

POLITECHNIKA BIAŁOSTOCKA

Wydział Informatyki

Paweł Żukowski

**Projekt i implementacja narzędzia do masowej
refaktoryzacji identyfikatorów plików z
uwzględnieniem ich zawartości**

PRACA INŻYNIERSKA

Promotor
dr inż. Marcin Skoczyła

Białystok 2013

Streszczenie

Niniejsza praca inżynierska na temat "Projekt i implementacja narzędzia do masowej refaktoryzacji identyfikatorów plików z uwzględnieniem ich zawartości" opisuje projekt architektury oraz implementację programu Multifile Renaming Utility.

Multifile Renaming Utility — w skrócie MRU — jest programem narzędziowym mającym na celu umożliwienie automatycznej zmiany nazw wielu plikom ze względu na metadane w nich zawarte.

Praca jest podzielona na sześć rozdziałów zawierających teoretyczny jak i praktyczny opis problemu i jego rozwiązania.

Pierwszy rozdział zawiera wstępny opis projektu i motywację do jego utworzenia.

Drugi rozdział zawiera teoretyczne podstawy problemu oraz opisuje terminologię użytą w pracy.

Trzeci rozdział przedstawia przykłady istniejących aplikacji wraz z ich subiektywną oceną pod względem skuteczności w stosunku do przedstawionego problemu.

Czwarty rozdział opisuje środowisko pracy, które zostało wykorzystane do stworzenia implementacji.

Piąty rozdział zawiera opis architektury, a także szczegóły implementacji gotowej aplikacji.

Ostatni, szósty rozdział składa się z wniosków na temat wykonanego projektu.

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

Summary

Following Engineer's Degree Thesis, titled "Project and implementation of content aware, file renaming tool" describes architecture project and implementation of Multifile Renaming Utility software program.

Multifile Renaming Utility — MRU for short — is a tool application which aims to allow the user to rename many files based on metadata contained therein.

This thesis is divided into six chapters containing theoretical and practical description of the problem and its solutions.

First chapter contains introduction to the project and motivation which led to its creation.

Second chapter describes theoretical fundamentals of described problem and terminology used in the work.

Third chapter is a review of few existing applications that aim to solve described problem.

Fourth section introduces work environment in which application were developed.

Fifth chapter contains architecture description as well as implementation details of the finished application.

The last, sixth chapter consists of the conclusions made about the project.

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

Spis treści

Spis treści	4
1 Wstęp	7
1.1 Motywacja	7
1.2 Cel i zakres pracy	8
1.3 Założenia	9
1.4 Plan pracy	9
2 Teoria	10
2.1 Dane w systemie komputerowym	10
2.2 Systemy plików i identyfikacja danych	11
2.2.1 Katalogi i ścieżki do plików	12
2.2.2 Różnice w identyfikacji plików wśród różnych systemów operacyjnych	12
2.3 Metadane	14
3 Przegląd istniejących rozwiązań	16
3.1 Bulk Rename Utility	16
3.2 Métamorphose	17
3.3 KRename	18
3.4 Inne rozwiązania	19
4 Środowisko pracy	21
4.1 Język C++	21

4.1.1	LLVM Clang	21
4.2	System operacyjny FreeBSD	22
4.3	Mercurial	22
4.4	CMake	23
4.5	Vim	23
5	Implementacja	24
5.1	Specyfikacja wymagań	24
5.1.1	Wymagania funkcjonalne	24
5.1.2	Wymagania niefunkcjonalne	24
5.2	Wykorzystane biblioteki	25
5.2.1	SigC++	25
5.2.2	<code>boost::filesystem</code>	25
5.2.3	wxWidgets	26
5.2.4	ICU	26
5.2.5	TagLib	26
5.3	Rdzeń aplikacji - klasa MruCore	26
5.4	<code>glue_cast</code> - łącznik technologii	27
5.5	Wyrażenia zawierające metatagi	28
5.6	System modułów	32
5.7	Typy modułów w MRU	36
5.8	Moduły UI	37
5.8.1	wxWidgetsUi	37
5.9	Moduły output	38
5.9.1	GenericBoost	38
5.10	Moduły metatagów	39
5.10.1	Count	39
5.10.2	Audio	39
5.10.3	Name	40

5.10.4 Ext	40
5.10.5 Dir	40
5.10.6 TextCase	41
5.11 Testy	41
5.12 Możliwości rozwoju i ponownego wykorzystania komponentów	42
6 Wnioski	45
Bibliografia	46

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. Dodatkowo, niektórzy administratorzy zarządzający serwerami aplikacji zmagają się z problemem wielkiej ilości plików generowanych przez użytkowników ich systemów.

1.1 Motywacja

Tysiące plików mogą stworzyć gęszcz 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 często na pojedyncze typy plików, a także wymagają od użytkownika przyzwyczajenia się do ich używania. Z drugiej strony, przeciętny użytkownik jest zwykle przyzwyczajony do standardowego

programu oferowanego przez większość 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 trzymany 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 metadanych w nich zawartych, a także ich stosowanie do zbiorów plików wybranych przez użytkownika.

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 bez ingerencji w istniejący kod źródłowy.

1.4 Plan pracy

W rozdziale 2 opisano teoretyczny schemat przechowywania danych oraz genezę systemów oraz identyfikacji plików. Rozdział 4 zawiera opis środowiska wykorzystanego do stworzenia projektu i implementacji aplikacji. Rozdział 5 zawiera szczegółowy opis architektury aplikacji wraz z rozwiązaniami wykorzystanymi do jej stworzenia. Ostatni rozdział — 6 — zawiera wnioski na temat wykonanego projektu.

Rozdział 2.

Teoria

W niniejszym rozdziale postaram się przybliżyć obraz problemu identyfikatorów (zwanym 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 potrzeby 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 bardziej pojemnej pamięci — dysku twardego.

Pojemności pierwszych dysków twardych stanowiły promil dzisiejszych jednostek toteż nie wymagały stosowania systemów plików — były po prostu nieulotnym rozszerzeniem pamięci operacyjnej RAM. Jednak wraz ze zwiększeniem ich pojemności oraz generalizacją

oprogramowania, pojawiła się potrzeba standaryzowania i kategoryzacji przechowywanych na dyskach danych, która spowodowała powstanie systemów 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 udostępniający API¹, a także blokuje lub zezwala na dostęp do danych ze względu na uprawnienia użytkownika, programu lub samego zasobu będącego podmiotem zapytania programu.

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 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.

¹API — Application Programming Interface

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, które identyfikują zasób w drzewie katalogów systemu 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 albo plikiem albo dowiązaniem do innego węzła. Dodatkowo węzły katalogów jako jedyne mogą 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 katalogu².

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, takie jak Apple MacOS, 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`

²W praktyce ilość plików które mogą należeć do jednego węzła zależy od rozmiaru licznika użytego w implementacji.

Systemy operacyjne z rodziny Windows korporacji Microsoft³ 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:

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

Dodatkowo w przypadku obu⁴ 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. 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. Takie zachowanie nie występuje w systemach kompatybilnych ze standardem 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.

³Istnieje więcej systemów operacyjnych używających podobnego schematu

⁴System 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 lub ostatniej modyfikacji czy też 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 wymaga informacji o typie.

Niektóre formaty plików (szczególnie multimedialnych) pozwalają na integrację metadanych z samym plikiem. Jako że pliki (szczególnie binarne) mogą stosować dowolną strukturę zapisu, nie istnieje ogólny algorytm wyciągnięcia zawartych w ten sposób informacji. Na szczęście popularne formaty plików posiadają wiele implementacji bibliotek umożliwiających dostęp do danych w nich zawartych.

Do metadanych można zaliczyć także dane generowane takie jak sumy kontrolne, które są wartościami liczonymi na podstawie zawartości samego pliku. Sumy kontrolne używane są w celu testów integralności czy identyczności.

Rozdział 3.

Przegląd istniejących rozwiązań

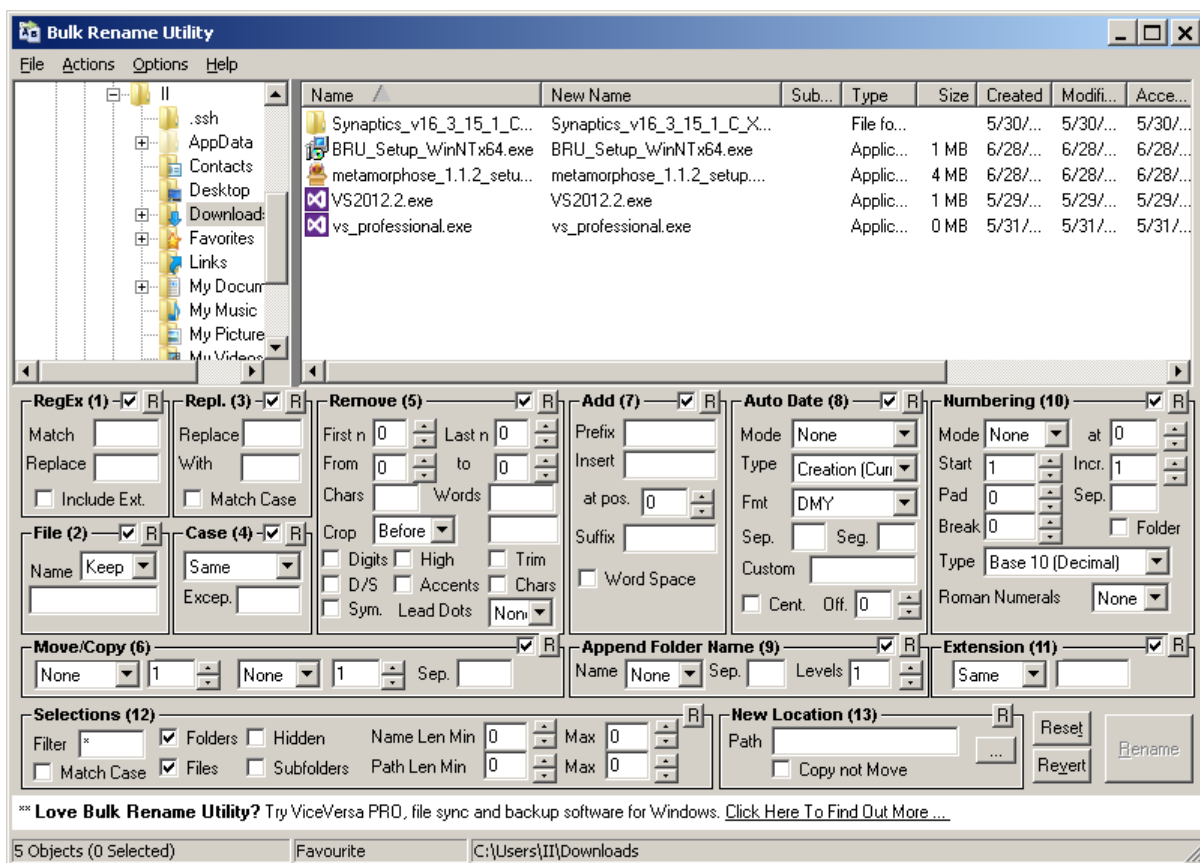
Jako że problem zmiany identyfikatorów plików znany już jest od lat, na rynku istnieje wiele programów próbujących się z nim uporać. Wiele z istniejących rozwiązań zostało zaprojektowanych dla plików konkretnego typu lub są modułami większych aplikacji lecz istnieje kilka¹ implementacji gotowych do ogólnego zastosowania.

Poniżej zostały przedstawione trzy wybrane implementacje wraz z subiektywną opinią o nich.

3.1 Bulk Rename Utility

Jednym z bardziej zaawansowanych i polecanych programów na platformę Microsoft Windows jest *Bulk Rename Utility*. Aplikacja umożliwia ekstrakcje metadanych z plików audio (z tagów ID3v1) i obrazów zawierających dane EXIF. Posiada ona także wiele funkcjonalności związanych z modyfikacją istniejącej nazwy — takich jak zastępowanie z użyciem wyrażeń regularnych. Program wyróżnia się wsparciem dla modyfikacji nazw i atrybutów katalogów, a także zwartym interfejsem. Narzędzie to nie wspiera jednak zmiany kolejności wykonywania działań na nazwie — wszystkie operacje mają swoją pozycję w kolejce wywołania i istnieje jedynie możliwość ich włączenia lub wyłączenia.

¹Aplikacje zostały wybrane ze względu na ich popularność i podejście do rozwiązania problemu

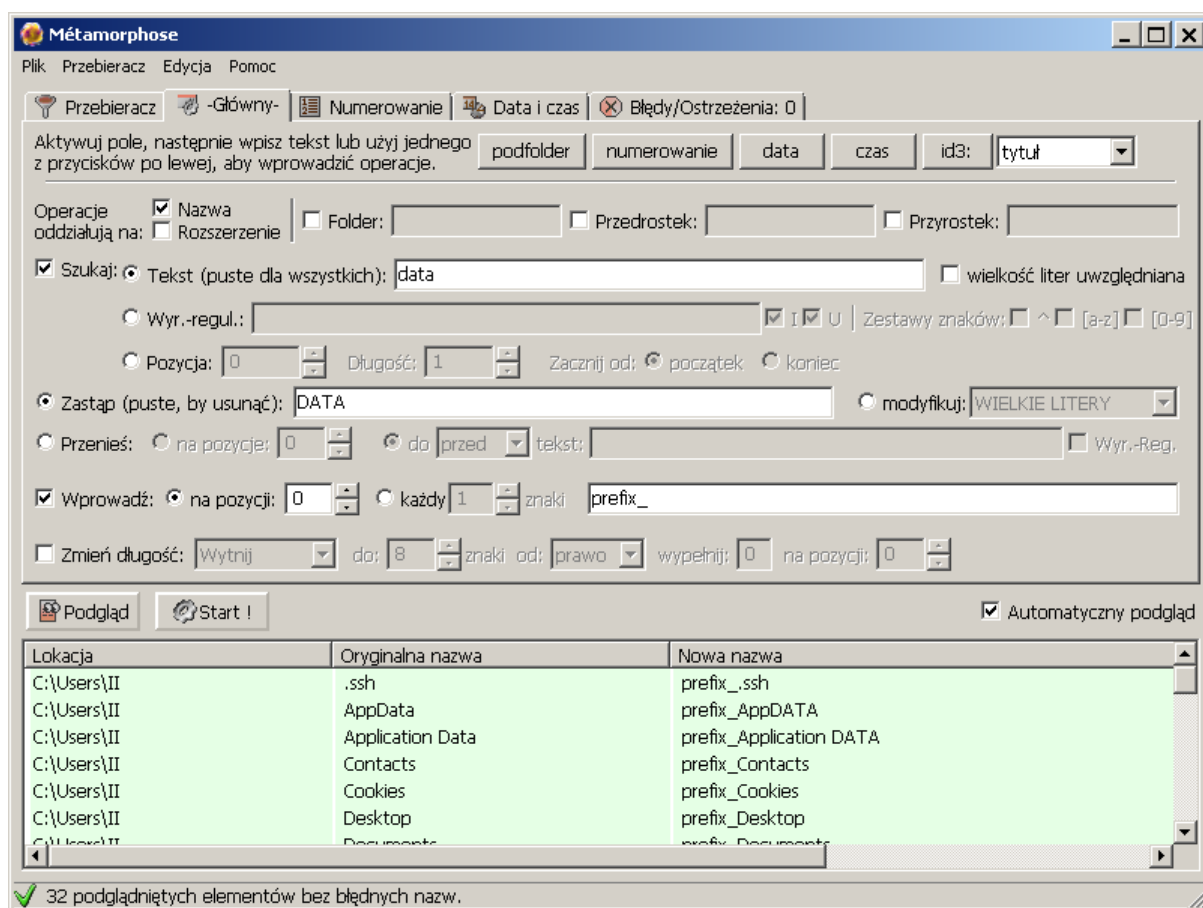


Rysunek 3.1: Okno główne programu Bulk Rename Utility

Bulk Rename Utility posiada również odpowiednik bez interfejsu graficznego — *Bulk Rename Command* — który jest oddzielnym programem udostępniającym funkcjonalność programu z linii poleceń.

3.2 Métamorphose

Métamorphose podobnie jak *Bulk Rename Utility* korzysta z wbudowanego zestawu funkcjonalności jednak posiada pewne wsparcie dla szablonów nazw plików. Jest również aplikacją wieloplatformową, a także dzięki zastosowaniu zakładek — bardziej przejrzystą.



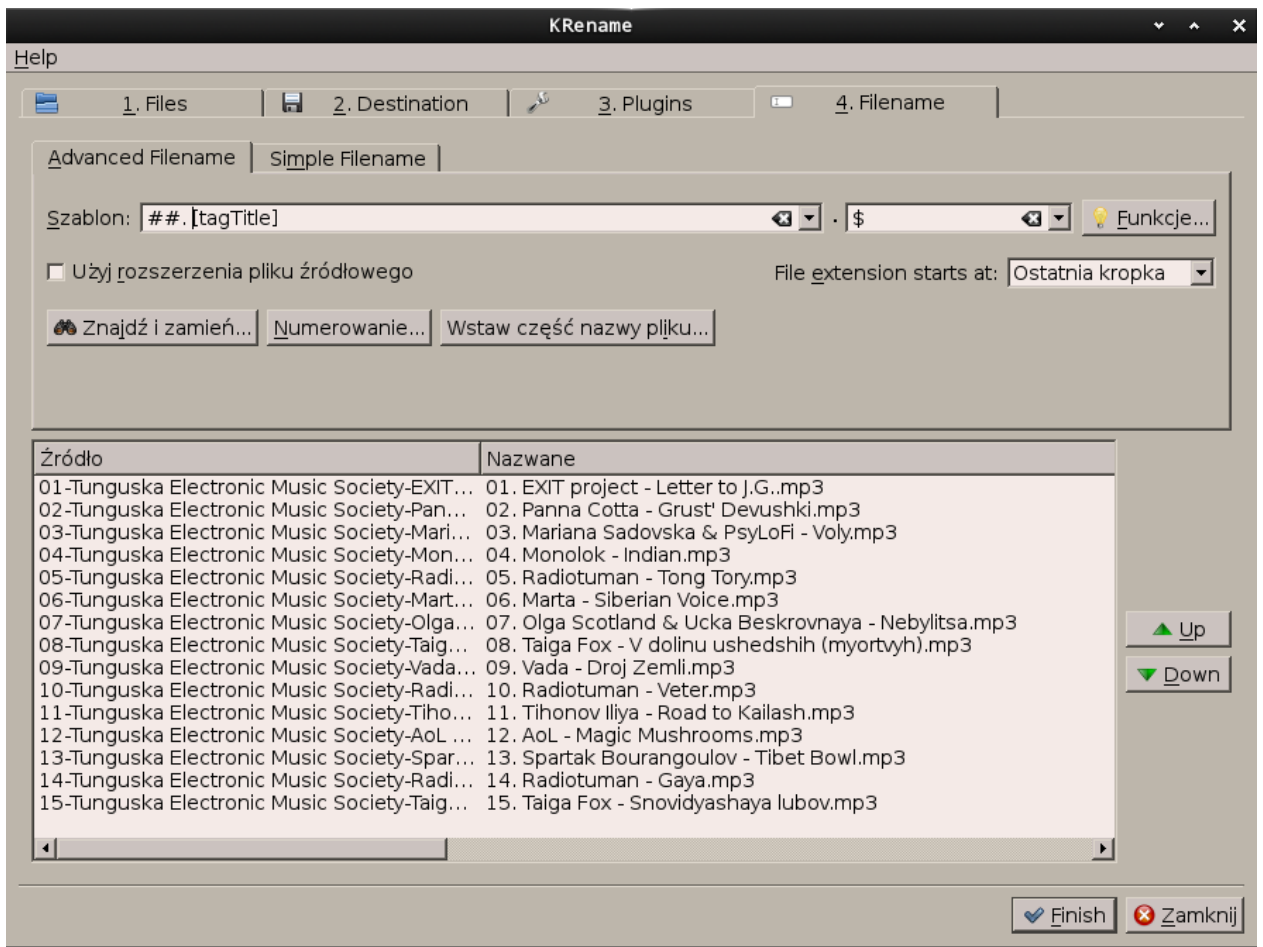
Rysunek 3.2: Jedna z zakładek programu MétaMorphose

3.3 KRename

KRename w odróżnieniu od poprzednich programów nie posiada wersji dla systemów Windows. Zestaw zakładek pozwala na znalezienie plików, wybranie akcji do wykonania, a także przegląd i edycję wtyczek umożliwiających ekstrakcję danych.

Poza trybem edycji szablonu dla nazw plików istnieje prostszy interfejs pozwalający na podstawowe operacje dodania sufiksu lub prefiksu, a także zmianę wielkości znaków w nazwie.

Zaletą programu jest duży wybór wtyczek pozwalających także na modyfikacje metadanych a nawet zawarcie w nowej nazwie rezultatu wywołania kodu JavaScript.



Rysunek 3.3: Konfiguracja szablonu nazwy pliku w KRename

3.4 Inne rozwiązania

Osoby korzystające z nowoczesnych systemów POSIX-owych posiadają ciekawą alternatywę dla jakichkolwiek specjalizowanych aplikacji — tekstową powłokę zwaną shellem.

Nowoczesne powłoki shell takie jak **zsh** czy **bash** posiadają funkcjonalności umożliwiające łączenie wyników wywołań wielu komend co wraz z bogatą liczbą programów dostępnych dla wspomnianych systemów, umożliwia stworzenie polecenia które mogłoby w prosty

modyfikować identyfikatory plików.

Listing 3.1Polecenie powłoki zmieniające rozszerzenia plików JPEG

```
find ./ -name *.JPG -exec rename -v 's /\.JPG /\.jpg/' {} \;
```

Na listingu 3.1 zostało pokazane przykładowe polecenie powłoki zamieniające rozszerzenia plików JPEG z '*JPG*' na '*jpg*'. Wykorzystuje ono dwa programy:

- **find** — znajduje pliki o rozszerzeniu kończącym się na '*JPG*' (przez zastosowanie parametru **-name *.JPG**)
- **rename** — dokonuje faktycznej zmiany nazwy dla pojedynczego pliku

Polecenia realizujące proste zmiany nazw mogą zostać napisane przez średnio-wprawionego użytkownika powłoki jednak bardziej zaawansowane wymagają użycie wielu programów i nie są tak trywialne jak wyżej wymieniony przykład.

Rozdział 4.

Środowisko pracy

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

4.1 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 C¹ pozwala na wykorzystanie wielu istniejących bibliotek napisanych zarówno w C jak i C++.

4.1.1 LLVM Clang

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

¹C++ 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 parserem² języków C i C++ dla architektury LLVM. Projekt jest otwarty (wydawany na licencji BSD) i zdobywa coraz większą popularność³ dorównując, a nawet przewyższając w niektórych testach GCC⁴.

4.2 System operacyjny FreeBSD

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

4.3 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 może być trzymane na każdej maszynie z której korzysta programista, a praca różnych programistów (zmiany w kodzie) może być synchronizowana między nimi samymi.



Rysunek 4.1: Logo systemu Mercurial

²Clang jest określany jako '*frontend*' lecz słowo to nie ma dobrego odpowiednika w języku polskim, a główną funkcjonalnością tego narzędzia jest właśnie parsowanie plików źródłowych z kodem C lub C++ do kodu pośredniego LLVM

³Od listopada 2012 Clang wraz z LLVM stał się domyślnym kompilatorem dla systemu FreeBSD

⁴GNU Compiler Collection

4.4 CMake

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



Rysunek 4.2: Logo narzędzia CMake

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.

4.5 Vim

Edytor Vim jest rozszerzoną wersją klasycznego edytora `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, koloruje składnie plików źródłowych i posiada wiele komend ułatwiających produkcję kodu. Dzięki wtyczkom istnieje możliwość rozszerzenia go o zaawansowane kompletowanie składni czy także szybkie wstawki kodu (ang. snippets).



Rysunek 4.3: Logo edytora Vim

Rozdział 5.

Implementacja

5.1 Specyfikacja wymagań

5.1.1 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 audio (MP3)
- Wybór operacji na samych plikach lub pełnych ścieżkach (wraz z katalogami)
- Automatyczna iteracja względem wybranych plików i zmiana ich nazwy
- Notyfikacja o błędach ekstrakcji metadanych

5.1.2 Wymagania niefunkcjonalne

- Minimalistyczny, skalowalny interfejs użytkownika

- Aplikacja powinna być przenośna na poziomie kodu źródłowego zarówno między platformami zgodnymi ze standardem POSIX.

5.2 Wykorzystane biblioteki

5.2.1 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 5.1: 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.

5.2.2 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.

5.2.3 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.



Rysunek 5.2: Logo biblioteki

wxWidgets

5.2.4 ICU

ICU — International Components for Unicode— jest biblioteką opracowaną przez IBM wspierającą lokalizację, globalizację i umożliwiającą operacje na łańcuchach znaków w kodowaniach UTF.

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.

5.2.5 TagLib

TagLib jest biblioteką pozwalającą na ekstrakcję metadanych z wielu typów plików multimedialnych. W metatagu Audio umożliwia dostęp do informacji o tytule, roku wydania, a także wykonawcy i albumie do którego należy utwór.

5.3 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 UI na kontrolę pracy programu bez implementacji powtarzalnej logiki, a sygnały zdefiniowane w klasie dostarczają informacji zwrotnej o pracy aplikacji.

5.4 `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 5.1 `glue.hpp`

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

Przedstawiona powyżej 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 5.2 Fragment 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.

5.5 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. Metatag rozpoczyna się od symbolu procent — '%' — po którym następuje nazwa metatagu składająca się ze znaków alfanumerycznych alfabetu łacińskiego¹. Po nazwie następuje para nawiasów —

`%Replace(" ", "_") {wyrażenie}`

Rysunek 5.3: Metatag z wyróżnionymi elementami na niego się składającymi

'(' wraz z ') ' — zawierających opcjonalnie listę parametrów inicjalizacyjnych metatag. Nie istnieją ograniczenia co do zawartości listy inicjalizującej — może ona zawierać pełen zakres znaków włączając to znaki zakończenia listy (nawiasy zamykające) o ile są odpowiednio oznaczone². 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.

`%Count()._MP3(artist) - _MP3(title) [%TextCase(upper)] {CRC32()}].mp3`

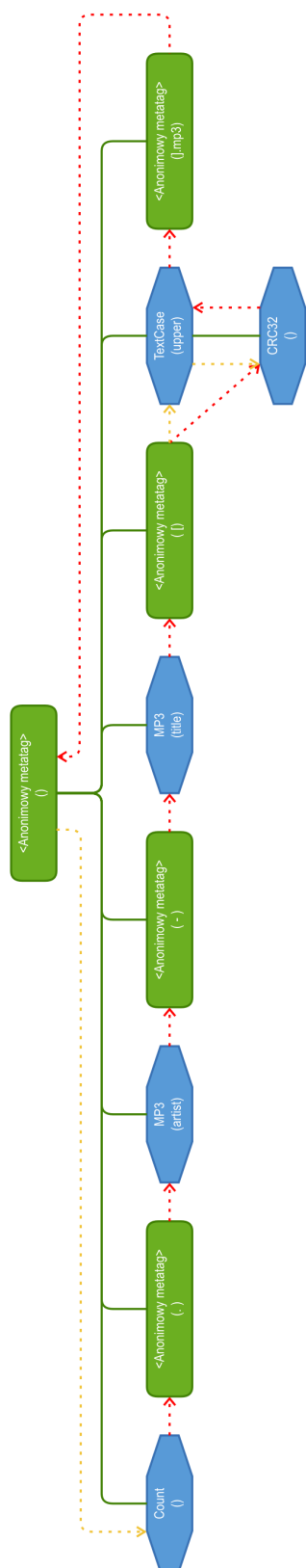
Rysunek 5.4: Przykładowe metawyrażenie wraz z wyróżnionymi elementami metatagów

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

¹Z technicznego punktu widzenia nic nie stoi na przeszkodzie aby do zapisu nazwy metataga zastosować pełen zestaw znaków, lecz ze względu na globalizację — nie wszyscy użytkownicy potrafili by używać każdej nazwy — zastosowano wyżej opisaną konwencję.

²Aby zignorować interpretację znaku specjalnego w metawyrażeniu, można użyć ogólnie znanego schematu wyłączania znaków — poprzedzania ich symbolem '\'



Rysunek 5.5: Drzewo wywołań stworzone z przykładowego metawyrażenia

(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 5.5 ukazuje strukturę drzewa zbudowanego z przykładowego wyrażenia z rysunku 5.4. 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 5.5 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 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.

5.6 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 — szablonu `plugin_manager`. Klasy menadżera wtyczek umożliwiają programiście-użytkownikowi ładowanie modułów z wcześniej zadeklarowanym interfejsem niezależnie od platformy systemowej na której uruchamiany jest program³.

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.

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

Sama wtyczka natomiast rejestruje w instancji przekazanego menadżera, fabryki klas w niej zawartych. Dzięki temu, obiekty wtyczek nie są tworzone do czasu gdy są

³Same moduły muszą być skompilowane pod platformę na której program ma być uruchamiany.

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 w jakiej ilości 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 menadżer 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 integrację 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 klasy⁴. Konstruktor interfejsu wymaga natomiast przekazania identyfikatora konkretnej implementacji co ułatwia identyfikację instancji wtyczek.

Listing 5.3 `test_module.hpp`

```
#include <plugin_manager.hpp>

class MyPlugin : public mru::plugin<MyPlugin> {
public:
    PLUGIN_INTERFACE("MyPlugin")
    MyPlugin(const mru::name_type &a_name)
```

⁴Najwygodniejszym rozwiązaniem zdaje się być stosowanie nazwy typu (klasy) jako nazwy interfejsu.

```

        : 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`, która podobnie jak metoda do identyfikacji interfejsu zwykle zwraca nazwę typu klasy.

Warto w tym miejscu zauważyć że każda biblioteka dynamiczna 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 5.4 `test_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 5.5 `main.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;
}

```

5.7 Typy modułów w MRU

Aplikacja obsługuje trzy interfejsy wtyczek:

- 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.

5.8 Moduły UI

Wtyczki interfejsu użytkownika pozwalają użytkownikowi końcowemu na interakcję z programem. Pojedynczy proces aplikacji może posiadać aktywny tylko jeden moduł UI. Wtyczki interfejsu odpowiadają za całkowitą komunikację między użytkownikiem i rdzeniem aplikacji — `MruCore`; to one udostępniają większość funkcjonalności narzędzia, a także informują użytkownika o jego stanie.

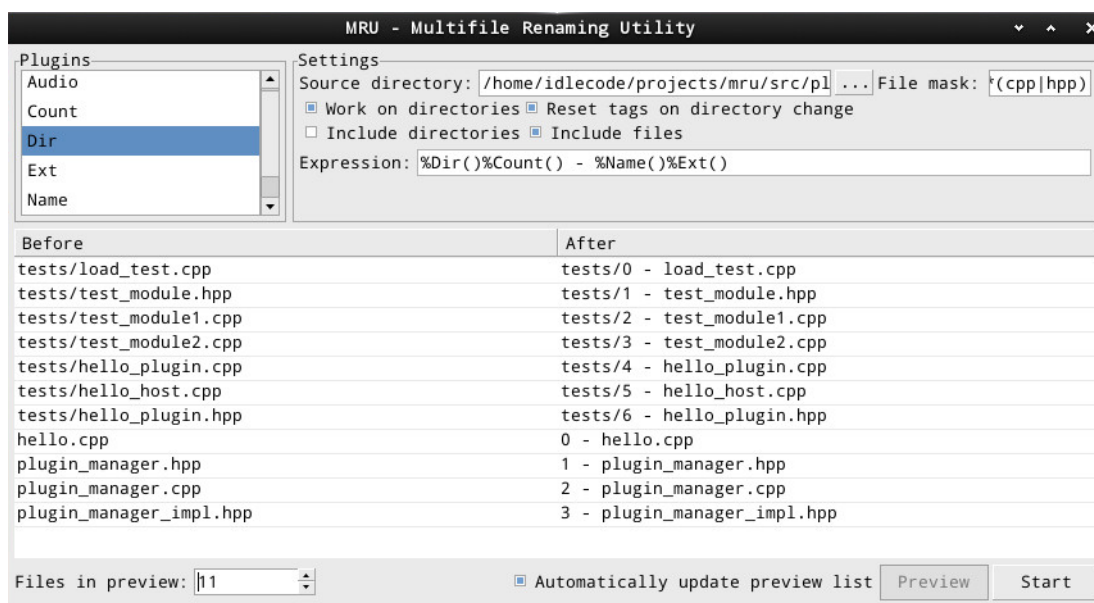
Jako że funkcjonalność aplikacji jest w dużej mierze determinowana przez klasę rdzenia (`MruCore`), interfejs `UiPlugin` nie posiada z góry zdefiniowanych metod jak inne wtyczki. Jedyna metoda w nim zawarta — `start` — pozwala na reinterpretację linii poleceń i służy do przekazania kontroli nad programem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

5.8.1 `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.



Rysunek 5.6: Okno aplikacji MRU — wtyczka wxWidgetsUi

5.9 Moduły output

Wtyczki wyjścia są warstwą abstrakcji pomiędzy systemem operacyjnym i jego drzewem katalogów, a rdzeniem aplikacji. Każde żądanie zmiany nazwy zostaje przekazane do modułu wyjścia. Kontrolują one również poprawność wygenerowanych nazw, a także zapewniają ich unikalność.

5.9.1 GenericBoost

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

5.10 Moduły metatagów

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

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

5.10.1 Count

Metatag 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 5.1: Zestaw argumentów inicjalizacyjnych dla metatagu Count

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

5.10.2 Audio

Metatag Audio został oparty na bibliotece `TagLib` i pozwala na ekstrakcję metadanych z wielu plików multimedialnych. 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 5.2: Zestaw argumentów inicjalizacyjnych dla metatagu **Audio**

5.10.3 Name

Metatag **Name** reprezentuje źródłową nazwę pliku bez rozszerzenia oraz katalogu zawierającego. Może być używany gdy zachodzi potrzeba dodania prefiksu lub sufiksu do istniejącej nazwy pliku. Dzięki składni metawyrażeń możliwa jest również modyfikacja źródłowego identyfikatora na przykład za pomocą tagu **TextCase**.

5.10.4 Ext

Metatag **Ext** reprezentuje rozszerzenie pliku. Jako że modyfikacja rozszerzenia może prowadzić to błędnej interpretacji pliku przez inne programy, często jest niepożądana. Metatag **Ext** użyty na końcu metawyrażenia stosowany jest w celu zachowania rozszerzenia pliku.

5.10.5 Dir

Dir jest metatagiem reprezentującym samą ścieżkę (listę katalogów) w której znajduje się przetwarzany plik. Ścieżka zwracana przez **Dir** jest ścieżką pośrednią w stosunku do katalogu przetwarzania wybranego przez użytkownika — dla plików znajdujących się bezpośrednio w wybranym katalogu, metatag **Dir** zwraca pustą ścieżkę.

5.10.6 TextCase

Metatag 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.

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 5.3: Zestaw argumentów inicjalizacyjnych dla metatagu TextCase

5.11 Testy

W celu przetestowania aplikacji ze strony <http://www.jamendo.com/pl/track/986557/pulsar> zostało pobrane archiwum zawierające album muzyczny z plikami w formacie *MPEG-1/MPEG-2 Audio Layer 3* — popularnie zwanym 'MP3' — które to zawierają poprawnie sformatowane tagi ID3 w wersji 2.4.0.

Celem testu było sprawdzenia parsera metawyrażeń, filtra regex, a także ogólnego działania programu. Na rysunku 5.7 widać stan katalogu przed uruchomieniem programu.

Do testów zostało wybrane metawyrażenie wykorzystujące metatagi Audio, TextCase oraz Ext. Celem użytego metawyrażenia jest kompozycja nazwy składającej się z nazwy albumu do którego utwór należy (metatag %Audio(album)) pisanej dużymi literami, po której następuje tytuł (%Audio(title)) oraz rok wydania utworu w nawiasach kwadratowych ([%Audio(year)]). Wykorzystany na końcu metatag Ext, zapewnia że pliki wynikowe będą posiadały takie samo rozszerzenia jak w nazwa źródłowa.

Dodatkowo w celu zmniejszenia ilości błędów został zastosowany filtr z wyrażeniem

```
urxvt
[10:04:52][ /home/idlecode/projects/mru/src/tests/files/JCRZ ]
[idlecode]> ls
01 - JCRZ - M.O.V.E. _My Own Vintage Evolution_.mp3
02 - JCRZ - Release.mp3
03 - JCRZ - Dream In Key Minor.mp3
04 - JCRZ - The Temple.mp3
05 - JCRZ - Schrdinger_s Cat On The Piano.mp3
06 - JCRZ - Something Controls Me Completely.mp3
07 - JCRZ - The Circles Of Chaos.mp3
08 - JCRZ - From Nothing.mp3
09 - JCRZ - Pulsar.mp3
10 - JCRZ - Etc....mp3
Cover.jpg
License.txt
[10:04:55][ /home/idlecode/projects/mru/src/tests/files/JCRZ ]
[idlecode]> 
```

Rysunek 5.7: Stan plików przed zmianą

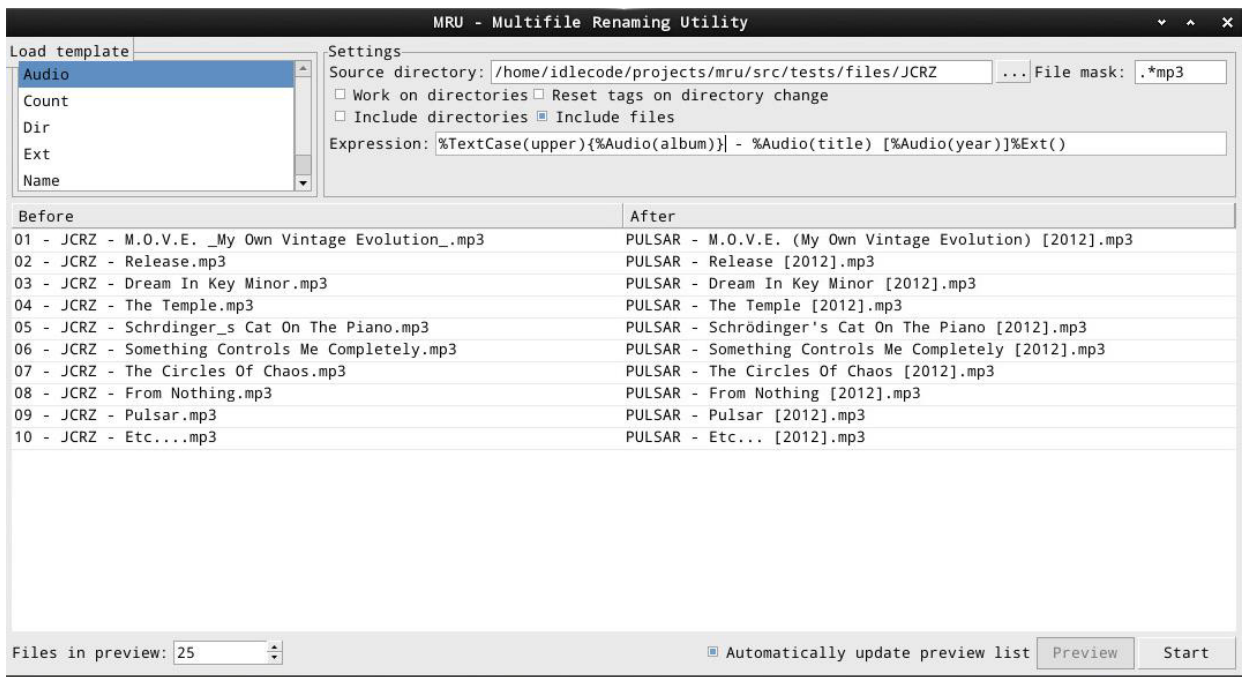
regularnym `".*mp3"`, który zapewnia że program będzie przetwarzać pliki zakończone ciągiem znaków `"mp3"`.

```
%TextCase(upper){%Audio(album)} - %Audio(title) [%Audio(year)]%Ext()
```

5.12 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.

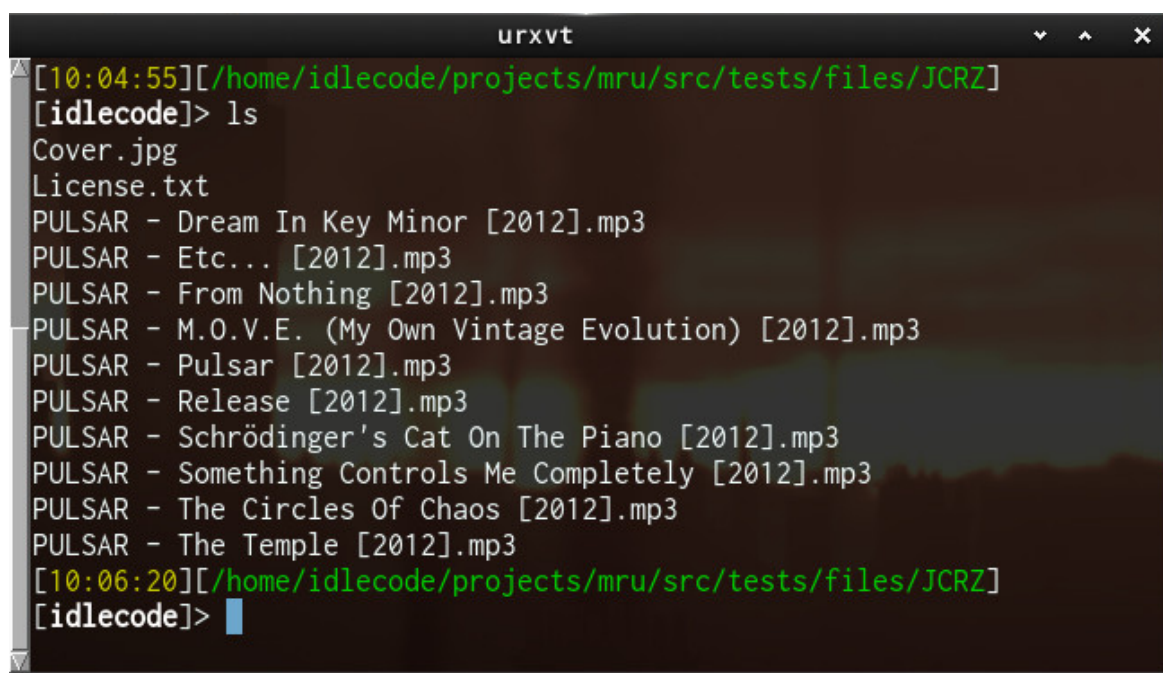


Rysunek 5.8: Konfiguracja aplikacji użyta do zmiany identyfikatorów

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 mogą znaleźć również zastosowanie przy porównywaniu plików w celu identyfikacji duplikatów.

Wtyczki wyjścia — OutputPlugin — nie mają w żaden sposób narzuconej implementacji. Nic więc nie stoi na przeszkodzie aby stworzyć wtyczkę która generuje na przykład tekstowe raporty dla istniejących plików zawierające metadane w nich zawarte.

A terminal window titled 'urxvt' with standard window controls. The prompt is '[idlecode]' and the current directory is '/home/idlecode/projects/mru/src/tests/files/JCRZ'. The user has entered 'ls' and the terminal displays a list of files: 'Cover.jpg', 'License.txt', and several MP3 files from the 'PULSAR' project, all dated '[2012]'. The prompt returns to '[idlecode]>' with a cursor.

```
[10:04:55][ /home/idlecode/projects/mru/src/tests/files/JCRZ]
[idlecode]> ls
Cover.jpg
License.txt
PULSAR - Dream In Key Minor [2012].mp3
PULSAR - Etc... [2012].mp3
PULSAR - From Nothing [2012].mp3
PULSAR - M.O.V.E. (My Own Vintage Evolution) [2012].mp3
PULSAR - Pulsar [2012].mp3
PULSAR - Release [2012].mp3
PULSAR - Schrödinger's Cat On The Piano [2012].mp3
PULSAR - Something Controls Me Completely [2012].mp3
PULSAR - The Circles Of Chaos [2012].mp3
PULSAR - The Temple [2012].mp3
[10:06:20][ /home/idlecode/projects/mru/src/tests/files/JCRZ]
[idlecode]> 
```

Rysunek 5.9: Stan plików po zmianie

Rozdział 6.

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 narzędzi 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 open-source.

Stworzona aplikacja sprostала 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 jak menadżer wtyczek) dzięki swojej uniwersalności mogą posłużyć do budowy kolejnych programów.

Bibliografia

- [1] Chuck Allison Bruce Eckel. *Thinking in C++ Tom 2*. 2004.
- [2] William Stallings2. *Organizacja i architektura systemu komputerowego*. 2004.