POLITECHNIKA BIAŁOSTOCKA

Wydział Informatyki

Paweł Żukowski

Multifile Renaming Utility
Program narzędziowy do grupowej zmiany nazw plików na podstawie metadanych w nich zawartch

PRACA INŻYNIERSKA

Promotor
dr inż. Marcin Skoczylas

Białystok 2013

Streszczenie

<!TODO: streszczenie pracy po angielsku!>

Słowa kluczowe: system plików, wtyczki, metadane, wxWidgets, boost, SigC++

Keywords: filesystem, plugins, metadata, wxWidgets, boost, SigC++

Spis treści

Rozdział 1.

Wstęp

Z każdym rokiem ludzie oraz same komputery generują coraz większą ilość informacji. Mimo że duża część z nich jest przechowywana w dobrze strukturyzowanych bazach danych, to ciągle, większość ludzi ma bezpośredni dostęp jedynie to tego co przechowuje w systemie plików własnego komputera.

Od dziesięcioleci dysk twardy pozostaje głównym kontenerem danych dla komputerów na całym świecie. Wiele osób na przestrzeni lat tworzy swoistego rodzaju kolekcje danych — albumy zdjęć, biblioteki muzyczne czy filmowe, a także duże ilości dokumentów na potrzeby działalności gospodarczej czy prywatnej pracy. Niektórzy administratorzy zarządzający serwerami aplikacji zmagają się z problemem wielkiej ilości plików generowanych przez użytkowników.

1.1 Motywacja

Tysiące plików mogą stworzyć gąszcz informacyjny w którym człowiek będzie czuł się zagubiony. Przy coraz większej ilości informacji nie bez znaczenia pozostaje czynnik ludzki którego możliwości percepcji są ograniczone. Istnieje wiele programów ułatwiających katalogowanie danych jednak skierowane są one zwykle na pojedyncze typy plików i wymagają od użytkownika przyzwyczajenia się do ich używania. Z drugiej strony, przeciętny użytkownik jest zwykle przyzwyczajony do standardowego programu

systemów operacyjnych — przeglądarki plików.

Problemem jednak jest fakt że programy rzadko generują przyjazne użytkownikowi nazwy plików, zwykle ograniczając się do prostego prefiksu i grupy kolejnych numerów zapewniających unikalność.
W przypadku obrazów, często bywa iż kolekcja musi być trzymana w wielu katalogach ponieważ nazwy plików się pokrywają.
Rzadko można również znaleźć interesujący utwór w bibliotece muzycznej której pliki posiadają nazwy różniące się jedynie numerem ścieżki.
Wreszcie istnieją też sytuacje gdy wiele różnych plików jest trzymanych w pojedynczym katalogu co skutecznie utrudnia nawigację i znalezienie tego czego użytkownik faktycznie poszukuje.

1.2 Cel i zakres pracy

Niniejsza praca ma na celu stworzenie programu narzędziowego pozwalającego na automatyczne generowanie identyfikatorów (nazw) plików na podstawie (meta)danych w nich zawartych, a także ich stosowanie do zbiorów plików.

Zakres pracy obejmuje:

- Przegląd istniejących rozwiązań - programów i technik wspomagających masową zmianę identyfikatorów plików.
- Porównanie funkcjonalności istniejących narzędzi i ich ograniczeń.
- Projekt oraz implementacja wieloplatformowej architektury modułów.
- Stworzenie parsera wyrażeń zawierających metatagi.
- Projekt graficznego interfejsu użytkownika opartego na bibliotece wxWidgets.
- Implementacja backendu do systemu plików opartego na bibliotece boost::filesystem.

- Implementacja przykładowych modułów metatagów.
- Testy aplikacji.

1.3 Założenia

Gotowa aplikacja powinna być niezależna od systemu operacyjnego w stopniu w jakim pozwalają na to zależności użytych bibliotek. Dzięki modułowej budowie powinna także udostępniać interfejs pozwalający na jej łatwą rozbudowę.

1.4 Plan pracy

W rozdziale 2 opisano teoretyczny schemat przechowywania danych oraz genezę systemów oraz identyfikacji plików. Rozdział 3 zawiera opis środowiska wykorzystanego do stworzenia projektu i implementacji aplikacji. Rozdział 4 zawiera szczegółowy opis architektury aplikacji wraz z rozwiązaniami wykorzystanymi do jej stworzenia. Ostatni rozdział — 5 — zawiera wnioski na temat wykonanego projektu.

Rozdział 2.

Teoria

W niniejszym rozdziale postaram się przybliżyć obraz problemu identyfikatorów (zwanych również nazwami) plików opisując środowisko i w którym występuje.

2.1 Dane w systemie komputerowym

Jednym z podstawowych elementów systemu komputerowego jest jego pamięć. Od początku istnienia komputerów istniała potrzeba składowania danych wymaganych przy praktycznie każdych operacjach wykonywanych przez jednostkę centralną komputera. Jako że pierwsze systemy komputerowe były wykorzystywane do obliczeń typowo matematycznych, algorytmy na nich uruchamiane nie wymagały wielkich kontenerów na dane. W tych czasach wbudowane rejestry oraz ulotna pamięć RAM zaspokajały potrzemy rynku. Jednak wraz z rozwojem sprzętu i algorytmów na nim uruchamianych pojawiła się potrzeba przechowywania coraz to większej ilości danych jak i (dzięki zastosowaniu architektury von Neumanna) samych programów przez coraz dłuższy czas. Pojawiła się idea nieulotnej oraz pojemnej pamięci - dysku twardego.
<! sprawdzić !>

Pojemności pierwszych dysków twardych stanowiły promil dzisiejszych jednostek toteż nie wymagały stosowania systemów plików - były po prostu nieulotnym rozszerzeniem pamięci operacyjnej RAM.

Wraz ze zwiększeniem ich pojemności oraz generalizacją oprogramowania, pojawiła się potrzeba standaryzowania, kategoryzacji przechowywanych na dyskach danych - tak powstały systemy plików.

2.2 Systemy plików i identyfikacja danych

System plików stanowi warstwę abstrakcji między programami, a danymi zapisanymi na nośniku — dysku twardym, karcie pamięci czy też płycie CD. System plików jest metodą zapisu danych — schematem dzięki któremu, programy nie muszą operować na surowych blokach danych lecz mogą korzystać z bardziej wysokopoziomowych deskryptorów plików - węzłów bądź ścieżek dostępu.
Zwykle systemem plików zarządza system operacyjny — to on udostępnia API, a także blokuje lub pozwala na dostęp do danych ze względu na uprawnienia użytkownika, programu lub samego zasobu.

Istnieje wiele typów oraz implementacji systemów plików, które można podzielić na dwie kategorie:

- tradycyjne - znajdujące zastosowanie przy przechowywaniu dowolnych (ogólnych) danych w postaci plików
- specjalne - dostosowane do specyficznych rozwiązań (jak na przykład bazy danych)

Oddzielną kategorię mogą stanowić zdobywające coraz większą popularność wirtualne systemy plików - różnią się one od tradycyjnych i specjalistycznych tym że nie przechowują danych fizycznie na nośniku, a są raczej aplikacjami udostępniającymi (generującymi) struktury danych na żądanie użytkownika/programu. Przykładem takich systemów mogą być: `procfs` - udostępniający dostęp do procesów systemowych i ich atrybutów w systemach rodziny GNU/Linux oraz *BSD, czy też `NFS` (Network File System) — pozwalający na dostęp do systemów plików znajdujących się na innych komputerach w sieci.

2.2.1 Katalogi i ścieżki do plików

Niniejsza praca skupia się na problemie opisywania danych w tradycyjnych systemach plików za pomocą tak zwanych ścieżek do plików.

Tradycyjne systemy plików pozwalają na przechowywanie danych w drzewiastej strukturze danych zwanej drzewem katalogów. W większości implementacji każdy węzeł takiego drzewa może być katalogiem, plikiem lub dowiązaniem do innego węzła. Dodatkowo węzły katalogów jako jedyne mogę posiadać węzły podległe — podkatalogi.
Każdy węzeł prócz węzła-korzenia jest identyfikowany przez unikalny względem węzła-rodzica identyfikator zwany nazwą pliku/katalogu.
Warto zauważyć iż struktura drzewa katalogów nie wymusza sposobu rozkładu danych w systemie plików — tak długo jak identyfikatory pozostają unikalne, pliki przez nie opisywane mogą znajdować się w tym samym katalogu[1].

2.2.2 Różnice w identyfikacji plików wśród różnych systemów operacyjnych

Format ścieżki do pliku narzucany jest niezależnie od zastosowanego systemu plików przez system operacyjny.

Systemy kompatybilne ze standardem `POSIX`, wywodzące się z Unixów takie jak Apple MacOS czy rodzina BSD, a także rodzina GNU/Linux używają drzew katalogów z pojedynczym, nienazwanym korzeniem oznaczanym symbolem prawego ukośnika (slash) — '`/`'.
Symbol prawego ukośnika jest również używany jako separator elementów (poziomów) ścieżki i nie może stanowić elementu identyfikatora węzła w wymienionych środowiskach.
Przykład ścieżki zgodnej ze standardem `POSIX`:

```
/home/idlecode/projects/mru/doc/main.tex
```

[1] W praktyce ilość plików które mogą należeć do jednego węzła zależy od rozmiaru licznika użytego w implementacji.

Systemy operacyjne z rodziny Windows korporacji Microsoft[2] wykorzystują natomiast lewy ukośnik (backslash) — '\' — jako separator komponentów ścieżki oraz uniemożliwiają stosowanie większej ilości symboli w nazwach.
System plików systemu Windows może posiadać kilka korzeni (po jednym dla każdego wolumenu/dysku) oznaczanych pojedynczymi, zwykle dużymi literami alfabetu łacińskiego. Litera dysku wraz z symbolem dwukropka poprzedza właściwą ścieżkę do pliku.
Przykład ścieżki używanej w systemach operacyjnych Windows korporacji Microsoft:

```
C:\Users\idlecode\My Documents\Projects\MRU\doc\main.tex
```

Dodatkowo w przypadku obu[3] wyżej wymienionych schematów, nazwy elementów nie mogą zawierać znaku zerowego (NUL — o kodzie heksadecymalnym `0x00`), który może zostać zinterpretowany jako koniec łańcucha znaków.

Większość implementacji pozwala zawrzeć pełen zakres symboli (znaków) w ścieżce za pomocą kodowań z rodziny `UTF` przy czym pojedynczy identyfikator może mieć maksymalną długość 255 bajtów. Ograniczenie długości całkowitej ścieżki, jeśli istnieje jest wynosi ustawione na poziomie. Warto tu zauważyć iż systemy z rodziny Windows zachowują wielkość liter w identyfikatorach lecz przy interpretacji ścieżek — rozwijaniu ich do odpowiadających węzłów — nie gra ona znaczenia co nie ma miejsca w systemach klasy `POSIX`. Istnieje również możliwość stosowania ukośników prawych do rozdzielania komponentów ścieżki tak jak to ma miejsce w systemach `POSIX`-owych.

Istnieje jeszcze kilka schematów zapisu ścieżek, które nie zostały przybliżone ze względu na zakres niniejszej pracy.

[2]Istnieje wiele więcej systemów operacyjnych używających podobnego schematu
[3]System MacOS nie posiada tego ograniczenia

2.3 Metadane

Metadane z definicji są danymi opisującymi inne dane. Metadane stosowane są w przypadkach gdy nie istnieje fizyczna możliwość dołączenia lub dodatkowe informacje są zbyt luźno powiązane z opisywanymi danymi. Przykładem metadanych mogą być karty biblioteczne — informują one o statusie i historii np. książki nie będąc jej integralną częścią.

W systemach plików, metadane dostarczają informacji o plikach zapisanych w drzewie katalogów. Przykładem komputerowych metadanych może być wspominana wcześniej nazwa czy ścieżka do pliku, która nie jest jego integralną częścią — może zostać zmieniona bez naruszania struktury przechowywanego dokumentu. Dodatkowo systemy plików często dostarczają ogólnych atrybutów — meta-informacji możliwych do uzyskania z dowolnego typu pliku takich jak jego rozmiar, czas utworzenia/ostatniej modyfikacji czy prawa dostępu. Ciekawym przykładem metadanych są rozszerzenia nazw plików — sufiksy rozpoczynające się od ostatniego znaku kropki w nazwie. Rozszerzenia odgrywały ważną rolę w systemach operacyjnych korporacji Microsoft gdzie stanowiły integralną część nazwy i pozwalały systemowi skojarzyć typ pliku z programem go obsługującym. W systemach `POSIX`-owych informacja o typie pliku jest zwykle przekazywana wraz z kontekstem uruchomienia aplikacji operującej na pliku (za pomocą linii komend) lub pomijana całkowicie — wiele aplikacji takich systemów operuje na plikach jako ciągu bajtów i nie rozróżnia ich typów.

Niektóre formaty plików (szczególnie multimedialnych) pozwalają na integracje metadanych z samym plikiem. Jako że pliki (szczególnie binarne) mogą stosować dowolną strukturę zapisu, nie istnieje ogólny algorytm wyciągnięcia zawartych w ten sposób informacji. Istnieje natomiast wiele bibliotek umożliwiających odczyt informacji z określonego typu pliku.

Do metadanych można zaliczyć także sumy kontrolne. Są to wartości liczone na podstawie zawartości pliku używane w celu testów integralności oraz identyczności.

Rozdział 3.

Środowisko pracy

Rozdział ten zawiera opis środowiska które zostało użyte do stworzenia implementacji, a także architektury samej aplikacji.

3.1 Język C++

Do implementacji aplikacji `MRU` został użyty język C++ w standardzie z roku 2003 (ISO/IEC 14882:2003). Język C++ jest dojrzałym, wieloplatformowym językiem programowania średniego poziomu, używanym od wielu lat przez programistów na całym świecie do tworzenia aplikacji, sterowników czy nawet systemów operacyjnych. Dzięki kompatybilności z C[1] pozwala na wykorzystanie wielu istniejących bibliotek napisanych zarówno w C jak i C++.

`LLVM` — Low Level Virtual Machine — jest modułową architekturą do budowy kompilatorów. Pozwala ona na oddzielenie parserów różnych języków programowania (frontendów) od modułu optymalizacji (wspólnych dla wszystkich języków kompilowalnych) i emiterów kodu bajtowego (backendów) dla różnych platform.

[1] C++ nie jest całkowicie kompatybilny z C, jednak różnice w obu tych językach są na tyle małe że zwykle nie wpływają negatywnie na kompatybilność (szczególnie na poziomie ABI).

Clang jest frontendem języków C i C++ dla architektury `LLVM`. Projekt jest otwarty (wydawany na licencji BSD) i zdobywa coraz większą popularność[2] dorównując i przewyższając w niektórych testach GCC.

3.2 System operacyjny FreeBSD

System FreeBSD jest systemem operacyjnym z rodziny BSD wywodzącej się z kolei z rodziny `UNIX`-ów. Podobnie do dystrybucji GNU/Linux, sam w sobie wraz w wieloma, otwartymi bibliotekami tworzonymi przez społeczność stanowi środowisko przyjazne programistom.

3.3 Mercurial

Do zarządzania plikami źródłowymi oraz kopią zapasową został wykorzystany rozproszony system kontroli wersji Mercurial. Wraz z serwisem bitbucket.org pozwala on na synchronizację kodów źródłowych między wieloma maszynami, ułatwiając tym samym pracę nad pojedynczym projektem wielu programistów.

W odróżnieniu od scentralizowanych systemów kontroli wersji takich jak SVN, Mercurial, podobnie jak Git nie wymaga pojedynczego serwera, ani serwera w ogóle. Pełne repozytorium jest trzymane na każdej maszynie z której korzysta programista, a praca różnych programistów (zmiany w kodzie) może być synchronizowana między nimi samymi.

Rysunek 3.1: Logo systemu Mercurial

[2]Od listopada 2012 Clang wraz z LLVM stał się domyślnym kompilatorem dla systemu FreeBSD

3.4 CMake

Aby projekt był jak najbardziej przenośny i niezależny od platformy, ważne jest aby jego proces budowania był również taki był. W celu zapewnienia łatwego wsparcia dla budowania projektu na wielu platformach i wielu łańcuchach narzędziowych, MRU stosuje CMake — narzędzie do zarządzania procesem kompilacji i zależnościami.

CMake pozwala programiście określić z jakich elementów składa się program i jakich zewnętrznych zasobów (bibliotek) wymaga. Narzędzie następnie interpretuje skryptowy plik konfiguracyjny i tworzy natywne dla danej platformy pliki projektowe zawierające odpowiednią do zbudowania projektu konfigurację.

3.5 Vim

Edytor `Vim` jest rozszerzoną wersją klasycznego `vi`, który jest standardowym oprogramowaniem w przypadku dystrybucji zarówno GNU/Linux jak i systemów z rodziny BSD. Vim jest platformą dla wielu pluginów które tworzą jego faktyczną funkcjonalność. Vim sam w sobie wspiera pracę z wieloma dokumentami, koloruję składnie plików źródłowych i posiada wiele komend ułatwiających produkcję kodu. Dzięki wtyczkom istnieje możliwość rozszerzenia go o zaawansowane kompletowanie składni czy szybkie wstawki kodu (ang. snippety).

Rysunek 3.2: Logo edytora Vim

Rozdział 4.

Implementacja

4.1 Specyfikacja wymagań

4.1.1 Wymagania funkcjonalne

Użytkownikiem aplikacji jest administrator lub osoba posiadająca dużą kolekcję plików. Wymagane funkcjonalności:

- Możliwość wyboru katalogu zawierających pliki wymagające zmiany nazw
- Udostępnienie filtrów glob pozwalających na automatyczną selekcję plików
- Możliwość ekstrakcji metadanych z plików typu
  - MP3
  - PNG
- Wybór operacji na samych plikach lub pełnych ścieżkach (wraz z katalogami)
- Automatyczna iteracja względem wybranych plików i zmiana ich nazwy
- Notyfikacja o powtarzających się identyfikatorach plików
- Notyfikacja o możliwych problemach z kompatybilnością spowodowanych zastosowanym zestawem znaków

- Notyfikacja o błędach ekstrakcji metadanych
- Zachowywanie konfiguracji programu między uruchomieniami

4.1.2 Wymagania niefunkcjonalne

- Minimalistyczny, skalowalny interfejs użytkownika
- Aplikacja powinna być przenośna na poziomie kodu źródłowego zarówno między platformami z rodziny Microsoft Windows jak i zgodnymi ze standardem `POSIX`.

4.2 Wykorzystane biblioteki

4.2.1 SigC++

SigC++ jest biblioteką dla języka C++ implementującą bezpieczny (ze względu na typy) mechanizm sygnałów. Sygnały (zdarzenia) są wysokopoziomowym odpowiednikiem wywołań zwrotnych używanych do wstrzykiwania kodu programisty-użytkownika do istniejącej implementacji. W językach niskopoziomowych, takich jak C często stosuje się do tego celu wskaźniki do funkcji, jednak ich niskopoziomowa natura może powodować trudne do wykrycia błędy spowodowane przekazaniem złego typu wskaźnika lub błędnej jego sygnatury. Biblioteka udostępnia wysokopoziomowe szablony obiektów sygnałów jak i interfejsy do zastosowania w klasach użytkownika, ułatwiające w znaczny sposób zarządzanie podpiętymi zdarzeniami.

Rysunek 4.1: Logo biblioteki SigC++

SigC++ jest często używana w projektach GUI takich jak projekt pulpitu GNOME; w takim też celu zostanie ona użyta w aplikacji MRU.

4.2.2 `boost::filesystem`

Biblioteka `boost::filesystem` pozwala na niezależny od systemu operacyjnego dostęp do drzewa katalogów. Ze względu na swoją uniwersalność została użyta jako podstawowy sterownik (moduł wyjścia — output module) w aplikacji MRU.

4.2.3 `boost::property_tree`

`boost::property_tree` jest drzewiastym (hierarchicznym) kontenerem ogólnego przeznaczenia[1], który posłuży jako główne źródło informacji o wtyczkach i samym rdzeniu aplikacji MRU.

4.2.4 `boost::program_options`

Biblioteka `boost::program_options` udostępnia wygodny i rozszerzalny parser argumentów przekazanych programowi z linii komend.

4.2.5 wxWidgets

wxWidgets jest wieloplatformową biblioteką do tworzenia graficznych interfejsów użytkownika (ang. GUI). W projekcie została wykorzystana do stworzenia wtyczki interfejsu (ui module) wxWidgetsUi. wxWidgets udostępnia i pozwala tworzyć przenośny zestaw klas kontrolek, które są tłumaczone na natywne kontrolki środowiska uruchamiającego aplikacje.

4.2.6 ICU

ICU — International Components for Unicode-– jest biblioteką opracowaną przez IBM wspierającą lokalizacje, globalizacje i umożliwiającą operacje na łańcuchach znaków w kodowaniach UTF.

[1] Z założenia biblioteka `boost::property_tree` została stworzona do reprezentacji struktury ogólnych plików konfiguracyjnych lecz nic nie stoi na przeszkodzie aby traktować ją jako ogólny kontener

Jako że główne operacje w aplikacji MRU przeprowadzane są na łańcuchach znaków, istotne jest aby wykonywane były one z należytą precyzją. ICU jest najbardziej zaawansowaną, ogólnie dostępną biblioteką tego typu z długą historią zastosowań.

4.3 Rdzeń aplikacji - klasa MruCore

Rdzeniem aplikacji jest klasa `MruCore` stanowi ona interfejs do całej funkcjonalności programu i udostępnia informacje o jego działaniu.

Klasa `MruCore` zawiera metody umożliwiające wtyczkom GUI na kontrolę pracy programu. Sygnały zdefiniowane w klasie dostarczają informacji zwrotnej o pracy aplikacji.

4.4 glue_cast - łącznik technologii

Jako że w aplikacji zostały wykorzystane różne biblioteki, wprowadziły one wiele wymagań co do obsługiwanych typów danych. Biblioteka ICU korzysta głównie z klas takich jak `UnicodeString` podczas gdy biblioteki `boost` zostały oparte na strukturach ze standardowej biblioteki `STL` takich jak `std::string`. Do tego dochodzi niskopoziomowa warstwa API systemu operacyjnego która często operuje na surowych łańcuchach znaków — `const char *`.

Aby ułatwić konwersję między różnymi redundantnymi typami danych, został opracowany szablon `glue_cast` podobny w zastosowaniu do wbudowanych w język język rzutowań takich jak `dynamic_cast` czy `reinterpret_cast`.

Listing 4.1glue.hpp

```
template<typename DstType, typename SrcType> inline
DstType
glue_cast(const SrcType &a_value)
{
  return DstType(a_value);
```

```
}
```

Generyczna implementacja szablonu często jest nieodpowiednia dla typów dla których realizowana jest jego specjalizacja jednak problem ten został rozwiązany — każda para typów używanych w aplikacji posiada dwie jawne specjalizacje szablonu umożliwiające ich konwersję.

Listing 4.2Fragment glue_impl.hpp — specjalizacja dla std::wstring i wxString

```
template<> inline
wxString
glue_cast<wxString, std::wstring>(const std::wstring &a_value)
{
  return wxString(a_value.c_str(), wxConvUTF8);
}

template<> inline
std::wstring
glue_cast<std::wstring, wxString>(const wxString &a_value)
{
  return std::wstring(a_value.wc_str());
}
```

Dzięki wykorzystaniu szablonów nie ma potrzeby tworzenia nowych funkcji konwersji, a całość wygląda bardziej spójnie i jest łatwiejsza w utrzymaniu. Dodatkowym atutem użytego rozwiązania jest jego przenośność — wybrane specjalizacje można wykorzystać w jakimkolwiek projekcie używających specjalizowanych typów. Dodawanie nowych konwersji sprowadza się do dopisania kolejnej pary szablonów.

4.5 Wyrażenia zawierające metatagi

Najważniejszym elementem projektu MRU są metatagi wraz metawyrażeniami na które się składają. Metawyrażenia używane są jak wzorzec (szablon) na podstawie którego generowane są kolejne nazwy plików.

Za każdym razem gdy MRU zmienia plik na którym operuje, metawyrażenie jest ewaluowane. Każde wystąpienie tagu jest przekładane na wywołanie odpowiedniej metody na obiekcie wtyczki, a rezultat tego wywołania jest wstawiany w miejsce wystąpienia tagu. Metatagi są reprezentacjami wywołań do odpowiadającym im wtyczek.

Metatag jest identyfikatorem wprowadzonym do zwykłego tekstu, składającym się z czterech elementów które nie mogą zostać rozdzielone białymi znakami. Metawyrażenie rozpoczyna się od symbolu procent — '%' — po którym następuje nazwa metatagu składająca się ze znaków alfanumerycznych alfabetu łacińskiego[2]. Po nazwie następuje para nawiasów — '(' wraz z ')' — zawierających opcjonalnie listę parametrów inicjalizacyjnych metatag. Nie istnieją ograniczenia co do zawartości listy inizjalizującej — może ona zawierać pełen zakres znaków włączając to znaki zakończenia listy (nawiasy zamykające) o ile są odpowiednio oznaczone[3]. Ostatnim elementem jest opcjonalny zakres działania metatagu — jest to obszar zawierający się między parą nawiasów klamrowych ('{' oraz '}') który sam w sobie jest metawyrażeniem. Dzięki temu, efekty metatagów mogą się na siebie nakładać.

```
%Replace(" ", "_"){wyrażenie}
```

Rysunek 4.2: Metatag z wyróżnionymi elemetami na niego się składającymi

```
%Count(). %MP3(artist) - %MP3(title) [%TextCase(upper){%CRC32()}].mp3
```

Rysunek 4.3: Przykładowe metawyrażenie wraz z wyróżnionymi elementami metatagów

[2] Z technicznego punktu widzenia nic nie stoi na przeszkodzie aby do zapisu nazwy metataga zastosować pełen zestaw znaków, lecz ze względu na globalizacje — nie wszyscy użytkownicy potrafili by używać każdej nazwy — zastosowano wyżej opisaną konwencję.

[3] Aby zignorować interpretację znaku specjalnego w metawyrażeniu, można użyć ogólnie znanego schematu wyłączania znaków — poprzedzania ich symbolem '\'

Parsowanie metawyrażenia rozpoczyna się od tokenizacji — wydzieleniu znaczących dla wyrażenia elementów takich jak symbole (procent, nawiasy), a także ciągi znaków alfanumerycznych oraz białych. Na podstawie listy tokenów budowane jest drzewo wywołań, które jest strukturą zawierającą kolejność oraz zależności między metatagami.

Drzewo wywołań składa się jedynie z metatagów. Aby otrzymać taką strukturę, ciągi surowego tekstu (nie będące metatagami) zostają zamienione na wywołania anonimowych (nienazwanych) metatagów, których argumentami inicjalizującymi są właśnie surowe ciągi tekstu, a jedyną funkcją — zwrócenie argumentów z listy inicjalizującej. Dzięki temu ewaluacja wyrażeń jest prostsza, a dodatkowy anonimowy metatag może zostać wykorzystany na przykład do zmiany kodowania surowego tekstu.

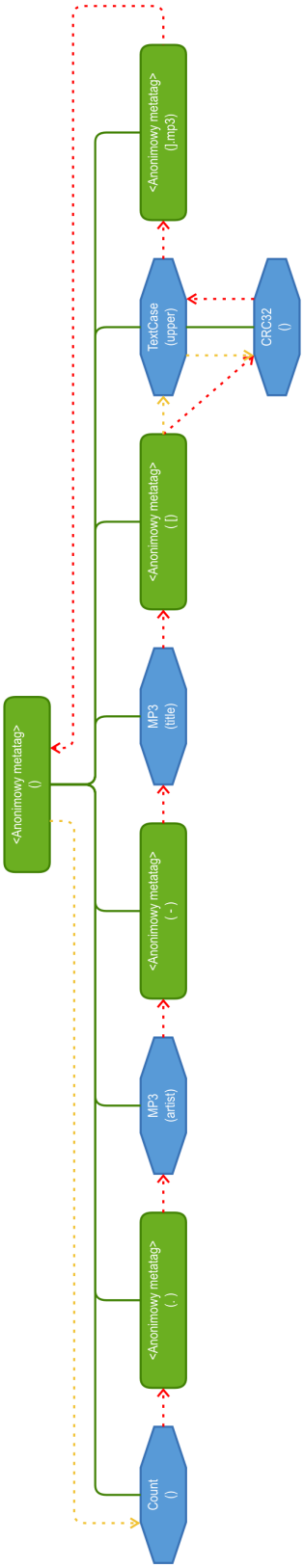
Rysunek 4.4 ukazuję strukturę drzewa zbudowanego z przykładowego wyrażenia z rysunku 4.3. Widoczna tu jest wygenerowana hierarchia, na której szczycie znajduje się anonimowy tag. Pojedynczy korzeń ułatwia parsowanie i wpasowuje się w logiczną strukturę wyrażenia które nawet nie zagnieżdżone może składać się z kilku następujących po sobie elementów.

Drzewo wywołań przeszukiwane jest w głąb (co zostało zaznaczone żółtą przerywaną linią), a jego elementy są ewaluowane od lewej do prawej przy czym metawyrażenia zagnieżdżone w zakresach operacyjnych innych wyrażeń są ewaluowane przed otaczającym je metatagiem-rodzicem. Kolejność ewaluacji jest widoczna na rysunku 4.4 i oznaczona czerwoną, przerywaną linią. W ten sposób rezultat wykonania pod-wyrażenia jest dostępny dla tagu-rodzica, co pozwala na wiązanie wywołań niespotykane w żadnym istniejącym programie.

Każdy obiekt metatagu musi być zgodnym z interfejsem klasy `Metatag`, która zawiera następujące metody wymagające implementacji:

1. `void initialize(const UnicodeString &arguments)`
2. `UnicodeString execute(const UnicodeString &area_of_effect)`

Pierwsza z metod zostaje wywołana na obiekcie podczas łączenia drzewa wywołań z listą fabryk metatagów i pobiera jako parametr łańcuch znaków będący zawartością



Rysunek 4.4: Drzewo wywołań stworzone z przykładowego metawyrażenia

nawiasów tuż po nazwie metatagu. Proces tego typu nazywany jest często *bindowaniem* (od ang. bind).
Metoda `execute` wywoływana jest za każdym razem podczas ewaluacji wyrażenia dla danego pliku. Jej argumentem jest zakres operacyjny (opcjonalne pod-wyrażenie zawarte w nawiasach klamrowych za listą argumentów).
Obie metody mogą wyrzucać wyjątki informując o niepoprawnym argumencie lub błędzie ekstrakcji metadanych.

4.6 System modułów

Aby ułatwić proces projektowania a także zwiększyć rozszerzalność aplikacji, duża część funkcjonalności została oddelegowana do oddzielnych modułów zwanych również wtyczkami. Wtyczki są klasami ładowanymi w trakcie działania programu z bibliotek dynamicznych. W celu udostępnienia aplikacji funkcjonalności zawartych w modułach wtyczek, niezbędne było zaprojektowanie menadżera wtyczek — plugin manager. Klasy menadżera wtyczek umożliwiają programiście-użytkownikowi ładowanie modułów z wcześniej zadeklarowanym interfejsem niezależnie od platformy systemowej na której uruchamiany jest program[4].

Problemem który rozwiązuje menadżer wtyczek jest fakt że biblioteki dynamiczne przechowują głównie funkcje; klasy które istnieją jedynie w trakcie kompilacji nie mogą zostać wyeksportowane do pliku jak ma to miejsce w językach wspierających introspekcje/refleksje typów — takich jak Java czy C#. Aby umożliwić ładowanie wtyczek w języku C++ należy najpierw zdefiniować czym właściwie jest sama wtyczka.

W MRU (jak i wielu innych programach) wtyczka jest obiektem udostępniającym metody określone przez interfejs wtyczki. Także biblioteka dynamiczna musi w jakiś sposób udostępnić owe obiekty.

[4]Same moduły muszą być skompilowane pod platformę na której program ma być uruchamiany.

Menadżer wtyczek po załadowaniu biblioteki dynamicznej przeszukuje ją pod kątem funkcji o nazwie `register_plugins` wyeksportowanej bez przesłaniania nazw (name mangling) np za pomocą konstrukcji `extern ''C'' { ... }` i jeśli takowa istnieje — uruchamia ją przekazując jako argument wskaźnik własną instancje.

Sama wtyczka natomiast rejestruje w instancji przekazanego menadżera fabryki klas w niej zawartych. Dzięki temu, obiekty wtyczek nie są tworzone od czas gdy są faktycznie potrzebne. Zmniejsza to obciążenie pamięciowe programu jak i ułatwia pracę twórcom wtyczek, którzy mogą skupić się na implementowaniu faktycznej funkcjonalności modułów. Takie rozwiązanie pozwala również programowi-hostowi na decydowanie ile i kiedy mają być tworzone wybrane obiekty.

Z założenia menadżer wtyczek powinien umożliwiać ładowanie wielu wtyczek z jednej biblioteki dynamicznej. Problem ten został rozwiązany dzięki zastosowaniu klasy identyfikatorów interfejsów — każdy menażer i każda fabryka wtyczki posiada identyfikator informujący jaki typ interfejsu obsługuje. Dzięki zastosowaniu dystrybutora (brokera) fabryk, podczas ładowania modułu możliwe jest rejestrowanie fabryk wtyczek różnych interfejsów pod warunkiem że w czasie ładowania stworzone zostały ich instancje.

Poniżej został przedstawiony przykładowy interfejs wtyczki, jej implementacja oraz program ją wykorzystujący.

Przykładowa wtyczka posiada jedynie jedną metodę wywoływaną przez aplikację-hosta: `say_hello`.

Aby umożliwić integracje klasy interfejsu MyPlugin z menadżerem wtyczek, każdy interfejs musi dziedziczyć z szablonu `mru::plugin`, który to udostępnia metody do pobrania identyfikatorów interfejsu. Makro `PLUGIN_INTERFACE` używane jest do zdefiniowania statycznej funkcji `static_interface_name`, zwracającej identyfikator interfejsu klasy[5]. Konstruktor interfejsu wymaga natomiast przekazania identyfikatora konkretnej implementacji — wymusza to dostarczenie przez konkretne implementacje identyfikatorów wtyczek i ułatwia to identyfikacje instancji wtyczek.

[5]Najwygodniejszym rozwiązaniem zdaje się być stosowanie nazwy typu (klasy) jako nazwy interfejsu.

Listing 4.3test_module.hpp

```
#include <plugin_manager.hpp>

class MyPlugin : public mru::plugin<MyPlugin> {
public:
  PLUGIN_INTERFACE("MyPlugin")
  MyPlugin(const mru::name_type &a_name)
    : mru::plugin<MyPlugin>(static_interface_name(), a_name)
  { }
  virtual void say_hello(void) = 0;
};

typedef mru::plugin_manager<MyPlugin> MyPluginManager;
```

Implementacja wtyczki typu `MyPlugin` o nazwie `MPlg1` jest równie prosta i wymaga od programisty-użytkownika biblioteki jedynie dostarczenia identyfikatora instancji — metody `static_implementation_name`, który podobnie jak metoda do identyfikacji interfejsu zwykle zwraca nazwę typu klasy.
Warto w tym miejscu zauważyć że biblioteka dynamiczna stworzona zawierająca implementacje wtyczki powinna udostępniać funkcję `register_plugins`. Aby zmniejszyć powtarzalność kodu, zastosowano w tym celu makra: `EXPORT_START`, `EXPORT_END` i `EXPORT_PLUGIN`

Listing 4.4test_module.cpp

```
#include "test_module.hpp"

class MPlg1 : public MyPlugin {
public:
  PLUGIN_NAME("MPlg1");
  MPlg1(void)
```

```
    : MyPlugin(static_implementation_name()) { }

  void say_hello(void)
  {
    std::cout << "Hello from MPlg1!" << std::endl;
  }
};

EXPORT_START
  EXPORT_PLUGIN(MPlg1)
EXPORT_END
```

Program korzystający z wtyczek musi stworzyć instancję menadżera wtyczek — w tym przypadku specjalizacji szablonu `mru::plugin_manager<MyPlugin>`.
Po utworzeniu menadżer wtyczek umożliwia ładowanie bibliotek dynamicznych (za pomocą metody `load_module`). Po załadowaniu biblioteki możliwe jest odpytanie menadżera o nazwy dostępnych wtyczek, służy ku temu metoda `available_plugins`. Zadaniem menadżera również jest zarządzanie czasem życia obiektu wtyczki, które tworzone są za pomocą metody `create_plugin` i niszczone z użyciem `destroy_plugin`;

Listing 4.5main.cpp

```
#define PLUGIN_HOST
#include "test_module.hpp"

int
main(int argc, char const *argv[])
{
  using namespace mru;
  MyPluginManager::set_instance(new MyPluginManager("MyPlugin"));
```

```
  MyPluginManager* my_pm = MyPluginManager::get_instance();

  if(0 == my_pm->load_module("./test_module")) {
    //ERR("No module named test_module1 found");
    return 1;
  }

  std::list<name_type> my_plugins = my_pm->available_plugins();
  std::cout << my_plugins.size() << std::endl;
  for(std::list<name_type>::iterator i = my_plugins.begin(); i != my_plu
    std::cout << *i << std::endl;
  }

  MyPlugin *mplg1 = my_pm->create_plugin("MPlg1");

  if(mplg1)
    mplg1->say_hello();

  my_pm->destroy_plugin(mplg1);

  my_pm->destroy();
  my_pm = NULL;

  return 0;
}
```

4.7 Typy modułów w MRU

Aplikacja obsługuje trzy interfejsy wtyczek:

Rysunek 4.5: Okno aplikacji MRU — wtyczka wxWidgetsUi

- UiPlugin - moduły interfejsu; pozwalają na implementacje różnych interfejsów użytkownika.
- OutputPlugin - sterowniki wyjścia — umożliwiają korzystanie z różnych interfejsów do systemu plików.
- TagPlugin - moduły udostępniające fabryki do tworzenia wszelkich metatagów.

4.8 Moduły UI

Wtyczki interfejsu użytkownika pozwalają użytkownikowi końcowemu na interakcję z programem. Pojedynczy proces aplikacji może posiadać aktywną tylko jedną wtyczkę interfejsu. Decyzja o wyborze interfejsu użytkownika dokonywana jest na podstawie pliku konfiguracyjnego lub odpowiedniego przełącznika linii poleceń. Wtyczki interfejsu odpowiadają za całkowitą komunikację między użytkownikiem i rdzeniem aplikacji — MruCore; to one udostępniają większość funkcjonalności aplikacji, a także informują użytkownika o stanie programu.

Jako że funkcjonalność aplikacji jest w dużej mierze determinowana przez klasę rdzenia (MruCore), interfejs UiPlugin nie posiada z góry zdefiniowanych metod jak inne wtyczki. Jedyna metoda w nim zawarta — `start` — pozwala na reinterpretacje linii poleceń i służy do przekazania kontroli nad programem (klasą MruCore) właśnie do samej wtyczki.

4.8.1 wxWidgetsUi

wxWidgetsUi jest implementacją graficznego interfejsu użytkownika opartego na wspomnianej bibliotece `wxWidgets`. Założeniem tego modułu jest udostępnienie użytkownikowi końcowemu prostego oraz szybkiego dostępu do funkcjonalności programu, a także pomoc w zapoznaniu się z aplikacją.

Okno aplikacji stworzone przez wtyczkę wxWidgetsUi jest podzielone na trzy sekcje:

- Sekcja górna odpowiada za selekcję plików oraz pozwala na edycję metawyrażenia które ma zostać zastosowane na wybranych plikach. W lewym górnym rogu widnieje lista dostępnych Metatagów, a pola po prawej stronie pozwalają na wybór katalogu, filtru glob oraz samego metawyrażenia.

- Środkowa część okna stanowi podgląd wybranych plików jak i efektów zastosowania edytowanego wyrażenia do nich. Lista plików może być ograniczona i odświeżana w zależności od opcji znajdujących się pod nią.

- Na dole okna widoczne są przyciski do (ręcznego) generowania podglądu, jego konfiguracji, a także rozpoczęcia transformacji nazw dla wybranych plików.

4.8.2 TextUi

TextUi jest wtyczką interfejsu udostępniającą funkcjonalność programu z poziomu linii komend. Pozwala ona na przekazanie parametrów konfiguracyjnych i rozpoczęcie transformacji nazw bez interakcji z użytkownikiem jak ma to miejsce w przypadku graficznych interfejsów użytkownika. Dzięki zastosowaniu tej wtyczki istnieje możliwość wykorzystania aplikacji MRU z poziomu skryptów powłoki, na maszynach nie wykorzystujących środowiska graficznego lub zdalnych.

4.9 Moduły output

Wtyczki wyjścia są warstwą abstrakcji pomiędzy systemem operacyjnym i jego drzewem katalogów, a rdzeniem aplikacji. Udostępniają one iteratory pozwalające na przeszukiwanie dysku w celu znalezienia plików pasujących do wzorca wybranego przez użytkownika, oraz przekazują polecenia zmiany identyfikatora pliku do API używanego systemu. Kontrolują one również poprawność wygenerowanych nazw, a także zapewniają ich unikalność.

4.9.1 GenericBoost

Wtyczka GenericBoost została opracowana na podstawie biblioteki `boost::filesystem`. Stanowi ona sprawdzone oraz przenośne rozwiązaniem problemu dostępu do drzewa katalogów, bezpieczne do wykorzystania na wielu systemach bez zmian w kodzie samej wtyczki.

4.10 Moduły metatagów

Główna funkcjonalność aplikacji została zawarta w modułach tagów — to one odpowiadają ekstrakcje metadanych lub generowanie wartości, które rdzeń aplikacji jedynie składa i przesyła wraz z komunikatem zmiany do systemu plików.

Każdy z poniżej wymienionych tagów może zostać dodany do wyrażenia po załadowaniu odpowiedniej biblioteki dynamicznej go zawierającej[6].

4.10.1 Count

Tag Count jest używany do numeracji wybranych plików. Dla każdego pliku generowany jest kolejny numer. Lista argumentów tagu pozwala na określenie wartości początkowej, prefiksu oraz systemu w którym ma odbywać się numeracja. W poniższej tabeli zawarte zostały parametry obsługiwane przez metatag:

Argument	**Opis**
start=N	Ustawia początkowy stan licznika na N— od tej wartości tag rozpocznie zliczanie
step=N	Ustawia rozmiar kroku — kolejny numer będzie większy o N w stosunku do poprzedniego

Tablica 4.1: Zestaw argumentów inicjalizacyjnych dla metatagu Count

[6]Część tagów nie wymaga ładowania — są wbudowane w plik wykonywalny aplikacji, natomiast część mimo iż jest dostarczana w standardzie z aplikacją może wymagać dodatkowej konfiguracji w postaci określenia ścieżki ładowania bibliotek

Aby wykorzystać kilka argumentów jednocześnie należy oddzielić je od siebie za pomocą symbolu przecinka — ’**,**’.

4.10.2 MP3

Tag MP3 pozwala na ekstrakcję danych z plików audio zakodowanych w standardzie MPEG-1/MPEG-2 Audio Layer 3 oraz zawierających tagi ID3. Tag ten obsługuje następujące argumenty:

Argument	**Opis**
title	Konfiguruje tag do ekstrakcji tytułu utworu
artist	Konfiguruje tag do ekstrakcji nazwy artysty wykonującego utwór
album	Konfiguruje tag do ekstrakcji nazwy albumu w którym zawiera się utwór
year	Konfiguruje tag do ekstrakcji roku powstania utworu
comment	Konfiguruje tag do ekstrakcji komentarza

Tablica 4.2: Zestaw argumentów inicjalizacyjnych dla metatagu MP3

4.10.3 CRC32

Metatg CRC32 służy liczenia cyklicznej sumy kontrolnej CRC o wielkości słowa 32 bity. Zwracana suma kontrolna jest sformatowana jako wartość heksadecymalna.

4.10.4 TextCase

Tag TextCase jest używany do zmiany wielkości liter w skojarzonym z tagiem zakresie działania. Pewność działania dla pełnego zakresu kodów unicode jest zapewniona dzięki wykorzystaniu funkcji z biblioteki ICU.

4.11 Testy

<!TODO!>

Argument	Opis
upper	Konfiguruje tag do zamiany wszystkich znaków w zakresie na ich większe odpowiedniki
lower	Konfiguruje tag do zamiany wszystkich znaków w zakresie na ich mniejsze odpowiedniki
title	Konfiguruje tag do zamiany znaków w zakresie tak by wyglądały na tytuł (Pierwsze znaki każdego słowa są zamieniane na ich większe odpowiedniki

Tablica 4.3: Zestaw argumentów inicjalizacyjnych dla metatagu TextCase

4.12 Możliwości rozwoju i ponownego wykorzystania komponentów

Dzięki zastosowanej architekturze modułowej, części aplikacji nie posiadają dużych zależności między-modułowych co stwarza idealne warunki do ich rozwoju i ponownego użycia istniejącego kodu (na przykład wtyczek) w innych projektach.

System wtyczek został zaimplementowany w formie biblioteki niezależnej od platformy i nie posiadającej rozwiązań specyficznych dla aplikacji w której został wykorzystany. Dzięki temu jego wykorzystanie w innych projektach nie wymaga dodatkowego narzutu związanego z modyfikacją kodu.

Moduły metatagów wymagają jedynie dostępu do pliku na którym mają operować i same nie są świadome metawyrażeń w których występują. Dzięki zastosowanie takiej izolacji, dodawanie nowych wtyczek nie wywiera wpływu na działający program, a istniejące metatagi mogą zostać wykorzystane z powodzeniem w innych aplikacjach, które wymagają dostępu do metadanych pliku. Dzięki spójnemu interfejsowi, metatagi mogą stanowić alternatywę dla wykorzystywania dedykowanych bibliotek do obsługi formatów (plików), które często zawierają sporo narzutu związanego z funkcjami bezpośrednio nie związanymi z danymi — jak na przykład zapisem faktycznych danych.
Metatagi mo1gą znaleźć również zastosowanie przy porównywaniu plików w celu identyfikacji duplikatów.

Wtyczki wyjścia — OutputPlugin — nie mają w żaden sposób narzuconej implementacji. Ich zadaniem jest jedynie udostępnienie iteratora do faktycznych plików. Nic nie stoi na przeszkodzie aby stworzyć wtyczkę która generuje raporty dla istniejących plików zawierające metadane w nich zawarte.

Rozdział 5.

Wnioski

Wiele bibliotek i szczegółów implementacji sprawiło że projekt pracy inżynierskiej okazał się nieco trudniejszy niż było to przewidywane. Dużą częścią pracy stanowiło połączenie istniejących bibliotek i technologii aby mogły ze sobą współpracować. Różnica zaawansowania oraz styli interfejsów użytych technologii wymusiła tworzenie dodatkowych abstrakcji lecz dzięki temu, pozwoliła także na mniejsze zależności między-modułowe co zaowocowało powstaniem aplikacji o dużych możliwościach rozwoju. Omówiona aplikacja może być dalej rozwijana jako projekt spen-source.

Stworzona aplikacja sprostała założeniom pod które została zaprojektowana i stanowi dzięki temu użyteczne narzędzie które może pozwolić ludziom na to na co zostały stworzone komputery — zautomatyzowanie monotonnych czynności i przyspieszenie pracy.

Dodatkową zaletą wykonanej aplikacji jest fakt że niektóre jej części (takie jam menadżer wtyczek) dzięki swojej uniwersalności mogą posłużyć do budowy kolejnych programów.