenia nowego kodu.

```cpp
/**
 * @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 deducted 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)
```

Rysunek 5 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 deducted 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

Rysunek 4 Fragment wygenerowanej strony internetowej

3.2.6. Statyczna analiza kodu

GRADE	FILENAME	ISSUES	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

Rysunek 6 Raport statycznej analizy kodu (skala szarości)

codacy (16) – 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, 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 (17) – 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 (18) – 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 (19) – 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/µt (20) – eksperymentalna część wcześniej wspomnianej biblioteki *boost*, zawierająca pakiet narzędzi do tworzenia testów, wraz z raportowaniem. Biblioteka jest stworzona w zgodnie 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 (21) – 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 serwerowych. Narzędzie nazywa się *drogon-cli*. Okazało się ono bardzo pomocne w momencie tworzenia konfiguracji *cmake*. Był to istotny problem z uwagi na brak wpisów w dokumentacji biblioteki, jak należy podejść do tego zagadnienia. Rozwiązaniem okazało się wygenerowanie szablonowego projektu, a następnie użycie gotowych fragmentów plików wsadowych programu *cmake*.

3.2.9.4. Wspierająca proces logowania

rang (22) – otwarta, nagłówkowa biblioteka oferująca międzyplatformowe, strumieniowe wsparcie dla kolorów w konsoli. Jest to dość niedoceniony aspekt logowania, jednakże zważają 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 (23) – rozbudowana, platforma programistyczna (ang. *framework* (24)) 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 (25) – 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. Założeńia 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 (26), 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 (27), 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 (28), na prośbę o skomentowanie wyjścia, cztero-stronnicowego 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 (29), 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 podzielają 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.

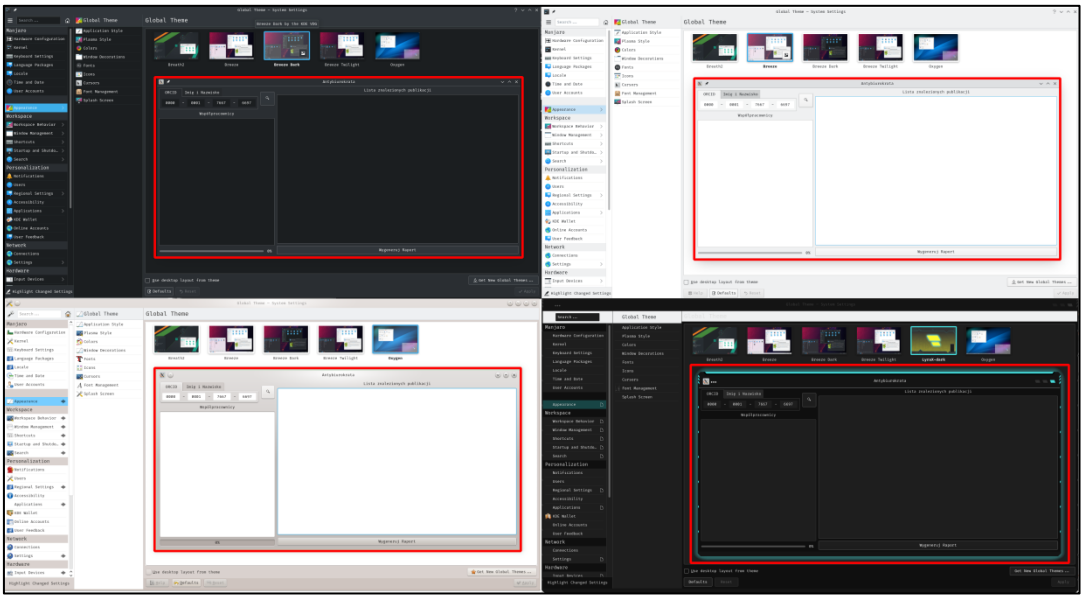
3.3.1.4. Przystępne wyjście

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. Realizacja

3.4.1. 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 (Rysunek 7).



Rysunek 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 (30). 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 funkcją i jest naturalnym przedłużeniem poprzedniej.

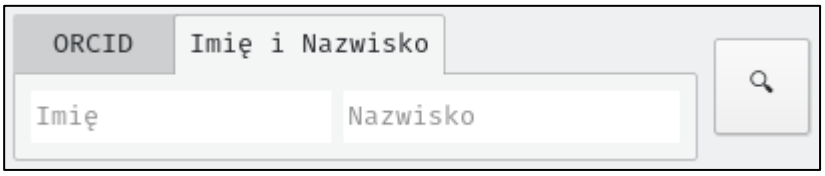
3.4.1.1. Panel wyszukiwania

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 (Rysunek 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 użytkowniku 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.

Rysunek 8 Domyślny widok panelu wyszukiwania



Rysunek 9 Alternatywny widok panelu wyszukiwania

Drugi widok 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: Wierzchosława.

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 dorobek, który to podane imię i nazwisko tłumaczy na numer *ORCID*. Po jego wyłuskaniu kolejne wątki rozpoczynają pracę. W przypadku podania identyfikatora, taki problem nie występuję.

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 motywacje. 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

Rysunek 10 Panel współpracowników z przykładowymi danymi (skala szarości)

Imiona i nazwiska wraz identyfikatorami *ORCID*, są wyświetlane w trakcie procesu wyszukiwania, jako efekt uboczny procesu przetwarzania danych z bazy *DOROBEK*. 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 objawia się w dwojaki sposób. Pierwszym z nich jest pasek postępu, który wskazuję 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 aktualizacje paska postępu w przypadku używania go przez część raportową i jest to spowodowane bardzo szybkim procesem generowania raportu, nawet na sporej ilości danych.

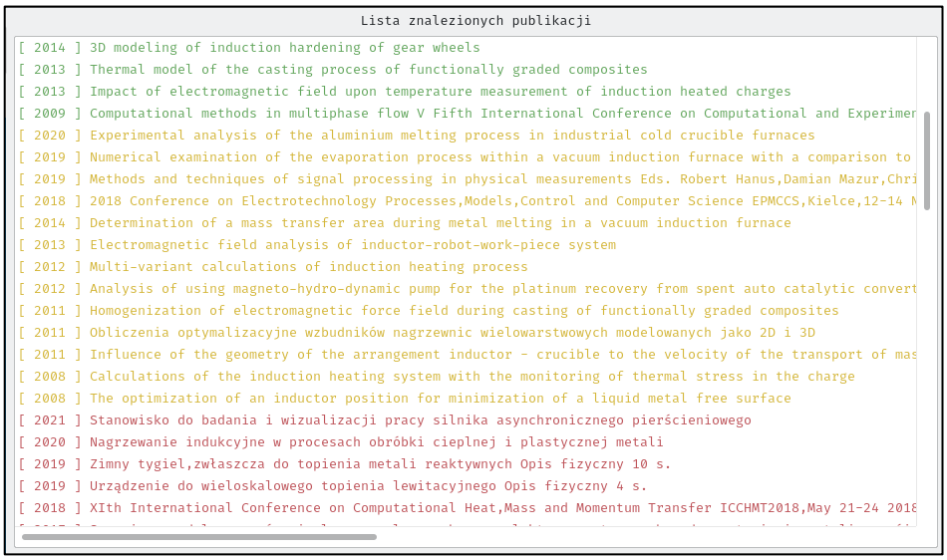
Pozycje na liście oznaczonej, jako „Współpracownicy" są posortowane według ilości wspólnych publikacji z wyszukiwanym podmiotem, znajdującym się zawsze na górze listy. Wybieranie dowolnej pozycji spowoduje wyświetlenie, na prawym panelu (opisanym w kolejnym podrozdziale) listy wspólnych publikacji. Tego 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, żeby 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.

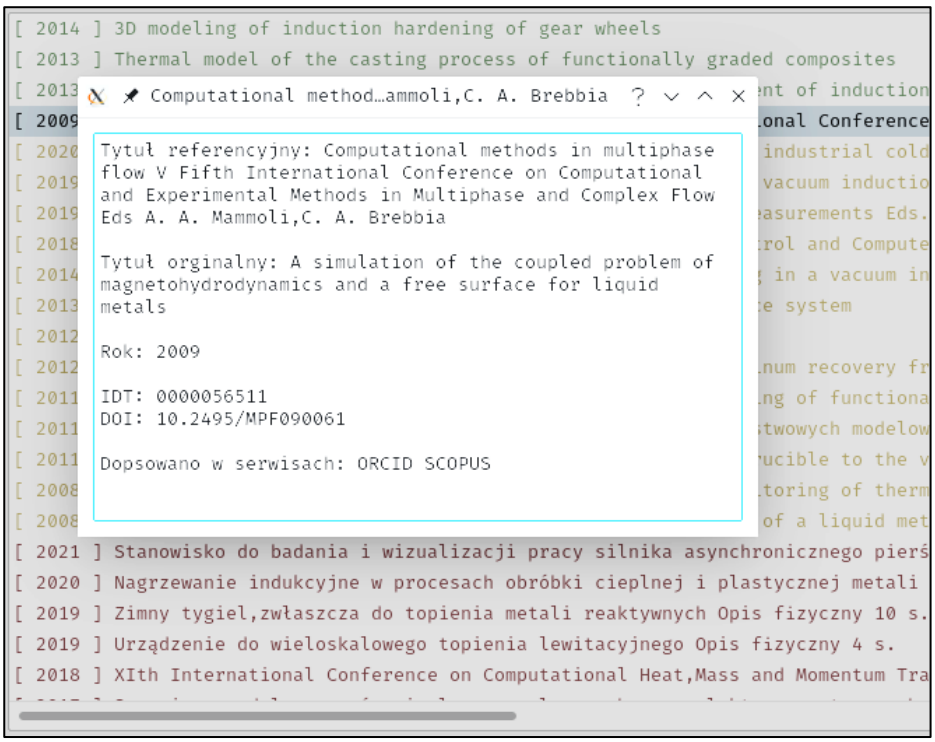
3.4.1.3. Lista publikacji

Największym panelem aplikacji, 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 tą 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, lub może wygenerować raport.



Rysunek 11 Panel publikacji z przykładowymi danymi



Rysunek 12 Okienko dialogowe, ze szczegółami przeszukiwania (rozjaśnione)

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*

3.5. Pobór danych

3.5.1. Baza *DOROBEK*

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

3.5.1.1. Przygotowanie zapytania

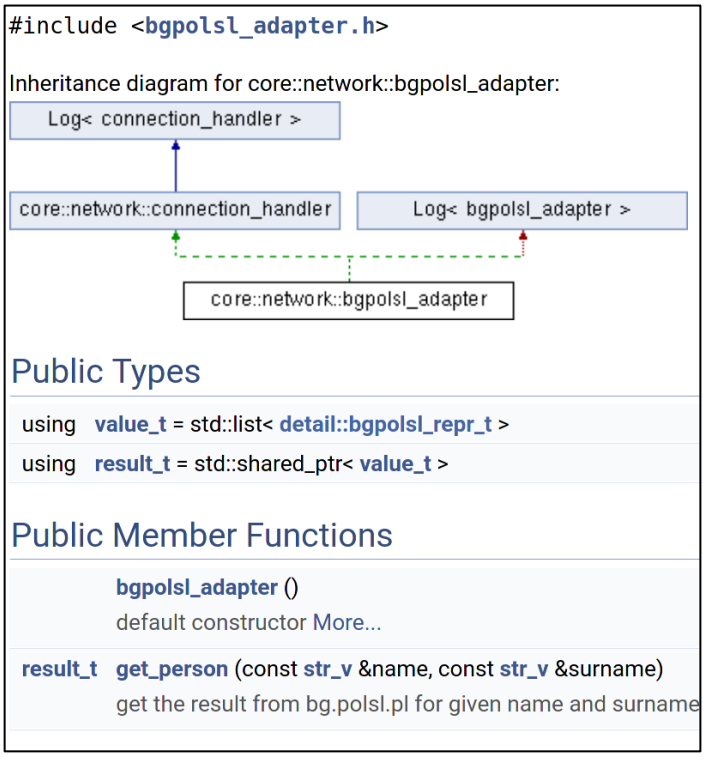
Pierwszym krokiem umożliwiającym stworzenie zapytania jest zapoznanie się z dostępną dokumentacją, bądź jak w tym wypadku analiza ruchu, w kontekście wprowadzonych w formularzu danych (Rysunek 1). Konstrukcja aplikacji daje dwa możliwe kryteria wyszukiwania. Pierwszym jest identyfikator *ORCID*, drugim zaś, para ciągów znakowych imię i nazwisko. Przeglądając dostępne pola, umożliwiające filtrowanie na stronie bazy *DOROBEK*, 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 *DOROBEK*.

Analiza ruchu sieciowego strony, wskazuje, że polem w żądaniu, pobierającym rekordy, odpowiedzialnym za przechowywanie imienia i nazwiska jest to, nazwane *V_00*. Dodatkowo zostaje uwidoczniona 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* (31), 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 (31). Dodatkowo podmiana znaków specjalnych zabezpiecza system, przed

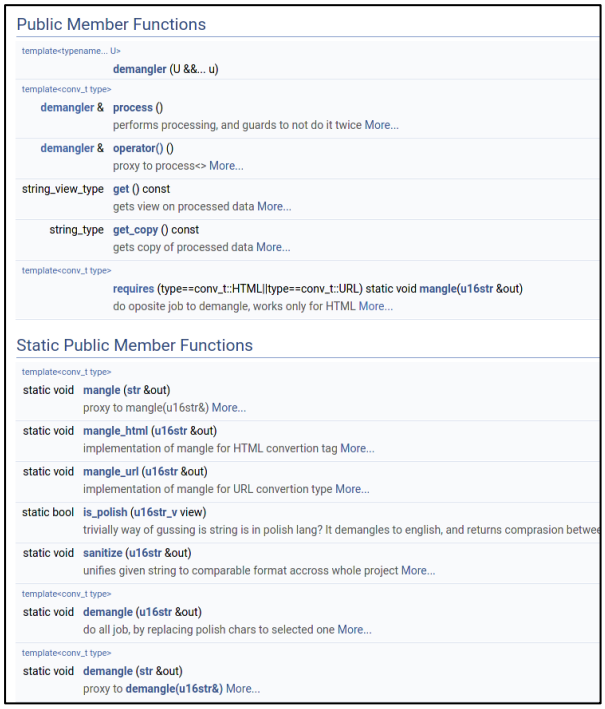
wstrzykiwaniem fragmentów kodu języka SQL (ang. *SQL Injection*) (32), oraz wstrzykiwaniem skryptów klienckich (ang. *Cross Site Scripting*) (33), które mogą narazić serwer, lub klientów na wykonanie niepożądanego kodu.

Przed przejściem do kolejnego etapu, jest sprawdzenie, jak serwer będzie reagował, gdy zapytanie nie nadchodzi za pośrednictwem aplikacji klienckiej, jaką jest aplikacja webowa bazy *DOROBEK*. 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.

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*. 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.



Rysunek 13 Interfejs klasy komunikującej się z bazą dorobek



Rysunek 14 Interfejs klasy *demangler*

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 w 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, zrozumiałego interfejsu. 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 cytat wybitnego programisty, twórcy jądra systemu *GNU/Linux*, Linusa Torvaldsa (35), jednakże z uwagi na jego wulgarność, nie zostanie tutaj przytoczona.

Konsekwencją nieudanych prób, było stworzenie dedykowanego interpretera kodu strony, bazy *DORORBEK*, opierającego się na słowach kluczowych. Składa się on 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 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* (Rysunek 15). Zawiera ona zbiór instrukcji w postaci tablicy asocjacyjnej. Jej kluczem jest

```
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  }
```

Rysunek 15 Konstruktor struktury *bgpolsl_repr_t*

oczekiwane słowo, natomiast wartość jest reprezentowana przez wskaźnik na zmienną, do której kolejne słowa mają trafić. Taka organizacja ma dwie zalety, oraz jedną poważną wadę.

Pierwsza zaleta dotyczy sporej elastyczności kodu. Jeżeli zajdzie 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*.

Zaprojektowany przepływ danych umożliwia zrównoleglenie procesu przetwarzania, jednakże koszt stworzenia nowego wątku, może przewyższać potencjalne zyski czasowe. Potencjalnym rozwiązaniem tego problemu, mogłoby być przetwarzanie wielowątkowe, gdzie każdy z wątków otrzymuje zakres linijek do przetworzenia. Pytanie, czy synchronizowanie docelowej kolekcji, podczas umieszczania w niej wyników prac wątków, nie pozbawiłoby takiej modernizacji sensu, bez testów, pytanie pozostaje bez odpowiedzi.

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 za wyjątkiem adaptera obsługującego bazę *DOROBEK*, 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 (Rysunek 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         }
140
```

Rysunek 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 */
            You, seconds ago | 1 author (You)
370         struct detail_publication_t : public serial_helper_t
371         {
372             u16ser<&detail_publication_t::_>                    title;
373             u16ser<&detail_publication_t::title>                polish_title;
374             dser<&detail_publication_t::polish_title, uint16_t> year;
375             dser<&detail_publication_t::year, ids_storage_t>    ids;
376
377             /**
378              * @brief compares two me with other
379              *
380              * @return int 0 = equal, 1 = greater, -1 = lesser
381              */
382             int compare(const detail_publication_t&) const;
383 >           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)···
391 >           inline friend bool operator<(const detail_publication_t& me, const detail_publication_t& other)···
395 >           inline friend bool operator>(const detail_publication_t& me, const detail_publication_t& other)···
399         };
400         using publication_t        = cser<&detail_publication_t::ids>;
401         using shared_publication_t = pd::shared_t<publication_t>;
402
```

Rysunek 17 Obiektowa reprezentacja publikacji

```
422
423         /** @brief object representation of person (author) */
            You, seconds ago | 1 author (You)
424         struct detail_person_t : public serial_helper_t
425         {
426                     dser<&detail_person_t::_, polish_name_t>              name;
427                     dser<&detail_person_t::name, polish_name_t>           surname;
428                     dser<&detail_person_t::surname, orcid_t>              orcid;
429               mutable  dser<&detail_person_t::orcid, publications_storage_t> publictions{};
430
431 >           friend inline bool operator==(const detail_person_t& p1, const detail_person_t& p2)···
438 >           friend inline bool operator≠(const detail_person_t& p1, const detail_person_t& p2)···
442 >           friend inline bool operator<(const detail_person_t& p1, const detail_person_t& p2)···
451         };
452         using person_t        = cser<&detail_person_t::publictions>;
453         using shared_person_t = processing_details::shared_t<person_t>;
454
```

Rysunek 18 Obiektowa reprezentacja osoby autora

3.6.1. Reprezentacja obiektowa

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

```
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      */
        You, seconds ago | 1 author (You)
165     template<typename validator = default_validator, typename unifier = default_unifier>
166     struct detail_string_holder_t : public serial_helper_t
167     {
168         ser<&detail_string_holder_t::_, u16str> data;
169         u16str raw;
170
171         using custom_serialize    = pd::string::serial;
172         using custom_deserialize  = pd::string::deserial;
173         using custom_pretty_print = pd::string::pretty_print;
174
```

Rysunek 19 Fragment klasy - obiektowej postaci zweryfikowanego i jednolitego tekstu

Przykładem takiej klasy jest *detail_string_holder_t*, 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, który przyjmuje 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 który 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 odnośnie tego typu to wyłącznie jednoargumentowy konstruktor, przyjmujący referencje na zweryfikowany łańcuch znaków. Klasa ma za zadanie przetworzyć łańcuch w sposób umożliwiający porównanie 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* (Rysunek 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
```

Rysunek 20 Deklaracja typu *polish_name_t*

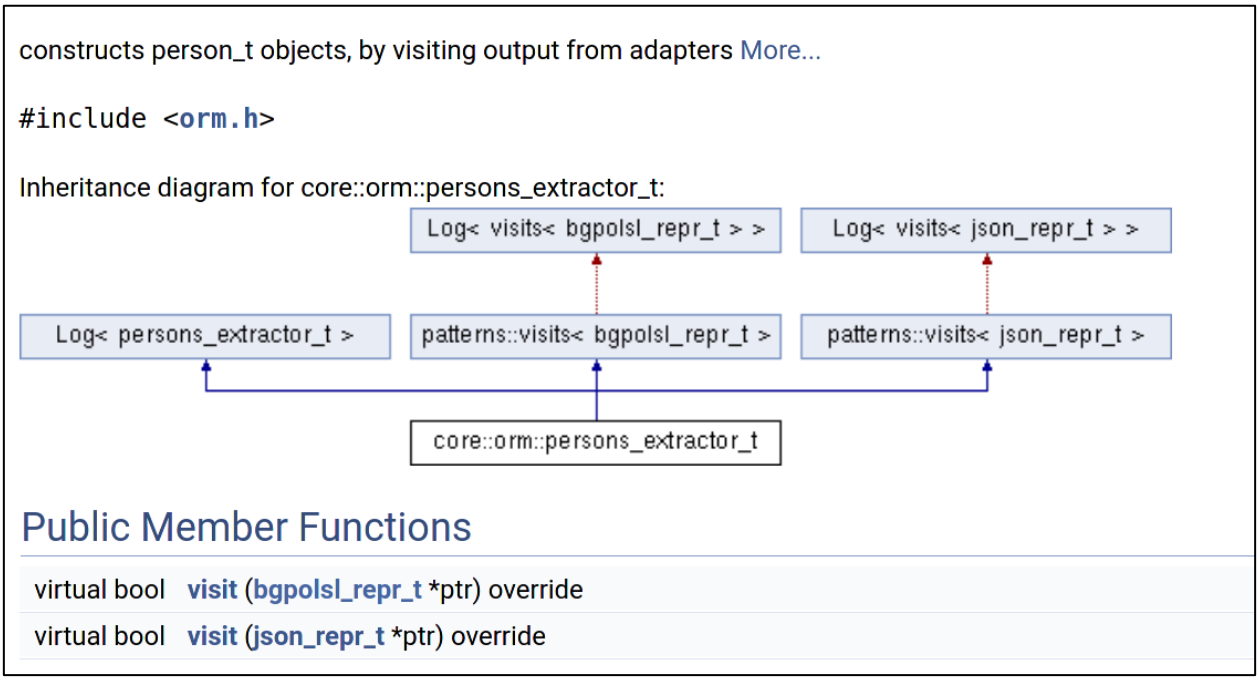
Rozwiązanie oparte o szablony, pozwala na delegację ról do pojedynczych klas lub struktur, oraz znacząco redukuje ilość pisanego kodu. Jeżeli będzie potrzeba zmiany sposobu unifikacji polskich imion, lub dostosowanie poziomu weryfikacji, wystarczy tego dokonać w jednym miejscu. W tym przypadku rolę kontenera 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

Przejściem 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 (Rysunek 21).



Rysunek 21 Interfejs klasy konwertującej dane na obiektową postać osoby (autora)

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, co nie jest pożądane.

3.6.2.1. Przetwarzanie danych z bazy DOROBEK

Rozpoczęcie procesu przetwarzania odbywa się poprzez odwiedzenie, przez klasę *publications_extractor_t*, struktury wyjściowej z adaptera bazy *DOROBEK* – *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 *DOROBEK*, 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 zbiór 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 funkcje biblioteczną *std::stoi*. Brak wsparcia dla dwubajtowych ciągów znakowych, ze strony biblioteki standardowej było zaskoczeniem, oraz wymusiło dodatkową konwersje do standardowego jednobajtowego kontenera tekstu. Jest to dobre miejsce do rozpoczęcia procesu optymalizacji, zastępując koniunkcję tekstowej konwersji oraz wywołania funkcji bibliotecznej, 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 *DOROBEK*. 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 konkatenacji tytułów w języku polskim i nowożytnym przy użyciu kropki, przecinka, średnika lub w najgorszym przypadku spacji, jako separatora, nie jest możliwe poprawne rozdzielenie dwóch tytułów, jeżeli znajdą 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 implementacje opisanego później, w niniejszej pracy, rozwiązania.

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

```
bool persons_extractor_t::visit(bgpolsl_repr_t* ptr)
{
    check_nullptr{ptr};
    shared_person_t result;

    const u16str_v affiliation{ptr->affiliation};   // alias
    auto& person            = result().data();      // alias
    const auto reset_person = [&person] { person.reset(new person_t{}); };
    for(u16str_v part_of_affiliation: string_utils::split_words<u16str_v>{affiliation, u','})
    {
        if(part_of_affiliation.empty()) continue;
        const string_utils::split_words<u16str_v> splitter{part_of_affiliation, u' '};
        auto it = splitter.begin();
        reset_person();

        u16str_v v             = (it == splitter.end() ? u"" : *it);
        const auto safely_move = [&]() -> bool {···

        if(polish_name_t::value_t::validate(v)) (*person)().surname(v);
        else···

        if(!safely_move()) continue;

        if(polish_name_t::value_t::validate(v)) (*person)().name(v);
        else···

        if(!safely_move()) continue;

        if(orcid_t::value_t::is_valid_orcid_string(v))
            (*person)().orcid(orcid_t::value_t::from_string(v));
        else···

        auto pair = this->persons->insert(result);
        if(pair.second)
            log.info() << "successfully added new author: "
                       << patterns::serial::pretty_print{(*(*pair.first)())} << logger::endl;
        if(current_publication())
            (*(*pair.first)())().publictions()()->insert(current_publication().data());
    }

    return true;
}
```

Rysunek 22 Metoda wyłuskująca kolejnych autorów z zadanej publikacji

Klasa wyłuskująca autorów, korzysta z autorskiego iteratora (38) zakresów tekstowych. Motywacją do implementacji takiej konstrukcji, było rozczarowanie nowym modułem 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 zakresu, 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 zrzucie ekranu (Rysunek 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 obecnej 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

przepisanie tytuły oraz przepisane zostają 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 kolejnym rozdziale. 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>;
```

Rysunek 23 Deklaracje struktur opisujące zależności pomiędzy publikacjami

Klasa, która zawiera definicję dopasowania publikacji, oraz generuje obiekty struktur przedstawionych na powyższym rysunku (Rysunek 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).

```
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 allready 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&) { matches.emplace(mt, y); });
72                      break;
73                  }
74              }
75          }
76      }
```

Rysunek 24 Pętla tworząca zależności, pomiędzy publikacjami

Linia 40 zawiera 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 przetwarzanie, 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 instancje 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

Klasą odpowiedzialną za logikę, w szczególności pilnowanie kolejności inicjalizacji ekstraktorów oraz

3.8.1. Zrównoleglenie przetwarzania danych

3.8.2. Generowanie raportu

3.9. Omówienie serializacji

3.9.1. Zasada działania

3.9.2. Problematyka

3.9.3. Wady i zaley

3.10. Narzędzia

3.10.1. Generowanie bibliotek

3.10.1.1. Biblioteki funkcyjne

3.10.1.2. Biblioteki graficzne

3.10.2. Asysta z systemu kontroli budowy

3.10.2.1. Pilnowanie aktualności bibliotek

3.10.2.2. Zapobieganie przypadkowej budowie

3.10.2.3. Generowanie dokumentacji

3.10.2.4. Formatowanie

4. Podsumowanie

4.1. Perspektywy rozwoju

4.2. Wnioski

5. Bibliografia

1. **SPLENDOR; Biblioteka Główna Politechniki Śląskiej.** Strona Główna Wyszukiwarki Bazy Dorobek. [Online] 29 05 2021. [Zacytowano: 30 06 2021.] https://www.bg.polsl.pl/expertus/new/bib/.

2. **ORCID.** Dokumentacja Interfejsu Sieciowego ORCID. [Online] [Zacytowano: 30 06 2021.] https://pub.orcid.org/v3.0/.

3. **© Elsevier B.V. .** Dokumentacja Interfejsu Sieciowego SCOPUS. [Online] 2021. [Zacytowano: 30 06 2021.] https://dev.elsevier.com/search.html.

4. **Clarivate.** Dokumentacja Interfejsu Sieciowego Web Of Science. [Online] 2021. [Zacytowano: 02 06 2021.] https://developer.clarivate.com/apis/wos.

5. **Microsoft.** Specyfikacja techniczna formatu XLSX. [Online] [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. [Online] [Zacytowano: 30 06 2021.] https://en.wikipedia.org/wiki/Comma-separated_values.

7. **Bertocci Vittorio.** Bezpieczeństwo protokołu OAuth 2.0. [Online] 08 01 2019. [Zacytowano: 30 06 2021.] https://auth0.com/blog/oauth2-implicit-grant-and-spa/.

8. **SMARTBEAR.** Strona domowa Swagger. [Online] [Zacytowano: 30 06 2021.] https://swagger.io/.

9. **Müller Philip.** Strona dystrybucji Manjaro. [Online] 2011. [Zacytowano: 30 06 2021.] https://manjaro.org/.

10. **Vinet Judd, Griffin Aaron i Polyák. Levente.** Strona dystrybucji Archlinux. [Online] 2002. [Zacytowano: 30 06 2021.] https://archlinux.org/.

11. Dokumentacja języka C++. *cppreference.* [Online] 11 08 2020. [Zacytowano: 30 06 2021.] https://en.cppreference.com/w/.

12. Specyfikacja notacji wężowej. [Online] 19 05 2021. [Zacytowano: 30 06 2021.] https://en.wikipedia.org/wiki/Snake_case.

13. Specyfikacja notacji wielbłądziej. [Online] 24 05 2021. [Zacytowano: 30 06 2021.] https://en.wikipedia.org/wiki/Camel_case.

14. **The Clang Team.** Dokumentacja programu clang-tidy. [Online] 2021. [Zacytowano: 30 06 2021.] https://clang.llvm.org/extra/clang-tidy/.

15. Strona główna programu git. [Online] 2021. [Zacytowano: 30 06 2021.] https://git-scm.com/.

16. **Heesch Dimitri van.** Strona główna narzędzia Doxygen. [Online] 2021. [Zacytowano: 30 06 2021.] https://www.doxygen.nl/index.html.

17. **Codacy.** Strona projektu codacy. [Online] 2021. [Zacytowano: 02 06 2021.] https://www.codacy.com.

18. **Kitware.** Strona projektu cmake. [Online] 2021. [Zacytowano: 02 06 2021.] https://cmake.org/.

19. **Microsoft.** Strona główna projektu Visual Studio Code. [Online] 2021. [Zacytowano: 30 06 2021.] https://github.com/microsoft/vscode.

20. Strona projektu Boost. [Online] 11 12 2020. [Zacytowano: 30 06 2021.] https://www.boost.org/.

21. Repozytorium projektu boost-ext/µt. [Online] 13 05 2021. [Zacytowano: 30 06 2021.] https://github.com/boost-ext/ut.

22. **an-tao.** Repozytorium projektu drogoncpp. [Online] 10 04 2021. [Zacytowano: 31 06 2021.] https://github.com/an-tao/drogon.

23. **Gauniyal Abhinav.** Repozytorium projektu rang. [Online] 24 11 2018. [Zacytowano: 31 06 2021.] https://github.com/agauniyal/rang.

24. **Chambe-Eng Eirik i Nord Haavard.** Strona projektu Qt. [Online] 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.* [Online] [Zacytowano: 31 05 2021.] https://app.glosbe.com/tmem/show?id=-2423176738232212855.

26. **Zhang Debao i dand-oss.** Repozytorium projektu QtXlsxWriter. [Online] 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 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. udziel. 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 konceptów w Szablonym Meta-Programowaniu. *Concepts.* [Online] 14 05 2021. [Zacytowano: 1 06 2021.] https://en.wikipedia.org/wiki/Template_metaprogramming#Concepts.

31. **Microsoft.** Guidelines for Keyboard User Interface Design. [Online] 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. [Online] [Zacytowano: 02 06 2021.] https://ascii.cl/.

33. Wikipedia. *Strona litery Ó.* [Online] [Zacytowano: 02 06 2021.] https://en.wikipedia.org/wiki/%C3%93.

34. **Hacksplaining.** Hacksplaining. *Protecting Against SQL Injection.* [Online] Hacksplaining. [Zacytowano: 02 06 2021.] https://www.hacksplaining.com/prevention/sql-injection.

35. —. Hacksplaining. *Protecting Your Users Against Cross-site Scripting.* [Online] 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.

37. Wikipedia. *Strona wzorca projektowego Wizytator.* [Online] [Zacytowano: 03 06 2021.] https://pl.wikipedia.org/wiki/Odwiedzaj%C4%85cy.

38. Wikipedia. *Strona słowa iterator.* [Online] [Zacytowano: 03 06 2021.] https://pl.wikipedia.org/wiki/Iterator.

39. **computerworld.** computerworld. *Strona tłumaczenia słowa spin lock.* [Online] [Zacytowano: 03 06 2021.] https://www.computerworld.pl/slownik/termin/47256/spin-lock.html.

40. **Demin Alexander.** demin. *Porównanie różnych mechanizmów synchronizacji.* [Online] 05 05 2012. [Zacytowano: 03 06 2021.] https://demin.ws/blog/english/2012/05/05/atomic-spinlock-mutex/.

6. Spis ilustracji