Design und Implementierung eines modularen Filmmetadaten Such- und Analysesystems

Als Open–Source–Software Projekt

Projektarbeit
an der Hochschule für Angewandte Wissenschaften Hof

Vorgelegt bei:
Prof. Dr. Günther Köhler
Alfons-Goppel-Platz 1
95028 Hof

Vorgelegt von:
Christoph Piechula
Breslaustr. 4
95028 Hof

Hof, 29. April 2014

Abstract

This paper demonstrates a way to build a modular movie metadata retrieval and analysis library. The main purpose of this library is to retrieve and postprocess movie metadata by user needs, making the user less dependent to a specific metadata source or metadata tool. In order to demonstrate and test the library, two command line tools are implemented. By using those command line tools a user may test plugins, but also use the library for scripting tasks. There is also a RESTful webservice, for usage in conjunction with XBMC, implemented.

Danksagung

Mein Dank gilt folgenden Personen: Herrn Prof. Dr. Günther Köhler (Projektbetreuung), Herrn Prof. Dr. Jörg Scheidt (Bereitstellung eines Raumes für den Projektzeitraum), Teegeschwendner, Cricetulus, Micrathene Whitneyi, Columbidae und natürlich auch — Alces alces.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
API	*Application Programming Interface*
CLI	*Command Line Interface*
HDTV	*High Definition Television*
LoC	*Lines of Code*
XML	*Extensible Markup Language*
URL	*Uniform Resource Locator*
ID	*Identifier*

1 | Einleitung

1.1 Motivation

Die Digitalisierung der modernen Konsumgesellschaft schreitet immer weiter voran. Vor ein paar Jahren war es noch üblich, die eigene Filmsammlung im Regal aufzubewahren. Heute wird sie häufig nur noch digital auf dem *Home–Theatre–PC*, *Smart–TV*, PC oder anderen Endgeräten digital aufgezeichnet und verwaltet. Das Aufkommen der digitalen HDTV–Sender und das große Angebot an Pay–TV–Sendern hat den Trend der letzten Jahre nochmal verstärkt. Hat man einen Spielfilm verpasst, so kann dieser bequem über einen der vielen Online–Videorecorder–Dienste nachträglich bezogen werden. Es geht sogar soweit, dass USB–Sticks (siehe [Link–1]) mit Hollywood–Spielfilmen beworben und verkauft werden.

Zeichnet man viele Filme auf oder digitalisiert seine Filmsammlung, so muss man sich mit der Pflege der inhaltsbezogenen *Metadaten* auseinandersetzen. Das sind Metadaten, wie sie auf jeder DVD–Hülle enthalten sind. Vor ein paar Jahren waren diese noch auf den DVD–Hüllen im Regal zu finden, heutzutage müssen sie nach dem Digitalisieren erst noch vom Benutzer nachträglich eingepflegt werden. Typische Metadaten bei Filmen sind Titel, Jahr, Inhaltsbeschreibung, Cover und Genre. Diese Metadaten sind bei Filmen im Unterschied zu Musik essentiell, da hierüber die Entscheidung getroffen wird, ob ein Film geschaut wird oder nicht.

Die Metadaten werden in der Regel über Onlinequellen, wie beispielsweise die Internet–Movie–Database (IMDb, siehe [Link–2]) bezogen. Die Anzahl der möglichen Onlinedienste, die Metadaten bereitstellen, ist sehr groß. Viele Abspielanwendungen wie das Windows–Media–Center oder das XBMC–Media–Center (XBMC, siehe [Link–3]) können ihre Metadaten automatisch aus dem Internet beziehen. Je nach Anwendung kann es hier im Hintergrund eine oder mehrere Bezugsquellen für Metadaten geben.

Ein Problem bei der Pflege der Metadaten im Filmbereich ist, dass es hier keinen Standard gibt, der sich durchgesetzt hat. Es gibt einerseits die Möglichkeit, bestimmte Metadaten in bestimmte Container–Formate (siehe Streaminfos in Tabelle [Link–4]) zu integrieren, andererseits werden diese in separaten Dateien oder Datenbanken der jeweiligen Abspiel-/Verwaltungssoftware gepflegt. Ein weiteres Problem ist die große Anzahl verschiedener Onlinequellen, von denen die Metadaten bezogen werden. Hier werden von Anwendung zu Anwendung unterschiedliche Quellen verwendet, die je nach Filmsammlung gut oder weniger gut geeignet sind. Die Onlinequellen unterscheiden sich stark in Qualität, Umfang und Art der angebotenen Metadaten. Zusätzlich kommt noch das Problem hinzu, dass die Metadaten je nach Quelle nur in einer bestimmten Sprache vorhanden sind.

Daraus resultierend sind über die Jahre sogenannte Movie–Metadaten–Manager entstanden, die den Benutzer bei der Pflege der Filmsammlung unterstützen sollen. Auch hier gibt es nicht das Tool der Wahl. Es gibt verschiedene Tools, die nur bestimmte Metadaten–Exportformate unterstützen, nur bestimmte Onlinequellen ansprechen können oder auch nur unter bestimmten Betriebssystemen verfügbar sind. Die unter Linux vorhandenen und getesteten Movie–Metadaten–Manager funktionieren bis auf wenige Ausnahmen unbefriedigend oder gar nicht. Die Hauptmotivation dieser Arbeit ist, diese Situation zu verbessern.

1.2 Projektziel

Das Ziel dieser Arbeit ist es, eine andere Herangehensweise im Vergleich zu den aktuell verfügbaren Tools zu zeigen und ein modulares pluginbasiertes System zu entwerfen, das die verschiedenen Metadaten–Exportformate und Metadaten–Bezugsquellen zusammenführt und über eine *einheitliche Schnittstelle* anbietet. Neben der Funktionalität der Metadatenbeschaffung soll es die Möglichkeit der Metadatenaufbereitung geben. Hierzu gehören beispielsweise das Säubern der Metadaten von unerwünschten Sonderzeichen, aber auch die automatische Extraktion von Schlüsselwörtern aus der Inhaltsbeschreibung mittels Data–Mining–Algorithmen.

Die aktuellen Tools zur Metadatenverwaltung verfolgen einen eher *monolithischen* Ansatz. Im Gegensatz dazu soll das zu entwickelnde System nach dem Baukastenprinzip erweiterbar sein und durch neue Plugins an verschiedene Anforderungen des Benutzers anpassbar sein.

Im Unterschied zu den bereits vorhandenen Tools, die hauptsächlich durch manuelle Interaktion des Benutzers gesteuert werden, soll das System auf eine automatisierte Verarbeitung ausgelegt sein. Hier liegt das Hauptaugenmerk auf der Pflege großer Filmsammlungen mit mehreren hundert Filmen.

Der modulare Aufbau und eine freie Lizenz sollen eine Weiterentwicklung durch die Community ermöglichen und zusätzlichen Spielraum für neue Ideen schaffen.

1.3 Projektlizenz

Da eine community–basierte Weiterentwicklung angestrebt wird und somit auch Verbesserungen in das Projekt zurückfließen sollen, wird das System unter der GPLv3–Lizenz (siehe [Link–5]) entwickelt. Alle erstellten Grafiken sind unter Creative–Commons–Licence (siehe [Link–6]) gestellt.

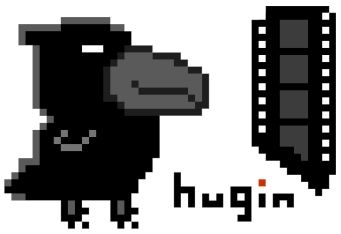

Abbildung 1.1.: Libhugin Logo, das einen Pixelraben und ein Stück Filmrolle zeigt.

1.4 Namensgebung

Um dem Projekt ein „Gesicht" zu geben und den Wiedererkennungwert zu steigern, wird das Projekt auf den Namen *libhugin* „getauft" und ein Logo entwickelt (siehe Abbildung 1.1), welches einen Raben in Pixelgrafik und ein Stück Filmrolle zeigt. Der *lib*–Präfix wurde angehängt da es sich bei dem System um eine Bibliothek (engl. Library) handelt.

Der Name Hugin kommt aus der nordischen Mythologie:

> *Hugin gehört zum altnordischen Verb huga „denken", das hierzu zu stellende Substantiv hugi „Gedanke, Sinn" ist seinerseits die Grundlage für den Namen Hugin, der mit dem altnordischen Schlussartikel –in gebildet wurde. Hugin bedeutet folglich „der Gedanke".*
>
> —http://de.wikipedia.org/wiki/Hugin_und_Munin [Link–7]

Die beiden CLI–Tools, Geri und Freki, wurden nach den beiden Wölfen die Odin begleiten benannt (siehe [Link–8]).

2 | Begriffsdefinitionen

Im Folgenden werden Fachbegriffe und Begriffe, die im Kontext von *libhugin* anders als im normalen Sprachgebrauch besetzt sind, definiert:

2.1 Fachbegriffe

Metadaten:
Metadaten sind beschreibende Daten, die Informationen über andere Objekte enthalten.

Home–Theatre–PC:
Ein auf PC–Komponenten basierendes Gerät zur Wiedergabe multimedialer Inhalte. Dieser wird oft mit sogenannter Media–Center–Software wie dem XBMC betrieben.

Smart–TV:
Bezeichnung für einen Fernseher, der eine Computerfunktion integriert hat und internetfähig ist.

XML:
Extensible Markup Language, eine Auszeichnungssprache zur baumartig strukturierten Darstellung von Daten in Form von Textdateien.

JSON:
JavaScript–Object–Notation, eine Auszeichnungssprache ähnlich wie *XML* mit dem Ziel, von Mensch und Maschine einfacher lesbar zu sein als *XML*.

Hashtabelle:
Eine spezielle Datenstruktur, bei welcher Daten jeweils einem eindeutigen Index zugeordnet sind. Der Zugriff auf die Daten erfolgt mit konstantem Aufwand.

Scraper:
Scraper werden Anwendungen genannt, die in der Lage sind Informationen aus Webseiten zu extrahieren. Ein Scraper gibt sich einer Webseite gegenüber als ein normaler Webbrowser aus. So wird eine maschinelle Verarbeitung der Informationen auf dieser Webseite möglich.

RESTful:
Eine Form einer Web–API, bei der mithilfe von Standard HTTP–Verben und speziellen, men-

schenlesbaren URLs bestimmte Aktionen getriggert werden können. So kann man sich beispielsweise mit einer GET–Operation Informationen von einem Service beschaffen.

2.2 Kontextspezifische Fachbegriffe

Plugin:
Im Kontext von *libhugin* sind Plugins kleine Module, welche zur Laufzeit austauschbar sind. Diese können von Dritten geschrieben werden, um das System an die eigenen Bedürfnisse anzupassen.

Provider:
Im Kontext von *libhugin* ist ein Provider eine Art „Vermittler" zwischen der entwickelten Bibliothek und einem Online–Webservice der Metadaten anbietet. Der Provider ist Bestandteil der Bibliothek und wird als Plugin in dieser implementiert.

Postprocessor:
Im Kontext von *libhugin* ist ein Postprocessor ein Plugin, welches für die direkte Nachbearbeitung der heruntergeladenen Metadaten verwendet wird.

Converter:
Im Kontext von *libhugin* ist ein Converter ein Plugin, das für das Konvertieren eines Ergebnisses in ein bestimmtes Metadatenformat wie beispielsweise das XBMC nfo–Format zuständig ist.

Modifier:
Im Kontext von *libhugin* ist ein Modifier ein Plugin, welches für die nachträgliche Bearbeitung von Filmmetadaten verwendet wird. Das kann beispielsweise das Ändern der Sprache der Inhaltsbeschreibung sein.

Analyzer:
Im Kontext von *libhugin* ist ein Analyzer ein Plugin, welches für die nachträgliche Analyse von Filmmetadaten verwendet wird. Dies kann beispielsweise die Erkennung der Sprache der Inhaltsbeschreibung sein.

Comparator:
Im Kontext von *libhugin* ist ein Comparator ein Plugin, welches für Vergleiche zuständig ist. Mit Hilfe dieser Pluginart soll im späteren Verlauf untersucht werden, wie gut sich Filme anhand von Metadaten vergleichen lassen und ob sich beispielsweise Film–Empfehlungen aufgrund der gewonnenen Daten aussprechen lassen. Diese Pluginart ist experimentell und nur konzeptionell in *libhugin* integriert.

3 | Stand der Metadatenpflege

Die vorgestellten Plattformen, Player und Tools zeigen nur einen Ausschnitt. Alle Plattformen, Player und Tools aufzulisten ist aufgrund der Vielfalt beziehungsweise Komplexität nicht möglich.

3.1 Metadatenarten und Quellen

3.1.1 Metadatenarten

Grundsätzlich lassen sich Film–Metadaten in zwei Kategorien einordnen. Metadaten, die das Videoformat (Auflösung, Bitrate, ...) beschreiben, und Metadaten, die den Inhalt beschreiben. Metadaten zur Beschreibung des Videoformats können je nach Container–Format direkt in die Datei eingebettet werden (siehe [Link–9]).

Inhaltsbezogene Metadaten sind Daten, die bei der Digitalisierung nachträglich gepflegt werden müssen. Typischerweise sind das Attribute wie Titel, Erscheinungsjahr, Genre, Inhaltsbeschreibung, Cover und noch einige weitere.

3.1.2 Metadatenquellen

Zum Bezug der Metadaten werden verschiedene Onlinequellen genutzt. Im Prinzip eignet sich jede Seite, die Filminformationen pflegt, als Metadatenquelle. Zu den gängigen Metadatenquellen — neben zahlreichen anderen Quellen — zählen:

- *Internet Movie Database (IMDb)*, englischsprachig
- *The Movie Database (TMDb)*, multilingual, community–gepflegt
- *Online Filmdatenbank (OFDb)*, deutschsprachig, community–gepflegt

IMDb ist mehr oder weniger der „Platzhirsch" unter den Metadatenquellen. Für viele ist Sie die Standardbezugsplattform. Über die bei der IMDb für jeden Film gepflegte *IMDb–ID* kann ein Film genau identifiziert werden. Da diese ID eindeutig ist, wird sie oft auch von anderen Onlinequellen mit erfasst und gepflegt, um auch über diese eine Suche zu ermöglichen. Leider bietet IMDb keine deutschsprachigen Daten an. Auch die grafischen Daten wie Cover und Hintergrundbilder sind hier, im Vergleich zu anderen Quellen, von schlechterer Qualität.

TMDb ist eine hauptsächlich Community gepflegte Onlinequelle. Die hier gepflegten Filme enthalten neben den Standardmetadaten auch hochauflösende Cover und Hintergrundbilder (sogenannte Backdrops und Fanart). Die Datenbank wird oft von Open–Source–Projekten wie auch dem XBMC verwendet.

OFDb ist eine im deutschen Raum bekannte Filmdatenbank, welche deutschsprachige Metadaten pflegt.

Die Metadaten, die von den jeweiligen Plattformen bezogen werden unterscheiden sich stark in Qualität, Art und Umfang. Insbesondere die Inhaltsbeschreibung ist hier sehr vielfältig — von kurz und knapp bis sehr ausführlich. Schaut man sich beispielsweise für den Film *„Per Anhalter durch die Galaxis (2005)"* die deutsche Inhaltsbeschreibung auf den vier Film–Plattformen *cinefacts.de* (siehe [Link–10]), *filmstarts.de* (siehe [Link–11]), *OFDb.de* (siehe [Link–12]) und *TMDb* (siehe [Link–13]) an, so wird man feststellen, dass jede dieser Plattformen eine andere deutsche Inhaltsbeschreibung gepflegt hat. Je nach persönlichen Präferenzen möchte man nur eine bestimmte Art von Inhaltsbeschreibung einpflegen.

3.2 Software und Metadatenformate

3.2.1 Abspielsoftware

Die Darstellung einer mit Metadaten gepflegten Filmsammlung erfolgt in den meisten Fällen über sogenannte Media–Center–Software, die für den Home–Theater–PC Betrieb im Wohnzimmer angepasst ist.

Beispiele hierfür wären das Windows–Media–Center oder auch das freie XBMC (siehe Abbildung 3.1), welches in letzter Zeit noch einmal durch den *Raspberry Pi* (siehe [Link–14]) Bekanntschaft erlangt hat. Neben den PC–basierten Lösungen gibt es hier auch zahlreiche Standalone–Lösungen wie beispielsweise Popcorn–Hour (siehe [Link–15]).

Die Media–Center–Software kann ihre Metadaten in der Regel je nach Applikation von einer oder mehreren Onlinequellen beziehen. Sie bietet dem Benutzer jedoch in der Regel nicht die Möglichkeit, Korrekturen durchzuführen und ist somit nur bedingt zum Pflegen von großen Filmsammlungen geeignet.

3.2.2 Movie–Metadaten–Manager

Neben den Media–Center–Lösungen gibt es spezielle Tools für die Pflege und Korrektur von Film–Metadaten, sogenannte *Movie–Metadaten–Manager*. Ein Movie–Management–Tool, welches es unter

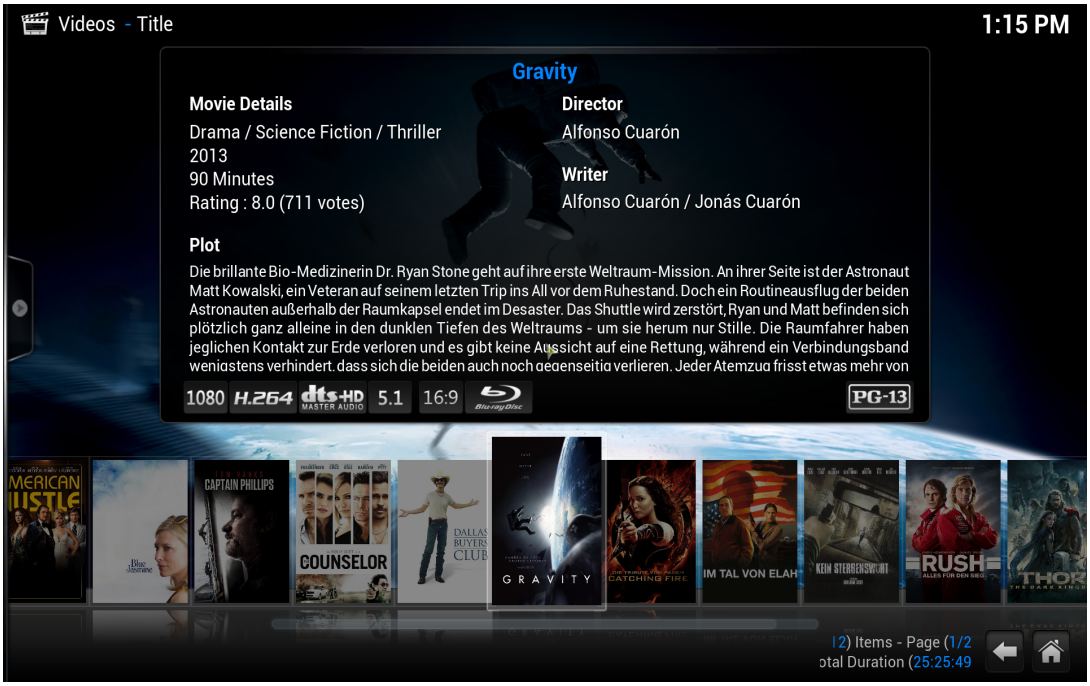

Abbildung 3.1.: Screenshot einer im XBMC gepflegten Filmesammlung.

unixoden Betriebssystemen gibt, ist beispielsweise MediaElch (siehe Abbildung 3.2, siehe [Link–16]). Hier gibt es unter Linux noch weitere Tools (siehe [Link–17]).

Diese Programme beziehen ihre Metadaten auf die gleiche Art und Weise wie auch die Media–Center–Lösungen. Die Management–Tools bieten dem Benutzer zusätzlich die Möglichkeit, fehlerhafte Metadaten manuell zu korrigieren oder Metadaten zu ergänzen.

Da die Programme nur für die Pflege von Metadaten gedacht sind, gibt es hier immer Import- und Exportschnittstellen, welche wiederum auf bestimmte Formate (siehe Metadatenformate, 3.2.3) begrenzt sind.

Bestimmte Onlinequellen wie die IMDb bieten ihre Metadaten nur in englischer Sprache an. Möchte man eine deutsche Inhaltsbeschreibung haben, so muss man auf eine Onlinequelle zugreifen, die diese in deutscher Sprache pflegt. Je nach Anwendung wird dies aber nicht immer unterstützt.

3.2.3 Metadatenformate

Im Gegensatz zum Musikbereich hat sich bei der Pflege von Metadaten im Filmbereich kein Standard durchgesetzt. Hier wird je nach Abspiel- oder Verwaltungssoftware jeweils ein anderes Format verwendet.

Das XBMC speichert seine Metadaten beispielsweise intern in einer Datenbank und schreibt diese beim Exportieren in *XML*–Dateien, das sogenannte *nfo*–Format, heraus (siehe [Link–18]). Nutzt man eine andere Abspielsoftware wie das Windows–Media–Center, so werden die Metadaten im *dvdxml*–

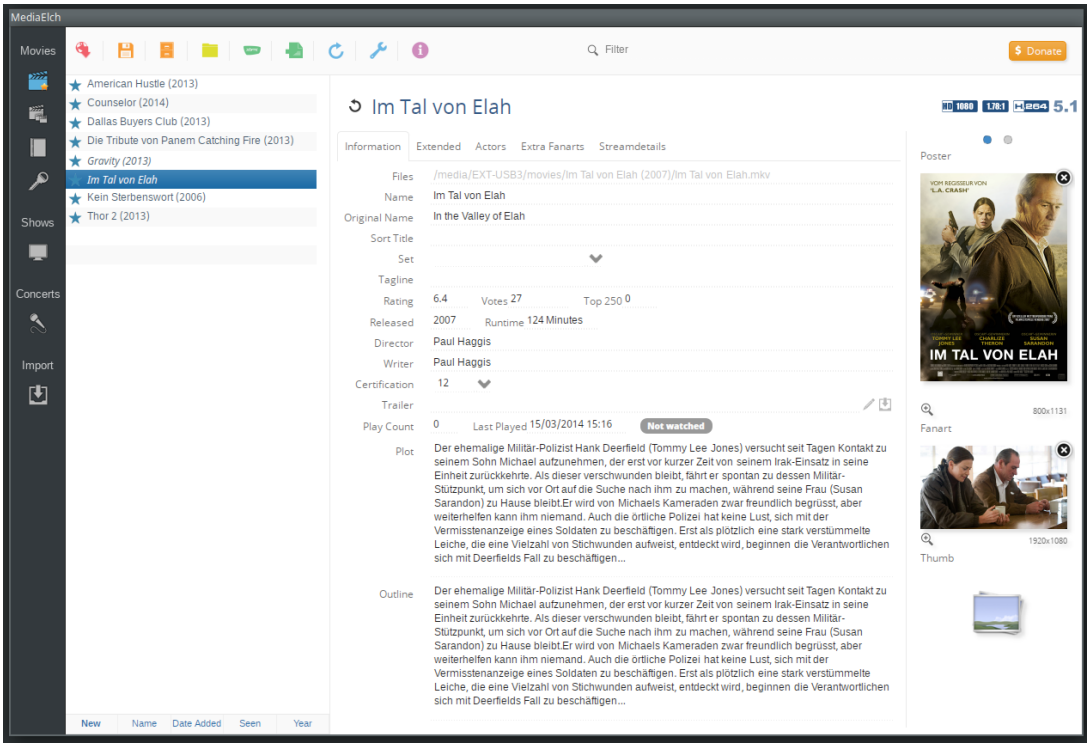

Abbildung 3.2.: Screenshot vom Movie–Metadaten–Manager MediaElch.

Format, auch ein *XML* basiertes Format, abgespeichert (siehe [Link–19]). Hier gibt es noch zahlreiche andere Formate, auch bei den Movie–Metadaten–Managern, auf die nicht weiter eingegangen wird.

Dieser Umstand erschwert das Pflegen der Film–Metadaten zusätzlich. Für die beiden genannten Formate bieten Movie–Metadaten–Manager häufig Import- und Exportmöglichkeiten an. Jedoch können andere Player oder auch Standalone–Lösungen hier wiederum ganz andere Formate verwenden, die von der Metadaten–Pflegesoftware nicht unterstützt werden.

3.3 Probleme bei der Metadatenpflege

3.3.1 Unbekannte und ausländische Filme

In den meisten Fällen werden bei den oben beispielhaft genannten Anwendungen die richtigen Metadaten für die *bekannten* Hollywood Filme gefunden. Hat man aber eine Filmsammlung, die viele *Independent Filme* [1] oder nicht–amerikanische Verfilmungen enthält, so kommt es immer wieder zu Problemen. Die grundlegenden Probleme hier sind, dass ein Film entweder gar nicht gefunden wird, nur ein Teil der Metadaten gefunden wird oder diese eben nur in einer bestimmten Sprache bezogen werden können.

[1] Bezeichnung für Filme, die von Produktionsfirmen finanziert werden, welche nicht zu den großen US Studios gehören.

Werden Metadaten für einen bestimmten Film über die standardmäßig eingestellte Onlinequelle nicht gefunden, so gibt es häufig die Möglichkeit eine andere Onlinequelle zu verwenden. Hierbei entstehen jedoch neue Probleme, die nun folgend betrachtet werden.

3.3.2 Redundante Metadaten

Grundlegende Problematik: Redundanzen treten in der Regel auf, wenn bei einer Filmsammlung die Daten aus unterschiedlichen Quellen stammen. Damit ist gemeint, dass beispielsweise das Genre auf unterschiedlichen Plattformen unter einem anderen Namen gepflegt ist. Beim Herunterladen von Metadaten aus mehreren Quellen, wird beispielsweise das Genre „SciFi" von einer Onlinequelle und das Genre „Science–Fiction" von einer andere Quelle bezogen. Durch diesen Umstand ist das eigentlich eindeutige Genre Science Fiction in diesem Fall zweimal in der lokalen Datenbank vorhanden. Neben dem Genre sind auch weitere Attribute von der Redundanz–Problematik betroffen, jedoch ist das Genre, neben der Inhaltsbeschreibung, nach Meinung des Autors, eins der wichtigsten Attribute, da es maßgeblich in die Entscheidung der Filmauswahl einfließt.

Folgende Punkte führen konkret im unten genannten Beispiel (siehe Praxisbeispiel für Dateninhomogenität, 3.3.4) zu Redundanzen:

Schreibweise des Genres: Die Schreibweise der gepflegten Genres unterscheidet sich (siehe Abbildung 3.6). Hier ist bei TMDb das Genre „Science Fiction" und bei OFDb „Science–Fiction" gepflegt.

Internationalisierung: Je nach Onlinequelle ist das Genre in einer unterschiedlichen Sprache gepflegt. IMDb listet hier das Genre „Comedy" (siehe Abbildung 3.4), TMDb die deutsche Bezeichnung „Komödie".

3.3.3 Divergente Metadaten

Divergente Genres: Die OFDb–Quelle liefert für den Film *Feuchtgebiete (2013)* das Genre *Erotik*, dieses Genre existiert bei IMDb (siehe [Link–20]) und bei TMDb (siehe [Link–21]) gar nicht.

3.3.4 Praxisbeispiel für Dateninhomogenität

Um das Problem zu veranschaulichen betrachten wir parallel zur oben genannten Problematik Auszüge von Metadaten der drei Onlinequellen *IMDb*, *TMDb* und *OFDb*.

Ausgehend von der Annahme, dass die Inhaltsbeschreibung (engl. Plot) und das Genre zu den *wichtigsten* Kriterien bei der Filmauswahl gehören und diese somit gepflegt sein müssen, werden diese nachfolgend explizit betrachtet.

In unserem Beispiel befinden sich folgende vier Filme in der Filmsammlung, die mit Metadaten versorgt werden sollen:

1. „*After.Life (2010)*", US–amerikanischer Spielfilm
2. „*Feuchtgebiete (2013)*", deutsche Romanverfilmung
3. „*Nymphomaniac (2013)*", europäischer Spielfilm
4. „*RoboCop (2014)*", US-amerikanischer Spielfilm

Die Inhaltsbeschreibung ist in der Regel problemlos austauschbar, jedoch unterscheidet sie sich auch je nach Quelle in Formatierung, Ausführlichkeit und Sprachstil. Nicht alle Inhaltsbeschreibungen haben beispielsweise hinter dem Rollennamen immer den Namen des Schauspielers in Klammern. Sollen die Metadaten in deutscher Sprache gepflegt werden, so fällt IMDb raus, da diese Onlinequelle nur Metadaten in englischer Sprache anbietet. Die Onlinequelle wird aber bezüglich des Genrevergleichs mit in die Tabellen aufgenommen.

After.Life (2010): Die Daten bei TMDb werden in verschiedenen Sprachen gepflegt und sind in der Regel *qualitativ hochwertig*. Der erste Film wurde bei TMDb gut eingepflegt, die Inhaltsbeschreibung ist deutschsprachig, das Genre *feingranular* gepflegt. Des Weiteren bietet TMDb hochauflösende grafische Metadaten (Cover, Hintergrundbilder). Bei OFDb ist das Genre „Mystery" nicht gepflegt und zudem gibt es nur ein niedrig auflösendes Cover und keine Hintergrundbilder (siehe Abbildung 3.3).

Quelle	*IMDb*	*TMDb*	*OFDb*
Plot	englischsprachig	deutschsprachig	deutschsprachig
Genre	Drama, Horror, Mystery	Drama, Horror, Mystery, Thriller	Drama, Horror, Thriller

Abbildung 3.3.: Übersicht Metadatenquellen für den Film After.Life (2010)

Zusammenfassung zum Genre: austauschbar, unterschiedlich feingranular gepflegt

Feuchtgebiete (2013): Der zweite Film ist bei TMDb und OFDb gut gepflegt. Jedoch fällt auf, dass das gepflegte Genre bei diesen beiden Onlinequellen keine Schnittmenge aufweist. Beim Betrachten des Wikipedia–Artikels (siehe [Link–22]) zum Film wird klar, dass das bei OFDb gepflegte Genre auch seine Daseinsberechtigung hat.

Quelle	*IMDb*	*TMDb*	*OFDb*
Plot	englischsprachig	deutschsprachig	deutschsprachig
Genre	Drama, Comedy	Drama, Komödie	Erotik

Abbildung 3.4.: Übersicht Metadatenquellen für den Film Feuchtgebiete (2013)

Zusammenfassung zum Genre: divergent, Problem der Internationalisierung

Nymphomaniac (2013): Hier ist bei TMDb die Inhaltsbeschreibung in Deutsch nicht vorhanden. Der Film ist im Vergleich zu Hollywood–Blockbuster in Deutsch relativ schlecht gepflegt. Bei OFDb ist wie auch beim ersten Film, eine deutschsprachige Inhaltsangabe vorhanden. Zur großen Überraschung ist hier das Genre im Vergleich zu den beiden anderen Onlinequellen feingranularer gepflegt — was laut Wikipedia (siehe [Link–23]) den Filminhalt besser widerspiegelt (siehe Abbildung 3.5).

Quelle	*IMDb*	*TMDb*	*OFDb*
Plot	englischsprachig	englischsprachig	deutschsprachig
Genre	Drama	Drama	Drama, Erotik, Sex

Abbildung 3.5.: Übersicht Metadatenquellen für den Film Nymphomaniac (2013)

Zusammenfassung zum Genre: divergent, unterschiedlich feingranular gepflegt

RoboCop (2014): Der vierte Film, eine Hollywood Remake–Produktion ist hier bei allen drei Anbietern sehr gut gepflegt (siehe Abbildung 3.6).

Quelle	*IMDb*	*TMDb*	*OFDb*
Plot	englischsprachig	deutschsprachig	deutschsprachig
Genre	Action, Crime, Sci–Fi	Action, Science Fiction, Krimi	Action, Krimi, Science–Fiction, Thriller

Abbildung 3.6.: Übersicht Metadatenquellen für den Film RoboCop (2014)

Zusammenfassung zum Genre: unterschiedliche Schreibweise, divergent, Problem der Internationalisierung, unterschiedlich feingranular gepflegt

Beim Bezug von Metadaten der vier Filme wird deutlich, welche Probleme bei der Beschaffung dieser entstehen können. Diese Probleme werden beim *„aktuellen Stand der Technik"* durch den Benutzer mühsam manuell behoben. Bei kleinen Filmsammlungen ist der Aufwand der manuellen Nachpflege noch vertretbar, nicht jedoch bei *größeren* Sammlungen von mehreren hundert Filmen.

3.3.5 Auswirkungen

Abspielsoftware wie das XBMC erlaubt es dem Benutzer, die Filme nach Genre zu gruppieren und zu filtern. Durch dieses *Feature* kann der Benutzer einen Film nach seinen Vorlieben aussuchen. Durch die Redundanzen ist eine eindeutige Gruppierung nicht mehr möglich. Die Folge ist ein ungeordneter Zustand.

3.4 Probleme bei der Metadatensuche

3.4.1 Grundlegende Probleme

Viele Metadaten–Tools erwarten exakte Suchbegriffe. Falsch geschriebene Filme wie „The Marix" oder „Sin Sity" werden oft nicht gefunden (siehe Abbildung 3.7).

Die Suche nach der *IMDb–ID* ist bei den getesteten Tools häufig nicht möglich, obwohl diese von manchen Onlineanbietern unterstützt wird (siehe Abbildung 3.7).

3.4.2 Probleme bei Movie–Metadaten–Managern

Es wurden neben der Abspielsoftware XBMC und dem Movie–Metadaten–Manager MediaElch, die bereits genannten Movie–Metadaten–Manager (siehe [Link–17]) *GCstar*, *vMovieDB*, *Griffith* und *Tellico* betrachtet. Die Resultate hier waren eher ernüchternd (siehe Abbildung 3.7). Bei den beiden Media–Managern GCstar und vMovieDB hat die Metadatensuche nicht funktioniert, hier wurde nichts gefunden. Das Verhalten wurde auf zwei Systemen nachgeprüft. Beim XBMC wurden die Plugins für die Onlinequellen TMDb und Videobuster getestet. Für die Unschärfesuche wurde nach „*Sin Sity*" und nach „*The Marix*" gesucht.

Software	*XBMC*	*MediaElch*	*Tellico*
IMDb–ID Suche	×	nur über IMDb u. TMDb	×
Unschärfesuche	×	×	nur IMDb, teilweise
Onlinequellen	verschiedene (plugin)	verschiedene (6)	wenige (3)
Metadatenformate	×	nur XBMC	×
Datenkorrektur	×	ja, manuell	ja, manuell
Bemerkungen	pluginbasierte Scraper	Onlinequellen kombinierbar	×
Typ	Medien–Player	Metadaten–Manager	Metadaten–Manager

Abbildung 3.7.: Übersicht Movie–Metadaten–Manager und ihre Funktionalität

Die nicht funktionierenden Movie–Manager *GCstar* und *vMovieDB* wurde nicht mit aufgenommen. Das Tool Griffith wurde auch aus der Tabelle genommen, da hier von den 40 Onlinequellen nur einzelne Quellen funktioniert haben — IMDb hat auch nicht funktioniert.

3.5 Anforderungen an das Projekt

Viele der genannten Schwierigkeiten lassen sich aufgrund ihrer Natur und der aktuellen Kombination aus Abspielsoftware und Movie–Metadaten–Manager nicht oder nur mit manuellen Eingriff durch den Benutzer beheben. Bei großen Filmsammlungen mehrerer hundert Filme ist dies jedoch mit keinem vernünftigen Aufwand umsetzbar.

Es soll *kein neuer* Movie–Metadaten–Manager entwickelt werden. Die Idee ist es, dem Entwickler beziehungsweise Endbenutzer einen *modularen Werkzeugbaukasten* in Form einer pluginbasierten Bibliothek über eine einheitliche Schnittstelle bereitzustellen, welcher an die persönlichen Bedürfnisse anpassbar ist.

Des Weiteren soll die zusätzliche Funktionalität der Datenanalyse, beispielsweise basierend auf Data–Mining–Algorithmik, möglich sein. Das Hauptaugenmerk des Systems liegt, im Gegensatz zu den bisherigen Movie–Metadaten–Managern, auf der *automatisierten* Verarbeitung großer Datenmengen.

4 | Softwarespezifikation

4.1 Anforderungen an die Bibliothek

Die Anforderungen werden aus den Schwierigkeiten der momentan vorhandenen Lösungen abgeleitet. Gute Ideen werden hier übernommen. Der konzeptionelle Entwurf der Software ist Bestandteil der Projektarbeit, die Internas und Algorithmik werden in der Bachelorarbeit behandelt.

4.1.1 Anforderungen an die Datenbeschaffung

Die Erweiterbarkeit soll durch Schreiben von Plugins erreicht werden. Folgende Pluginarten sollen bei der Datenbeschaffung umgesetzt werden:

Provider–Plugins: Die Onlinequellen, die verwendet werden, sollen austauschbar sein. Der Benutzer hat die Möglichkeit, durch Schreiben eines Plugins seine bevorzugte Onlinequelle als sogenanntes Provider–Plugin zu implementieren. Dieses Grundprinzip wird bereits bei der freien Musik–Metadatensuchmaschine *libglyr* (siehe [Link–24]) sowie im Ansatz beim XBMC (siehe [Link–25]) verwendet.

Um nicht direkt einen „Standardprovider" festlegen zu müssen, werden bei den Providern Prioritäten von 0–100 vergeben. Provider mit höheren Prioritäten werden beim Verarbeiten der Suchergebnisse bevorzugt.

Postprocessor–Plugins: Die Möglichkeit der Datenaufbereitung beim Herunterladen von Metadaten soll erweiterbar sein. Der Benutzer hat die Möglichkeit das Postprocessor–System durch Schreiben eines Plugins zu erweitern.

Converter–Plugins: Die Exportformate, die für die Speicherung der Metadaten verwendet werden, lassen sich vom Benutzer durch Schreiben eines Plugins erweitern.

Suche von Metadaten: Die Suche von Metadaten soll sich für das Projekt auf Film–Metadaten und Personen–Metadaten beschränken. Für TV–Serien–Metadaten soll jedoch auch eine Schnittstelle geboten werden.

Die Film–Metadatensuche soll feingranular konfigurierbar sein, das heißt die zu verwendenden Onlinequellen, die Anzahl der Ergebnisse und die *Art* der Metadaten soll bei einer Suchanfrage einstellbar sein. Die Unterscheidung der *Art* soll sich auf textuelle und grafische Metadaten beschränken.

Eine onlinequellen–übergreifende Suche über die IMDb–ID, welche exakte Ergebnisse liefert, ist wünschenswert.

Beim Suchverhalten über mehrere Onlinequellen soll es zwei verschiedene „Suchstrategien" geben. Durch diese Suchstrategien soll dem Benutzer die Kontrolle darüber gegeben werden, ob er möglichst genaue Ergebnisse von unterschiedlichen Providern erhalten möchte, oder ob er verschiedene Ergebnisse eines bevorzugten Providers wünscht.

Bevor die Suchergebnisse an den Benutzer zurückgegeben werden, werden diese nach Provider–Priorität gruppiert. Die gruppierten Ergebnisse je Provider werden nach Übereinstimmung mit der Suche sortiert.

Nun sollen die zwei Strategien zum Einsatz kommen, nach welcher die Ergebnisse zurückgegeben werden. Bei der *flat*–Strategie werden aus jeder Gruppe jeweils zuerst die höchstpriorisierten Ergebnisse ausgewählt bis das gewünschte Ergebnislimit erreicht ist. Bei der *deep*–Strategie wird zuerst der Provider mit der höchsten Priorität ausgeschöpft, im Anschluss der nächstniedrigere Provider.

Mit folgendem Beispiel sollen die zwei Strategien des Suchverhaltens anschaulich erläutert werden.

Onlinequelle	*A*	*B*	*C*
größte Übereinstimmung	Sin (2003)	Sin (2003)	Sin (2003)
↑	Sin Nombre (2009)	Sin City (2005)	Sin City (2005)
↓	Original Sin (2001)		Sin Nombre (2009)
kleinste Übereinstimmung			Original Sin (2001)

Abbildung 4.1.: Abbildung zeigt Metadatenanbieter (A, B, C) und die jeweils gelieferten Ergebnisse pro Anbieter.

Die Tabelle 4.1 zeigt die Suchanfrage nach dem Film „Sin" mit der Begrenzung auf vier Ergebnisse. Jedem Provider (A, B und C) ist intern eine *Priorität* zugeordnet. Die Provider sind nach *Priorität* sortiert. A hat die höchste Priorität. Die Ergebnisse pro Provider sind nach der Übereinstimmung mit dem Suchstring „Sin" sortiert.

Folgende Ergebnisse werden bei jeweiliger Strategie und der Begrenzung auf vier Ergebnisse an den Aufrufer gegeben:

- *flat*: Sin (2003) A, Sin (2003) B, Sin (2003) C, Sin Nombre (2009) A
- *deep*: Sin (2003) A, Sin Nombre (2009) A, Original Sin (2001) A, Sin (2003) B

Unschärfesuche: Der Benutzer soll auch Ergebnisse erhalten, wenn im Suchstring Tippfehler enthalten sind. Der Suchstring „The Marix" soll *metadatenanbieterübergreifend* den Film „*The Matrix (1999)*" liefern. Eine provider–übergreifende Suche wäre hier wünschenswert.

IMDb–ID Suche: Die Suche nach Filmen über die *IMDb–ID* soll möglich sein. Eine provider–übergreifende Suche wäre hier wünschenswert.

Genrenormalisierung: Um Redundanzen zu vermeiden, soll eine Genrenormalisierung implementiert werden. Hierdurch soll es möglich, sein Genre–Informationen von mehreren Providern zusammenzuführen oder zwischen den Providern austauschbar zu machen.

4.1.2 Anforderungen an die Datenanalyse

Die Analyse von Metadaten soll auf bereits existierende Metadaten anwendbar sein, mit dem Ziel die Qualität dieser zu verbessern. Hier soll es neben der reinen Analyse die Möglichkeit der Modifikation von Metadaten geben. Ein weiterer experimenteller Teil soll die Vergleichbarkeit von Metadaten für statistische Zwecke ermöglichen.

Aufgrund der genannten Anforderungen sollen folgende unterschiedliche *Pluginarten* umgesetzt werden:

Modifier–Plugins: Über diese Art von Plugins lassen sich die Metadaten direkt modifizieren. Ein Beispiel hierfür wäre das Entfernen von unerwünschten Sonderzeichen aus der Inhaltsbeschreibung.

Analyzer–Plugins: Diese Art von Plugins erlaubt es dem Benutzer die vorliegenden Metadaten zu analysieren, um neue Erkenntnisse zu gewinnen oder Defizite zu identifizieren. Ein Beispiel hierfür wäre die Erkennung der verwendeten Sprache der Inhaltsbeschreibung.

Comparator–Plugins: Diese Art von Plugins ist experimentell. Sie ist für statistische Auswertungen bezüglich der Vergleichbarkeit von Filmen anhand der Metadaten gedacht. Mit den entwickelten Plugins soll untersucht werden, ob und wie gut sich Filme anhand von Metadaten vergleichen lassen, um so in Zukunft neben der bereits erwähnten Funktionalität zusätzlich noch Empfehlungen für andere Filme aussprechen zu können.

4.1.3 Allgemeine Anforderungen an die Bibliothek

Asynchrone Bibliothek: Die Bibliothek soll eine asynchrone Ausführung von Suchanfragen implementieren. Das Herunterladen von Metadaten verschiedener Metadatenanbieter soll parallel geschehen, um die Wartezeit der Suchanfrage zu reduzieren.

Lokaler Zwischenspeicher (Cache): Es soll ein lokaler Cache implementiert werden, um valide Ergebnisse der Suchanfragen zu puffern um so die Geschwindigkeit zu erhöhen und das Netzwerk beziehungsweise die Onlinequellen zu entlasten. Manche Onlinequellen forcieren eine Volumenbegrenzung, welche man durch den Zwischenspeicher abmildern kann.

Implementierung eines kommandobasierten Frontends: Dieses soll sowohl zum Testen der Bibliothek entwickelt als auch für Demonstrationszwecke fungieren und für *Scripting–Tasks* geeignet sein.

Grundlegende Konfiguration des Download–Managers: Für das Herunterladen der Metadaten sollen die folgenden Parameter konfigurierbar sein:

- User–Agent
- Cache–Pfad
- Timeout in Sekunden
- Anzahl paralleler Download–Threads (paralleles Herunterladen)
- Anzahl der verwendeten Job–Threads (parallele Suchanfragen)

Konfigurationsmöglichkeiten für eine Suchanfrage: Folgende Parameter sollen bei einer Suchanfrage konfigurierbar sein:

- Providerart (Film, Person)
- Filmtitel, Jahr, *IMDb–ID* oder Personenname (je nach Providerart)
- Sprache in der Metadaten gesucht werden sollen (abhängig von Onlinequelle)
- Cache verwenden (ja/nein)
- Anzahl der maximalen Downloadversuche
- Anzahl der maximalen gewünschten Suchergebnisse
- Suchstrategie (*deep/flat*)
- Zu verwendende Metadatenanbieter
- Unschärfesuche (ja/nein)
- Provider übergreifende IMDb–ID–Suche (ja/nein)
- Metadatenart (textuelle Daten, grafische Daten)

4.1.4 Optionale Anforderungen

Die Bibliothek soll in ein bestehendes Open–Source–Projekt integriert werden. Hier wäre beispielsweise die Integration als Plugin in das XBMC denkbar.

4.1.5 Nicht–Anforderungen

Nicht Film–Metadaten: Die Suche und Analyse von Musikmetadaten oder anderen Metadatentypen ist nicht Bestandteil des Projekts.

Movie–Metadaten–Manager: Die Implementierung eines *neuen* Movie–Metadaten–Managers ist nicht Bestandteil des Projekts.

5 | Entwurf

Im Folgenden wird der systematische Entwurf der Software aufgezeigt. Die verwendeten Algorithmen, Probleme, sowie Möglichkeiten der technischen Umsetzung werden in der Bachelorarbeit genauer beleuchtet und diskutiert.

5.1 Grundüberlegungen

Metadaten werden über verschiedene Onlinequellen bezogen. Hier wird grundsätzlich zwischen Onlinequellen mit API und ohne API unterschieden. Onlinequellen mit API bieten dem Entwickler direkt eine Schnittstelle, über welche die interne Datenbank des Metadatenanbieters abgefragt werden kann. Die Onlinequellen mit API unterteilen sich in die zwei Technologien RESTful (vgl. [1]) und SOAP (vgl. [3]).

Folgende Shellsitzung demonstriert einen Zugriff mit dem Webtransfer–Tool *cURL* (siehe [Link–26]) auf die API der OMDb–Onlinequelle (siehe [Link–27]), es wird nach dem Film „The Matrix" gesucht (Ausgabe gekürzt):

```
$ curl http://www.omdbapi.com/?t=The+Matrix
{"Title":"The Matrix",
"Year":"1999",
"Runtime":"136 min",
"Genre":"Action, Sci--Fi",
"Director":"Andy Wachowski, Lana Wachowski",
"Actors":"Keanu Reeves, Laurence Fishburne, Carrie--Anne Moss, Hugo Weaving",
"Plot":"A computer hacker learns from mysterious [...]",
"Country":"USA, Australia",
"Poster":"http://ia.media-imdb.com/images/M/SX300.jpg",
"imdbRating":"8.7",
"imdbID":"tt0133093",
[...]
"Type":"movie",
"Response":"True"
}
```

Die *Such–URL* bekommt den Titel als Query–Parameter (`t=The+Matrix`) übergeben, zurück kommt ein in *JSON* formatiertes Response. Dieses kann nun vom Aufrufer der API beliebig verarbeitet werden.

Bietet eine Webseite wie beispielsweise *filmstarts.de* (siehe [Link–28]) keine API an, so muss der „normale Weg" wie über den Webbrowser erfolgen. Hierzu muss man herausfinden, aus welchen Parametern sich die *Such–URL* zusammensetzt, die im Hintergrund an den Webserver geschickt wird, sobald die Suchanfrage losgeschickt wird. Bei *filmstarts.de* setzt sich die *Such*–URL wie folgt zusammen:

```
http://www.filmstarts.de/suche/?q=Filmtitel
```

Wird nun mit *cURL* die URL mit dem oben genannten Film als Query–Parameter aufgerufen, so kommt als Antwort ein HTML–Dokument zurück, welches auch der Webbrowser erhalten würde. Folgende Shellsitzung demonstriert den Aufruf (Ausgabe gekürzt):

```
$ curl  http://www.filmstarts.de/suche/?q=The+Matrix
<html>
    <head><title>The Matrix - Suchen auf FILMSTARTS.de</title></head>
    <body><!-- Weiterer Content --></body>
</html>
```

Man bekommt als Aufrufer der URL die Webseite zurückgeliefert und muss nun die Daten aus dem Dokument extrahieren. Dies ist in der Regel mühsamer als der Zugriff über eine API, welche die Daten sauber im *JSON*– oder *XML*–Format zurückliefert.

5.1.1 Übertragung vom Grundprinzip auf das Pluginsystem

Ein Ziel bei der Entwicklung der Plugin–Spezifikation ist, den Aufwand für das Implementieren eines Plugins so gering wie möglich zu halten, um Programmierfehler beziehungsweise Fehlverhalten durch Plugins zu minimieren, aber auch um Entwickler zu motivieren, Plugins zu schreiben.

Das oben genannte Prinzip beim Beschaffen von Metadaten ist immer gleich und lässt sich somit gut auf das Pluginsystem übertragen. Die Provider–Plugins müssen im Prinzip nur folgende zwei Punkte können (siehe Abbildung 5.1):

- Aus den Suchparametern die *Such–URL* zusammenbauen.
- Extrahieren der Daten aus dem zurückgelieferten *HTTP–Response*.

Um den Download selbst muss sich das Provider–Plugin bei diesem Ansatz nicht kümmern. Das entlastet den Pluginentwickler und gibt *libhugin* die Kontrolle über das Downloadmanagement.

Damit der Provider weiß, welche *Roh–Daten* er zurückliefern soll, muss hierfür noch eine Struktur mit Attributen festgelegt werden, an welche sich alle Provider–Plugins halten müssen.

Für den Prototypen richten sich die möglichen Attribute nach der TMDb–Onlinequelle (siehe *libhugin*–API [Link–29]).

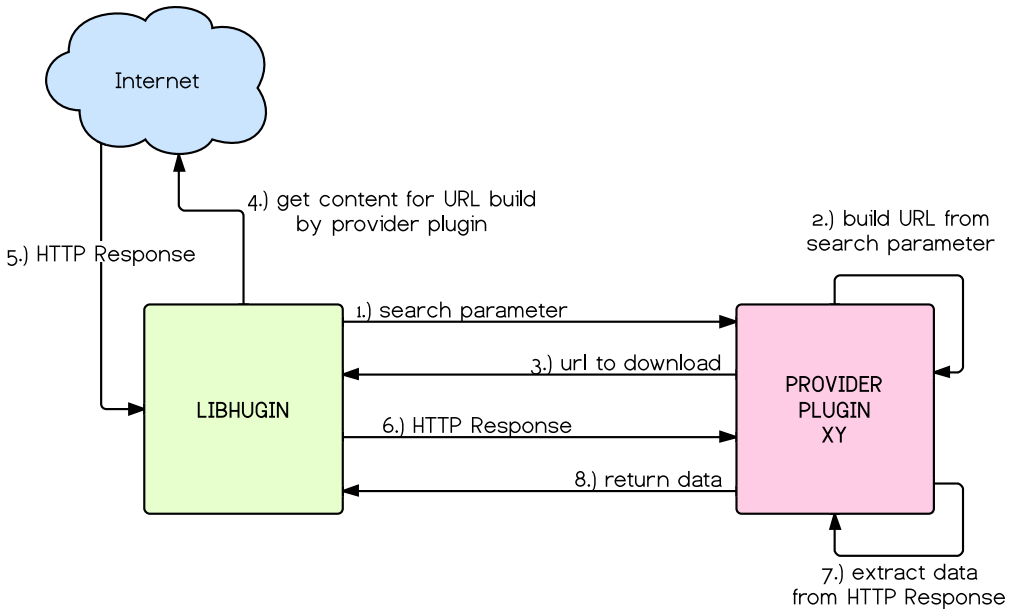

Abbildung 5.1.: Grundprinzip Kommunikationsablauf mit Provider–Plugin.

5.2 Libhugin Architektur–Überblick

Die Bibliothek soll über die Metadatenbeschaffung hinaus Werkzeuge zur Metadatenanalyse bereitstellen. Um eine saubere Trennung zwischen den beiden zu schaffen, wird die Bibliothek in die zwei Teile *libhugin–harvest* und *libhugin–analyze* aufgeteilt (siehe Abbildung 5.2).

libhugin–harvest: Dieser Teil soll für die Metadatenbeschaffung zuständig sein und Schnittstellen für die folgenden Pluginarten bereitstellen:

- Provider
- Postprocessor
- Converter

libhugin–analyze: Dieser Teil soll für die nachträgliche Metadatenanalyse zuständig sein und Schnittstellen für folgende Pluginarten bereitstellen:

- Modifier
- Analyzer
- Comparator

Der Analyze–Teil der Bibliothek soll über eine interne Datenbank verfügen, in welche externe Metadaten zur Analyse importiert werden. So können alle Plugins auf einem definierten Zustand arbeiten.

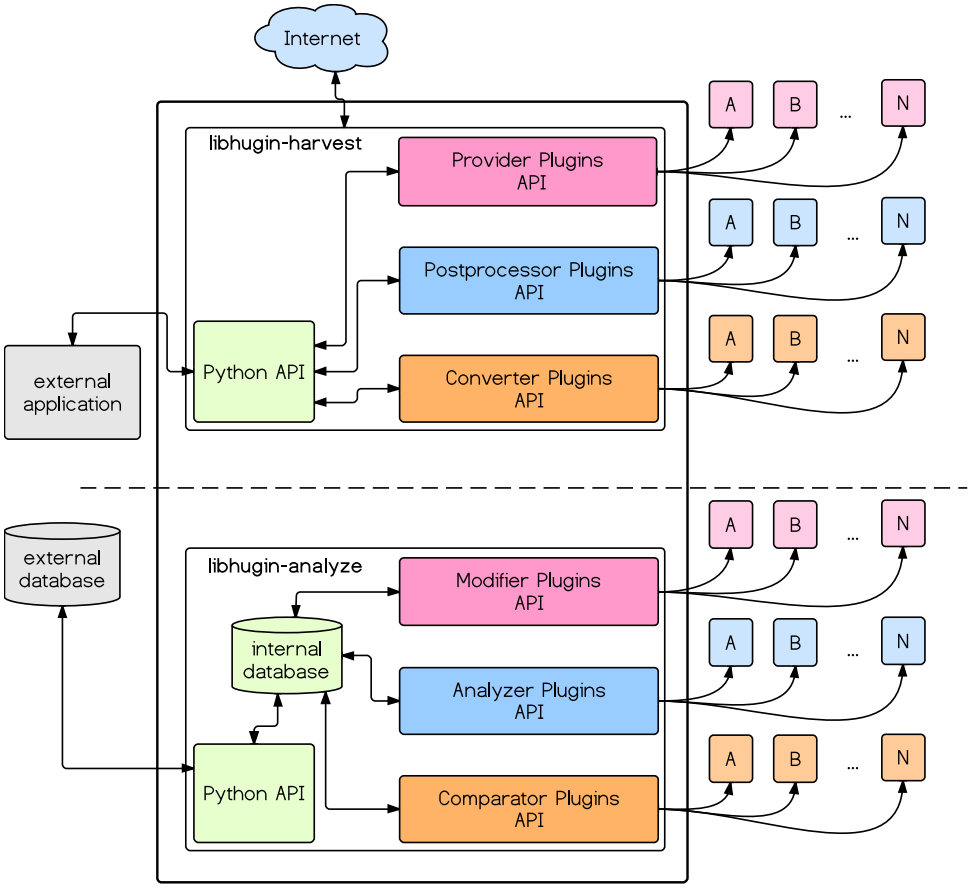

Abbildung 5.2.: Die Grafik zeigt eine Architekturübersicht der *libhugin*–Bibliothek welche sich in die zwei Teile *libhugin–harvest* und *libhugin–analyze* aufteilt.

5.2.1 Klassenübersicht von *libhugin–harvest*

Die Architektur von *libhugin* ist objektorientiert. Aus der Architekturübersicht und den Anforderungen an das System wurden die folgenden Klassen und Schnittstellen abgeleitet. Abbildung 5.3 zeigt eine Klassenübersicht von *libhugin–harvest*, samt Interaktion mit den Schnittstellen.

Im Folgenden werden die grundlegenden Objekte und Schnittstellen erläutert.

Session: Diese Klasse bildet den Grundstein für *libhugin–harvest*. Über eine Sitzung konfiguriert der Benutzer das System und hat Zugriff auf die verschiedenen Plugins. Von der Session werden folgende Methoden bereitgestellt:

`create_query(**kwargs)`: Schnittstelle zur Konfiguration der Suchanfrage. Die Methode gibt ein Query–Objekt zurück, das einem Python–Dictionary (Hashtabelle) entspricht. Diese Methode dient als Hilfestellung für den Benutzer der API. Theoretisch kann der Benutzer die Query auch manuell zusammenbauen. `Kwargs` ist eine optionale Liste aus Key–Value–Paaren. Für weitere Informationen und Konfigurationsparameter siehe *libhugin*–API [Link–30].

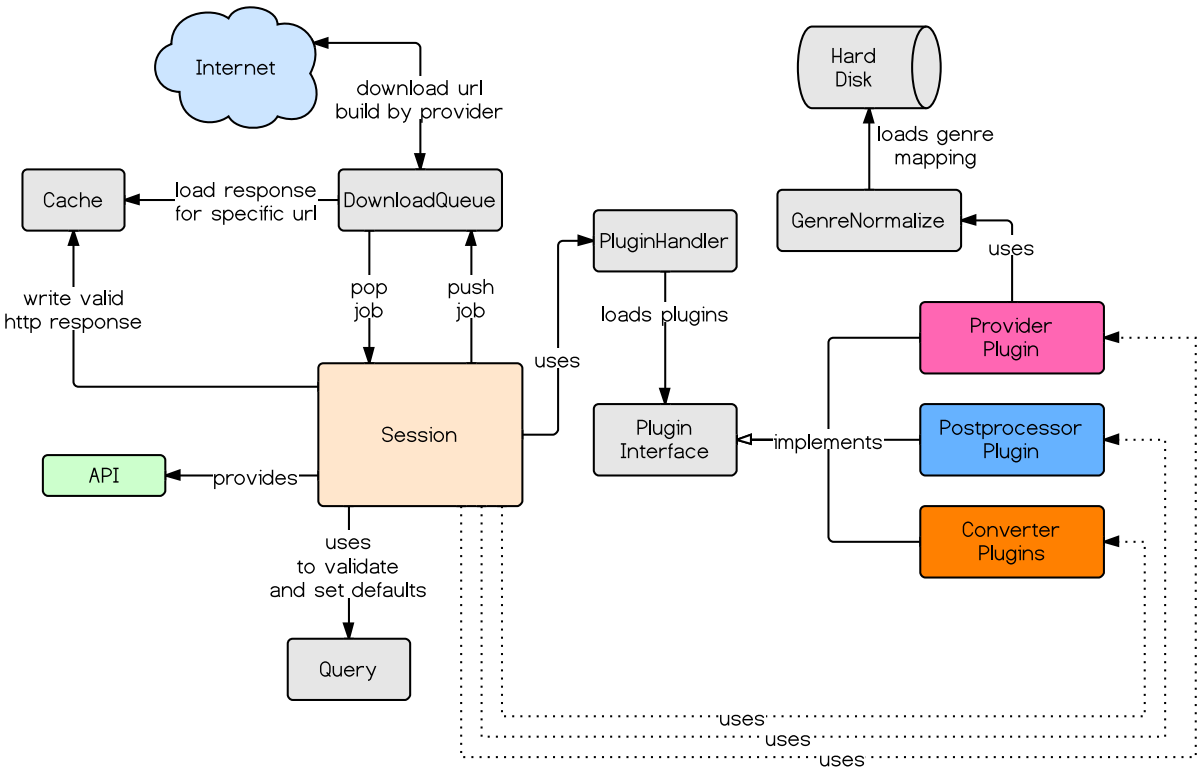

Abbildung 5.3.: *Libhugin–harvest* Klassenübersicht mit Klasseninteraktion.

`submit(query)`: Schnittstelle um eine Suchanfrage zu starten. Die Methode gibt eine Liste mit gefundenen Metadaten als *Ergebnisobjekte* zurück.

Die Methode initialisiert eine Downloadqueue und einen Zwischenspeicher (Cache), falls dieser vom Benutzer über die Query nicht deaktiviert wurde. Anschließend generiert sie für jeden Provider eine sogenannte *Job*–Struktur. Diese *Job*–Struktur kapselt jeweils einen Provider, die Suchanfrage und die Zwischenergebnisse, die während der Suchanfrage generiert werden.

Zur Veranschaulichung wird eine leere *Job*–Struktur in Python–Notation gezeigt:

```
job_structure = {
    'url': None,          # URL die als nächstes von Downloadqueue geladen werden soll
    'provider': None,     # Referenz auf Provider--Plugin
    'future': None,       # Referenz auf Future Objekt bei async. Ausführung
    'response': None,     # Ergebnis des Downloads, Http Response
    'return_code': None,  # Return Code der Http Anfrage
    'retries_left': None, # Anzahl der noch übrigen Versuche
    'done': None,         # Flag das gesetzt wird wenn Job fertig ist
    'result': None        # Ergebnis der Suchanfrage
}
```

Nachdem ein Job fertiggestellt wurde, wird er in ein *Ergebnisobjekt* gekapselt. Am Ende der `submit()`–Methode wird eine Liste mit *Ergebnisobjekten* an den Aufrufer zurückgegeben. Das *Ergebnisobjekt* kapselt die folgenden Informationen:

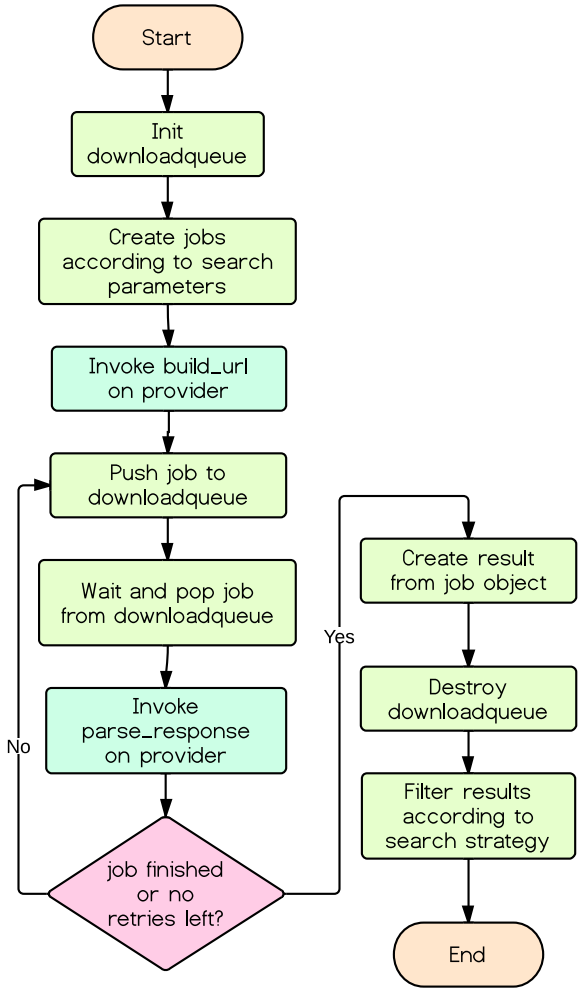

Abbildung 5.4.: Prinzipieller Ablauf der Submit–Methode.

- Provider, welcher das Ergebnis geliefert hat.
- Suchparemeter, welche für die Suchanfrage verwendet wurden.
- Metadatenart, Movie oder Person.
- Anzahl der Downloadversuche.
- Das eigentliche Ergebnis als Hashtabelle.

Der prinzipielle Ablauf der `submit()`–Methode wird in Abbildung 5.4 dargestellt.

`submit_async()`: Methode für eine asynchrone Nutzung der API. Diese führt `submit()` asynchron aus und gibt ein Python–*Future–Objekt* zurück, welches die Anfrage kapselt. Durch Aufrufen der `done()`–Methode auf dem *Future–Objekt*, kann festgestellt werden ob die Suchanfrage bereits fertig ist. Ein Aufruf der `result()`–Methode auf dem *Future–Objekt* liefert das eigentliche *Ergebnisobjekt* zurück. Für mehr Informationen siehe Python API [Link–31].

`provider_plugins(pluginname=None)`: Diese Methode gibt eine Liste mit den Provider–Plugins zurück oder bei Angabe eines Plugins, dieses direkt. Mit `pluginname=None` wird der Standardwert gesetzt, falls kein Wert übergeben wird.

`postprocessor_plugins(pluginname=None)`: Analog zu `provider_plugins()`.

`converter_plugins(pluginname=None)`: Analog zu `provider_plugins()`.

`cancel()`: Diese Methode dient zum Abbrechen einer asynchronen Suchanfrage. Hier sollte anschließend noch die `clean_up()`–Methode aufgerufen werden um alle Ressourcen wieder freizugeben.

`clean_up()`: Methode zum Aufräumen nach dem Abbrechen einer asynchronen Suchanfrage. Die Methode blockt solange noch nicht alle Ressourcen freigegeben wurden.

Queue: Die Queue kapselt die Parameter der Suchanfrage. Sie wird direkt mit den Parametern der Suchanfrage instanziiert, hierbei werden bestimmte Werte, die übergeben werden, validiert und es werden *Standardwerte* gesetzt.

Cache: Der Cache wird intern verwendet, um erfolgreiche Suchanfragen persistent zwischenzuspeichern. So können die Daten bei wiederholter Anfrage aus dem Cache geladen werden. Dadurch gewinnt man Geschwindigkeit und der Metadatenanbieter wird entlastet. Zum persistenten Speichern wird ein Python–Shelve (siehe [Link–32]) verwendet.

`open(path, cache_name)`: Öffnet den übergebenen Cache.

`read(key)`: Liest Element an Position *key* aus dem Cache.

`write(key, value)`: Schreibt das Element `value` an Position `key` in den Cache.

`close()`: Schließt den Cache.

Downloadqueue: Die Downloadqueue ist für den eigentlichen Download der Daten zuständig. Sie arbeitet mit den oben genannten *Job*–Strukturen. Die Provider–Plugins müssen so keine eigene Downloadqueue implementieren.

`push(job)`: Fügt einen `Job` der Downloadqueue hinzu.

`pop()`: Holt den nächsten fertigen `Job` aus der Downloadqueue.

`running_jobs()`: Gibt die Anzahl der `Jobs` die in Verarbeitung sind zurück.

GenreNormalize: GenreNormalize kann von den Provider–Plugins verwendet werden, um das Genre zu normalisieren. Hierzu müssen die Provider eine Genre–Mapping–Datei erstellen. Für mehr Informationen siehe auch API [Link–29].

`normalize_genre(genre)`: Normalisiert ein Genre anhand einer festgelegten Abbildungstabelle.

`normalize_genre_list(genrelist)`: Normalisiert eine Liste aus Genres jeweils mittels der `normalize_genre()` Funktion.

Die Problematik der Genrenormalisierung ist Bestandteil der Bachelorarbeit.

PluginHandler: Das Pluginsystem wurde mit Hilfe der *Yapsy*–Bibliothek (siehe [Link–33]) umgesetzt. Es bietet folgende Schnittstellen nach außen:

`activate_plugin_by_category(category)`: Aktiviert Plugins einer bestimmten Kategorie.

`deactivate_plugin_by_category(category)`: Deaktiviert Plugins einer bestimmten Kategorie.

`get_plugins_from_category(category)`: Liefert Plugins einer bestimmten Kategorie zurück.

`is_activated(category)`: Gibt einen Wahrheitswert zurück, wenn eine Kategorie bereits aktiviert ist.

5.2.2 Plugininterface libhugin–harvest

Libhugin–harvest bietet für jeden Plugintyp eine bestimmte Schnittstelle an, die vom jeweiligen Plugintyp implementiert werden muss (siehe Abbildung 5.5).

Diese *libhugin–harvest* Plugins haben die Möglichkeiten von verschiedenen Oberklassen abzuleiten (siehe Abbildung 5.6). Mehrfachvererbung ist unter Python möglich.

Plugins, die für die Metadatenbeschaffung zuständig sind, müssen von den Providerklassen ableiten (siehe Abbildung 5.6). Des Weiteren müssen diese Plugins die folgenden Methoden implementieren:

`build_url(search_params)`: Diese Methode bekommt die *Such–Parameter* übergeben und baut aus diesen die *Such–URL* zusammen. Für weitere Informationen siehe auch API [Link–34].

`parse_response(response, search_params)`: Diese Methode bekommt die HTTP–Response zu der vorher von `build_url(search_params)` erstellten *Anfrage–URL*. Die Methode ist für das Extrahieren der Attribute aus dem Response zuständig. Sie gibt entweder eine neue URL zurück, die angefordert werden soll, oder befüllt eine Hashtabelle mit gefundenen Attributen und gibt diese zurück. Für weitere Informationen siehe auch *libhugin*–API [Link–35].

`supported_attrs()`: Diese Methode gibt eine Liste mit Attributen zurück die vom Provider befüllt werden.

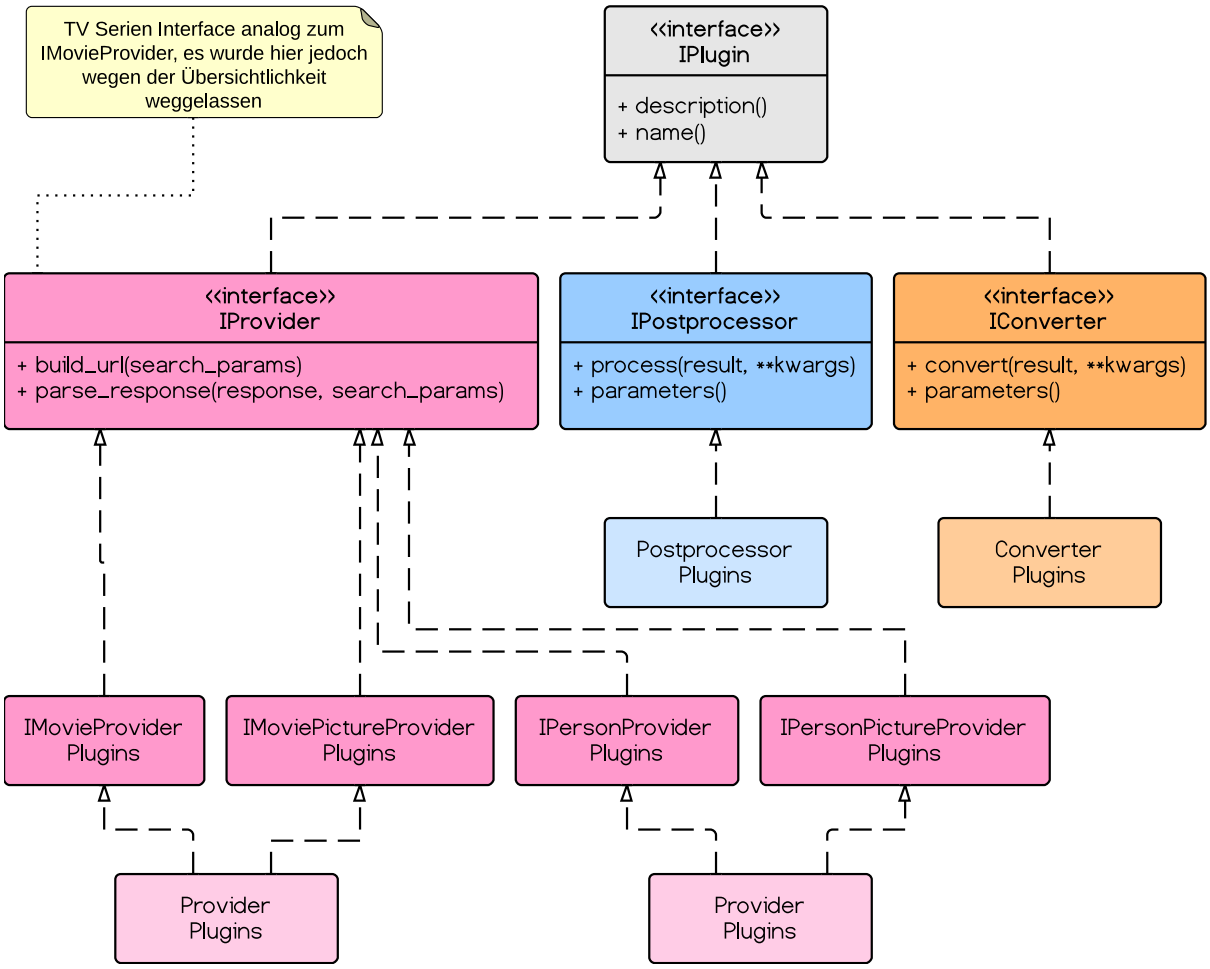

Abbildung 5.5.: Libhugin–harvest Plugin Schnittstellenbeschreibung.

Schnittstellenname	*text*	*grafisch*	*Beschreibung*
IMovieProvider	✓		Provider–Plugins, liefert Filmmetadaten
IMoviePictureProvider		✓	Provider–Plugins, liefert Filmmetadaten
IPersonProvider	✓		Provider–Plugins, liefert Personenmetadaten
IPersonPictureProvider		✓	Provider–Plugins, liefert Personenmetadaten
IPostProcessor			Postprocessor–Plugins für Metadatennachbearbeitung
IConverter			Converter–Plugins für verschiedene Metadatenformate

Abbildung 5.6.: Libhugin Plugininterfaces für die verschiedenen libhugin–harvest Plugins.

Plugins, die für die Metadatennachbearbeitung zuständig sind, müssen von *IPostProcessor* ableiten (siehe Abbildung 5.6). Des Weiteren müssen diese Plugins die folgenden Methoden implementieren:

`process(results, **kwargs)`: Diese Methode bekommt ein Liste mit *Ergebnisobjekten* übergeben und manipuliert dieses nach bestimmten Kriterien oder gibt eine neue Liste mit *Ergebnisobjekten* zurück.

`parameters()`: Die Methode listet die Keyword–Argumente für ein *Postprocessor*–Plugin.

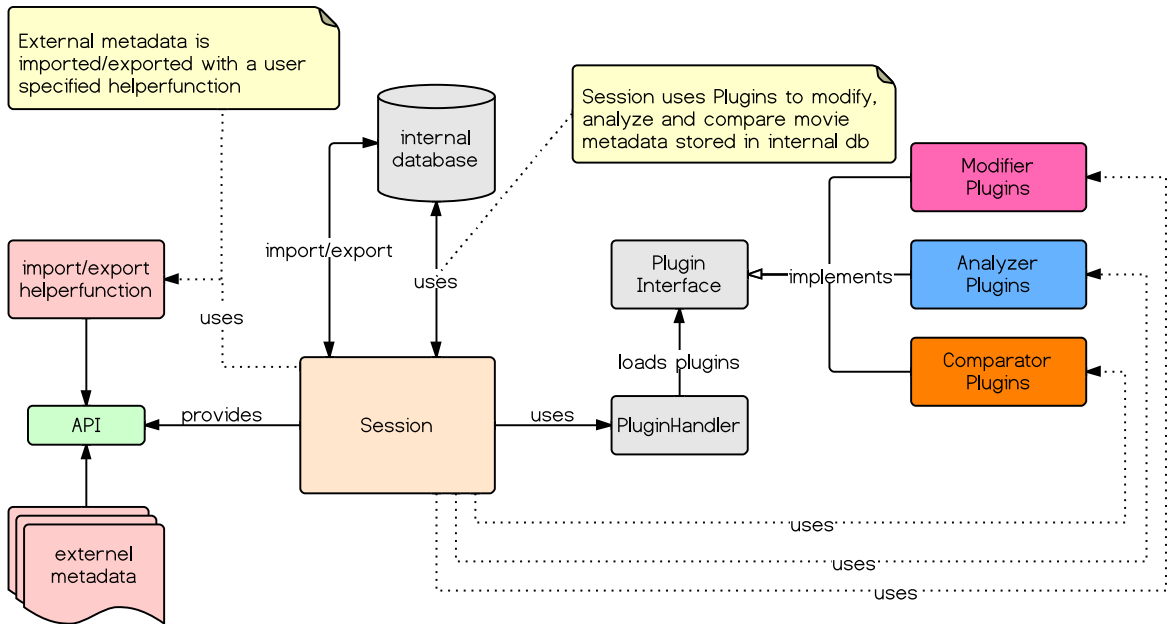

Abbildung 5.7.: *Libhugin–analyze* Klassenübersicht mit Klasseninteraktion.

Plugins, die für das Konvertieren der Ergebnisse in bestimmte Metadatenformate zuständig sind, müssen von *IConverter* ableiten (siehe Abbildung 5.6). Des Weiteren müssen diese Plugins die folgenden Methoden implementieren:

`convert(results, **kwargs)`: Diese Methode bekommt ein *Ergebnisobjekt* übergeben und gibt die String–Repräsentation von diesem in einem spezifischen Metadatenformat wieder.

`parameters()`: Die Methode listet die Keyword–Argumente für ein Converter–Plugin.

5.2.3 Klassenübersicht *libhugin–analyze*

Dieser Teil der *libhugin*–Bibliothek ist für die nachträgliche Metadatenaufbereitung zuständig (siehe Abbildung 5.7).

Session: Diese Klasse bildet den Grundstein für *libhugin–analyze*. Sie stellt analog zur *libhugin–harvest* Session die API bereit.

`add(metadata_file, helper)`: Diese Methode dient zum Importieren externer Metadaten. Sie erwartet eine Datei mit Metadaten (`metadata_file`) und als Callback–Funktion eine *Helferfunktion* welche weiß, wie die Metadaten zu extrahieren sind.

Kurzer Exkurs zur *Helferfunktion*. Die *Helferfunktion* hat folgende Schnittstelle:

```
helper_func(metadata, attr_mask)
```

Der `attr_mask` Parameter gibt die Abbildungen der Attribute zwischen der *externen* und *internen* Datenbank an.

Wir nehmen an unsere Metadaten sind im *JSON*–Format gespeichert, beim Einlesen der *JSON–Datei* wird diese zu einer *Hashtabelle* konvertiert, die wie folgt aussieht:

```
metadata_the_movie = {
    'Filmtitel' = 'The Movie',
    'Erscheinungsjahr' = '2025',
    'Inhaltsbeschreibung' = 'Es war einmal vor langer langer Zeit...'
}
```

Folgendes Python–Snippet zeigt nun die Funktionalität der *Helferfunktion*, welche die Abbildung von externer Quelle auf die interne Datenbank verdeutlicht:

```
 attr_mask = {
     'Filmtitel': 'title',
     # Filmtitel = Attributname unter welchem der Filmtitel
     # in der externen Metadatendatei hinterlegt ist
     # title = Attributname unter dem der Titel
     # in der internen Datenbank abgelegt werden soll
     #
     # folgenden zwei Attribute analog zum Filmtitel
     'Erscheinungsjahr' = 'year',
     'Inhaltsbeschreibung': 'plot'
 }

def helper(metadata, attr_mask):
    internal_repr = {}

    for metadata_key, internal_db_key in attr_mask.items():
        internal_repr[internal_db_key] = metadata[metadata_key]

    return internal_repr
```

Weitere Methoden der Session–Klasse:

`analyzer_plugins(pluginname=None)`: Liefert eine Liste mit den vorhandenen Analyzer–Plugins zurück. Bei Angabe eines bestimmten Pluginnamen, wird dieses Plugin direkt zurückgeliefert.

`modifier_plugins(pluginname=None)`: Analog zu `analyzer_plugins()`.

`comparator_plugins(pluginname=None)`: Analog zu `analyzer_plugins()`.

Folgende weitere Methoden erlauben es, die *libhugin–analyze* Plugins auf *externe* Daten anzuwenden:

`analyze_raw(plugin, attr, data)`: Wrapper Methode, welche es erlaubt die Analyzer–Plugins auf *externen* Daten auszuführen.

`modify_raw(plugin, attr, data)`: Analog zu `analyze_raw(plugin, attr, data)`.

`compare_raw(plugin, attr, data)`: Analog zu `analyze_raw(plugin, attr, data)`.

`get_database()`: Liefert die interne Datenbank (Python–Dictionary) zurück.

Für das Öffnen und Schließen der internen Datenbank der Session gibt es folgende zwei Methoden:

`database_open(databasename)`: Lädt die angegebene Datenbank.

`database_close()`: Schließt und schreibt die aktuelle Datenbank persistent auf die Festplatte.

Movie: Die Movie Klasse repräsentiert ein Metadatenobjekt welches in der internen Datenbank zur Analyse gespeichert wird. Es enthält folgende Attribute:

- Schlüssel, über den die Metadaten eindeutig zugeordnet werden können.
- Pfad zur Metadatendatei.
- Hashtabelle mit den Metadaten.
- Hashtabelle mit Analyzer–Analysedaten.
- Hashtabelle mit Comparator–Analysedaten.

PluginHandler: Die PluginHandler–Klasse hat analog zum *libhugin–harvest* die folgenden Schnittstellen:

`activate_plugin_by_category(category)`: Aktiviert Plugins einer bestimmten Kategorie.

`deactivate_plugin_by_category(category)`: Deaktiviert Plugins einer bestimmten Kategorie.

`get_plugins_from_category(category)`: Liefert Plugins einer bestimmten Kategorie zurück.

`is_activated(category)`: Gibt einen Wahrheitswert zurück, wenn eine Kategorie bereits aktiviert ist.

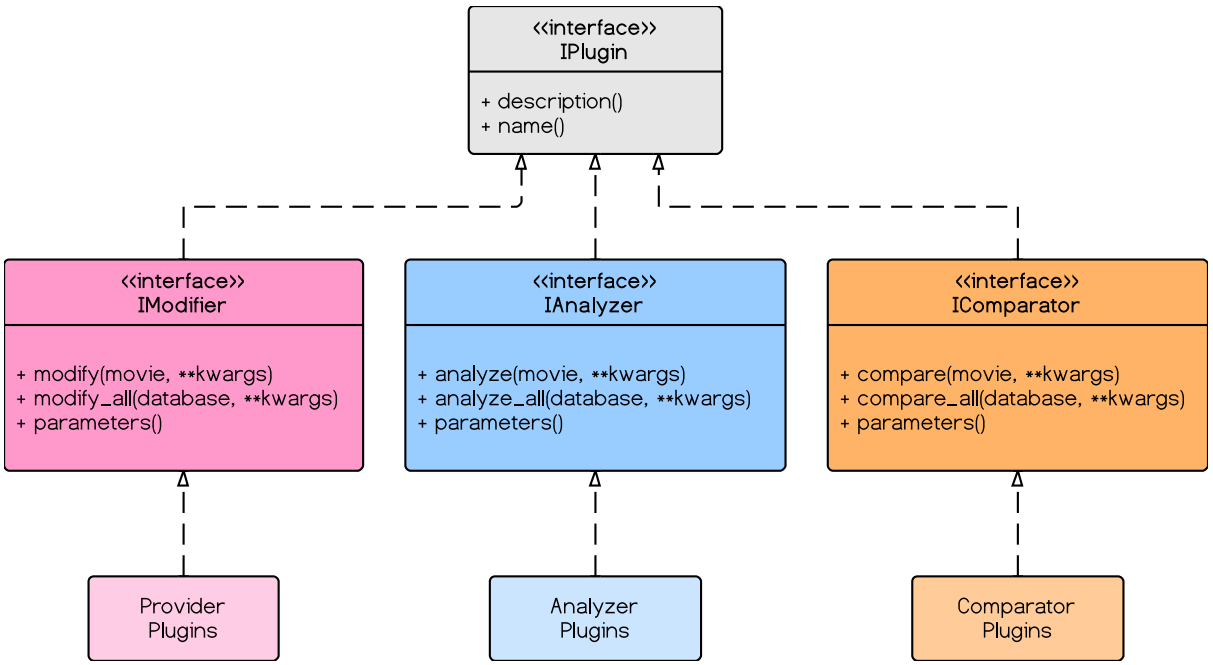

Abbildung 5.8.: Libhugin–analyze Plugin Schnittstellenbeschreibung.

5.2.4 Plugininterface libhugin–analyze

Libhugin–analyze bietet für jeden Plugintyp eine bestimmte Schnittstelle an, die vom jeweiligen Plugintyp implementiert werden muss (siehe Abbildung 5.8).

Die *libhugin–analyze* Plugins haben die Möglichkeiten, von den folgenden Oberklassen abzuleiten:

Schnittstellenname	*Beschreibung*
IModifier	Modifier–Plugins, die Metadaten direkt modifizieren.
IAnalyzer	Analyzer–Plugins, die für die Analyse der Metadaten zuständig sind.
IComparator	Comparator–Plugins, die Metadaten für statistische Zwecke vergleichen.

Abbildung 5.9.: Libhugin Plugininterfaces für die verschiedenen libhugin–analyze Plugins.

Plugins, die Metadaten modifizieren, müssen von *IModifier* ableiten (siehe Abbildung 5.9). Diese Plugins müssen folgende Methoden implementieren:

`modify(movie, **kwargs)`: Die Standardmethode für Modifierplugins. Die Methode bekommt ein *Movie–Objekt* und optional Keyword–Argumente übergeben. Die nötigen Keyword–Argumente können über die `parameters()`–Methode abgefragt werden.

`modify_all(database, **kwargs)`: Analog zur `modify(movie, kwagrs)`–Methode. Diese Methode arbeitet jedoch nicht mit nur einem Movie Objekt sondern mit der ganzen „Datenbank".

`parameters()`: Die Methode listet die Keyword–Argumente für ein Modifierplugin.

Plugins, die für die Analyse der Metadaten zuständig sind, müssen von *IAnalyzer* ableiten (siehe Abbildung 5.9). Diese Plugins schreiben ihre Analysedaten in das *Analyzerdata*–Attribut des *Movie–Objekts*. Sie müssen folgende Methoden implementieren:

`analyze(movie, **kwargs)`: Die Standardmethode für Analyzerplugins. Die Anwendung hier ist analog den Modifierplugins.

`analyze_all(database, **kwargs)`: Analog Modifierplugins.

`parameters()`: Analog zu Modifierplugins.

Plugins, die Metadaten für statistische Zwecke analysieren und vergleichen können, müssen von *IComparator* ableiten (siehe Abbildung 5.9). Des Weiteren müssen diese Plugins folgende Methoden implementieren:

`compare(movie_a, movie_b, **kwargs)`: Die Standardmethode für Comparatorplugins. Diese erwartet als Parameter zwei *Movie–Objekte*, die verglichen werden sollen. Die Keyword–Argumente können analog den Modifier– und Analyzerplugins verwendet werden.

`compare_all(database, **kwargs)`: Diese Methode vergleicht alle *Movie–Objekt* Kombinationen aus der Datenbank.

`parameters()`: Analog zu Modifier- und Analyzerplugins.

5.3 Bibliothek Dateistruktur

Die folgende Auflistung zeigt die Ordnerstruktur der Bibliothek. Normalerweise enthält unter Python jeder Ordner eine *__init__.py-Datei* welche diesen Ordner dann als Modul erscheinen lässt. Diese wurden wegen der Übersichtlichkeit weggelassen.

```
hugin
|-- harvest/                          # libhugin--harvest Ordner
|   |-- session.py                    # Implementierungen der Session
|   |-- query.py                      # Implementierungen der Query
|   |-- cache.py                      # Implementierungen vom Cache
|   |-- downloadqueue.py              # Implementierungen der Downloadqueue
|   |-- pluginhandler.py              # Implementierungen vom PluginHandler
|   |
|   |-- converter/                     # Ordner für Converter--Plugins
|   |-- postprocessor/                 # Ordner für Postprocessor--Plugins
|   |-- provider/                      # Ordner für Provider--Plugins
|   |   |-- genrefiles/                # Genre Dateien für ,,Normalisierung"
|   |   |   |-- normalized_genre.dat   # Globale Normalisierungstabelle Genre
|   |   |-- result.py                  # Implementierung ,,ErgebnisObjekt"
|   |   |-- genrenorm.py               # Implementierung Genrenormalisierung
|-- utils/                            # Gemeinsame Hilfsfunktionen
|   |-- logutil.py
|   |-- stringcompare.py
|
|-- analyze/                          # libhugin--analyze Ordner
|   |-- session.py                    # Implementierungen der o.g. Klassen
|   |-- movie.py                      # Implementierung des ,,Movie" Objektes
|   |-- pluginhandler.py
|   |-- rake.py                       # Implementierung Rake Algorithmus
|   |-- analyzer/                     # Ordner für Analyzer--Plugins
|   |-- comparator/                   # Ordner für Comparator--Plugins
|   |-- modifier/                     # Ordner für Modifier--Plugins
|-- filewalk.py                       # Helferfunktion für Import/Export
```

6 | Implementierung

Im Folgenden soll die API und die implementierten Plugins vorgestellt werden.

6.1 Libhugin-harvest API

Die API wurde sehr einfach gehalten und ermöglicht dadurch dem Benutzer ein schnelles Einarbeiten. Folgendes Beispiel, in der interaktiven Python-Shell, zeigt die typische Benutzung der API:

```
>>> from hugin.harvest.session import Session
>>> session = Session()
>>> query = session.create_query(title='Prometheus')
>>> results = session.submit(query)
>>> print(results)
[<TMDBMovie <picture, movie> : Prometheus (2012)>,
<OFDBMovie <movie> :  Prometheus - Dunkle Zeichen (2012)>,
<OMDBMovie <movie> : Prometheus (2012)>]
```

Für weitere Beispiele siehe offizielle *libhugin-harvest* API [Link-36].

6.2 Libhugin-harvest Plugins

6.2.1 Provider-Plugins

Libhugin-harvest hat aktuell verschiedene Provider implementiert (siehe Abbildung 6.1).

Ein paar der Provider, wie `Filmstarts`, `Videobuster` lassen sich noch weiter ausbauen. Diese unterstützen momentan nur textuelle Metadaten, würden sich aber um grafische Metadaten erweitern lassen. Die in der Tabelle festgelegten Prioritäten wurden hier willkürlich für Testzwecke vergeben. Diese können jederzeit an die eigenen Bedürfnisse angepasst werden.

	TMDb	*OFDb*	*Videobuster*	*Filmstarts*	*OMDb*
Priorität	90	80	70	65	65
Providerart	movie/person	movie/person	movie	movie	movie
Metadaten	textuell/grafisch	textuell	textuell	textuell	textuell
Sprache	multilingual	deutsch	deutsch	deutsch	englisch
Unschärfesuche Online	×	×	×	×	×
Unschärfesuche libhugin	✓	✓	✓	✓	✓
IMDB–Suche Online	✓	✓	×	×	✓
IMDB–Suche libhugin	✓	✓	✓	✓	✓
API verfügbar	✓	✓	×	×	✓

Abbildung 6.1.: Übersicht implementierter Provider und Funktionalität.

6.2.2 Postprocessor–Plugins

Die Postprocessor–Plugins beim *libgugin–harvest* Teil sind für die direkte „Nachbearbeitung" der Daten gedacht.

Compose: Das `Compose`–Plugin ist das momentane Kernstück der Postprocessor–Plugins. Das Plugin gruppiert die Ergebnisse verschiedener Onlinequellen nach Film und bietet dem Benutzer dadurch folgende Möglichkeiten:

1. Ergebnis komponieren.

2. Genre zusammenführen.

Zu 1.): Es erlaubt dem Benutzer sich ein nach seinen Wünschen zusammengesetztes Ergebnis zu komponieren. Der Benutzer kann über das Angeben einer Profilmaske bestimmen, wie sich die Metadaten zusammensetzen sollen. Hier kann er beispielsweise angeben, dass er den Filmtitel, Jahr und Cover vom Provider *TMDb* möchte, die Inhaltsbeschreibung jedoch immer vom *Filmstarts* Provider. Hier besteht auch die Möglichkeit eines „Fallbacks", falls *Filmstarts* keine Inhaltsbeschreibung hat, dann kann auch auf andere Provider zurückgegriffen werden.

Beispiel für eine Profilmaske, die TMDb als Standardprovider nimmt und die Inhaltsbeschreibung vom OFDb–Provider nimmt. Falls keine OFDb–Inhaltsbeschreibung vorhanden ist, dann erfolgt ein „Fallback" auf den OMDb–Provider:

```
$ echo "{'default':['tmdbmovie'], 'plot':['ofdbmovie', 'omdbmovie']}" > profilemask
```

Wird keine Profilmaske angegeben, so werden fehlende Attribute nach Provider–Priorität aufgefüllt.

Zu 2.): Dieses Feature erlaubt dem Benutzer divergente Genres beim gleichen Film zu verschmelzen. Das macht das Genre feingranularer und behebt die genannte Problematik (siehe Abbildung 3.4) divergenter Genres bei verschiedenen Onlinequellen. Das Genre wird hier wie folgt zusammengesetzt:

```
# Drei Genre der Unterschiedlichen Provider      # Zusammengeführtes Genre
[Comedy, Drama], [Komödie, Drama], [Erotik] ---> [Komödie, Drama, Erotik]
```

Trim: Dies ist vergleichsweise ein einfaches Plugin, welches dafür zuständig, ist vorangehende und nachziehende Leerzeichen bei den Metadaten zu entfernen. Das Plugin führt eine Bereinigung durch, diese muss nicht explizit vom Provider–Plugin durchgeführt werden.

6.2.3 Converter–Plugins

Bei den Converter–Plugins wurde zu Demonstrationszwecken ein *HTML*–Converter und ein *JSON*–Converter implementiert.

Des Weiteren wurde für den Produktiveinsatz ein XBMC *Nfo*–Converter implementiert, dieser wird vom *libhugin*–Proxy (siehe Libhugin–Proxy, 7.3.1) verwendet, um dem XBMC–libhugin Plugin (siehe XBMC Plugin Integration, 7.3) die Metadaten im richtigen Format zu liefern.

6.3 Libhugin–analyze API

Die API von *libhugin–analyze* ist vom Grundaufbau ähnlich zu der *libhugin–harvest* API. Folgendes Beispiel–Snippet zeigt die Anwendung des `BracketClean`–Plugins auf *Rohdaten*, welche nicht aus der internen Datenbank stammen.

```
>>> from hugin.analyze.session import Session
    # Beispieltext. Erstelle Sitzung mit Dummy DB. Hole BracketClean Plugin.
>>> example_text = "Aus diesem Text wird die Klammer (welche?) samt Inhalt entfernt!"
>>> session = session('/tmp/temporary.db')
>>> BracketClean = session.modifier_plugins('BracketClean')
    # Wende Plugin im raw Modus auf Daten an
>>> result = session.modify_raw(BracketClean, 'plot', example_text)
>>> print(result)
Aus diesem Text wird die Klammer samt Inhalt entfernt!
```

Weitere Beispiele bezüglich Einsatz von *libhugin–analyze* siehe Demoanwendung Freki (siehe 7.2).

6.4 Libhugin–analyze Plugins

6.4.1 Modifier–Plugins

BracketClean: Das `BracketClean`–Plugin ist für nachträgliche Manipulation der Inhaltsbeschreibung gedacht. Das Plugin entfernt alle Klammern samt Inhalt aus der Beschreibung. Das vereinheitlicht die Inhaltsbeschreibung in dem Sinne, dass alle Schauspieler oder Informationen in Klammern aus der Beschreibung entfernt werden.

PlotLangChange: Das `PlotLangChange`–Plugin ist für das nachträgliche Ändern der Inhaltsbeschreibung zuständig. Es hat die Funktion, die Sprache des Plots zu ändern.

6.4.2 Analyzer–Plugins

KeywordExtract: Dieses Plugin extrahiert aus einem Text, bei Filmen meist die Inhaltsbeschreibung, relevante Schlüsselwörter, die den Text beziehungsweise die darin dargestellte Thematik repräsentieren.

FileTypeAnalyze: Das `FileTypeAnalyze`–Plugin arbeitet mit den Videodaten selbst. Es ist für die Extraktion der Datei–Metadaten zuständig. Momentan extrahiert es:

- Auflösung
- Seitenverhältnis
- Videocodec
- Audiocodec, Anzahl der Audiokanäle, Sprache

LangIdentify: Der `LangIdentify`–Analyzer erkennt die Sprache des verwendeten Plots und schreibt die Information zu den Analysedaten.

6.4.3 Comparator–Plugins

Dieser Plugintyp ist experimentell, er ist für statistische Zwecke und Analysen bezüglich der Vergleichbarkeit von Filmen anhand der Metadaten gedacht.

Folgende Comparator–Plugins wurden konzeptionell implementiert:

GenreCmp: Ein Plugin, das die Genres verschiedener Filme miteinander vergleicht.

KeywordCmp: Ein Plugin, das die Schlüsselwörter verschiedener Filme miteinander vergleicht.

6.5 Verschiedenes

6.5.1 Testverfahren

Für das Testen der Software wird das Python Unittest–Framework verwendet. Bisher wurden Tests für die wichtigsten Grundklassen und das Provider–Pluginsystem erstellt, um ein valides Verhalten der Provider–Plugins zu gewährleisten.

Die Unittests wurden direkt in der „Main" der jeweiligen Klasse untergebracht. Diese werden dann beim Ausführen der Python–Datei gestartet.

Folgendes Beispiel zeigt die Funktionsweise:

```
def add(a, b): return a + b

if __name__ == '__main__':
    import unittest

    class SimpleTest(unittest.TestCase):
        def test_add_func(self):
            result = add(21, 21)
            self.assertTrue(result == 42)

    unittest.main()
```

Das Ausführen des Beispielcodes würde folgende Ausgabe produzieren:

```
Ran 1 test in 0.000s

OK
```

Alle geschriebenen Tests werden bei jedem „Einspielen" der Änderungen in das verwendete Quellcode–Versionsverwaltungssystem automatisiert über einen externen Dienst ausgeführt (siehe Entwicklungsumgebung, 6.5.2).

6.5.2 Entwicklungumgebung

Programmiersprache: Für die Entwicklung der Bibliothek wurde die Programmiersprache Python, in der Version 3.3, aus folgenden Gründen gewählt:

Rapid Prototyping Language:
Wichtig bei einem Projekt dieser Größe mit begrenztem Zeitraum (vgl. [2]).

Plattformunabhängigkeit:
Plattformunabhängigkeit ist ein sekundäres Ziel des Projekts.

Einfach erlernbar:
Wichtig für Pluginentwickler wegen der kurzen Einarbeitungszeit.

Verbreitungsgrad:
Gängige Skriptsprache bei vielen Open–Source–Projekten.

Optimierungsmöglichkeiten:
Möglichkeit der Erweiterung durch C/C++–Code, Optimierung von Python mittels Cython (siehe [Link–37], vgl. [2]).

Entwicklungssytem: Die Bibliothek wird unter *Archlinux* entwickelt. Für die Entwicklung wird der Editor *gVim* mit entsprechenden Python–Plugins zur Validierung der Python PEP–Stilrichtlinien (siehe [Link–38]) verwendet. Des Weiteren wird die interaktive Python Shell *IPython* eingesetzt.

Quellcodeverwaltung: Für die Quellcodeverwaltung wird das Versionsverwaltungssystem *git* eingesetzt. Der Quellcode selbst wird auf dem Hosting–Dienst für Software–Entwicklungsprojekte *GitHub* (siehe [Link–39]) gelagert. Das Projekt ist auf folgender GitHub Seite zu finden:

- https://github.com/qitta/libhugin

Automatisches Testen: Die oben genannten Softwaretests werden von *TravisCI* (siehe [Link–40]), einem sogenanntem „Continuous Integration Service" automatisch ausgeführt. Dies passiert bei jedem Hochladen von Quellcodeänderungen auf *GitHub*. *GitHub* hat hier eine Service–Schnittstelle zu *TravisCI,* welche aktiviert wurde.

Ein Symbol (siehe Abbildung 6.2) auf der *libhugin* Github–Projektseite teilt so dem Besuchern der Seite den aktuellen „Projektstatus" mit.

Projektdokumentation: Das Projekt wird nach dem Prinzip der *literalen Programmierung* entwickelt, wie von *Donald E. Knuth* (siehe [Link–41]) empfohlen. Hierbei liegen Quelltext und Dokumentation des Programms in der gleichen Datei.

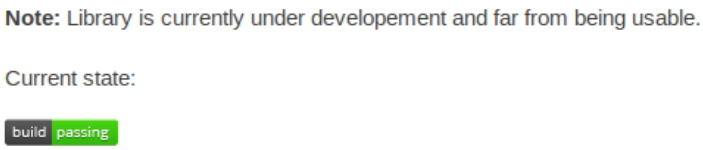

Abbildung 6.2.: Symbol, das den aktuellen „Build–Status" der GitHub–Projektseite zeigt.

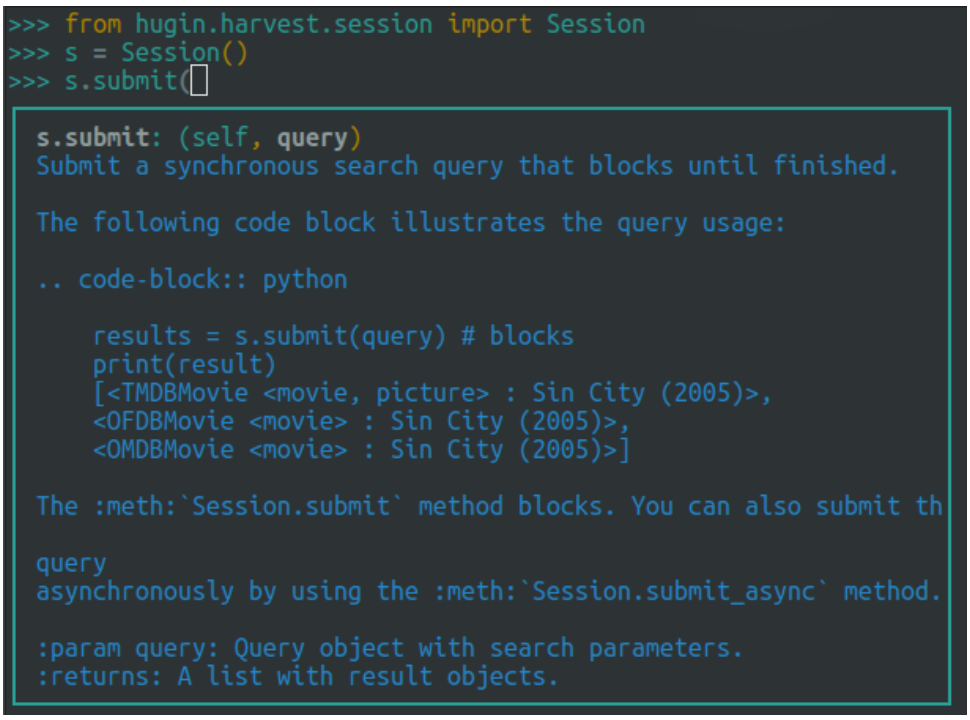

Abbildung 6.3.: API–Dokumentation als Hilfestellung in der interaktiven Python–Shell bpython.

Die Dokumentation kann so über spezielle Softwaredokumentationswerkzeuge generiert werden. Unter Python wird hier das Softwaredokumentationswerkzeug *Sphinx* (siehe [Link–42]) verwendet. Die offizielle Projektdokumentation, aktuell hauptsächlich der *libhugin-harvest* Teil, ist auf der Plattform *ReadTheDocs* (siehe [Link–43]) gehostet und unter folgender Adresse zu finden:

- http://libhugin.rtfd.org

Dieses kann eine Dokumentation in verschiedenen Formaten generieren, auch diese Projektarbeit wurde in *reStructuredText* (siehe [Link–44]) geschrieben und mit *Sphinx* generiert.

Des Weiteren wird dem Entwickler bei Nutzung der Bibliothek in der interaktiven Python–Shell eine zusätzliche Hilfestellung geboten (siehe Abbildung 6.3).

Projektumfang: Der Projektumfang beträgt ca. 3500 *lines of code*, hinzu kommt noch die Onlinedokumentation. Eine Statistik zum Projekt, welche mit dem Tool *cloc* erstellt wurde, ist im Anhang unter *Projektstatistik (cloc)* zu finden.

Externe Bibliotheken: Die Tabelle 6.4 listet alle verwendeten externen Abhängigkeiten für die *libhugin*–Bibliothek.

Abhängigkeit	*Verwendung in*	*Einsatzzweck*
yapsy	Pluginsystem	Laden von Plugins
charade	Downloadqueue	Encodingerkennung
parse	Plugins	Parsen von Zeitstrings
httplib2	Downloadqueue	Content download
jinja2	Plugins	HTML Template Engine
docopt	Cli–Tools	CLI–Optionparser
Flask	Huginproxy	Webframework, RESTful interface
guess_language-spirit	Plugins	Spracherkennung
PyStemmer	Plugins	Stemming von Wörtern
pyxDamerauLevenshtein	Plugins, Utils	Vergleich von Strings
Pyaml	Plugins	Verarbeitung von Yaml Dateien
beaufifulsoup4	Plugins	Parsen von HTML Seiten
xmltodict	Plugins	Verarbeitung von XML Dokumenten
hachoir-metadata	Plugins	Extraktion von Datei–Metadaten

Abbildung 6.4.: Übersicht über externe Abhängigkeiten.

7 | Demoanwendungen

Die vorgestellten CLI–Tools stellen nur einen kleinen Ausschnitt der Fähigkeiten der Bibliothek dar. Die Bibliothek selbst ist um fast jede denkbare Funktionalität der Metadatenaufbereitung erweiterbar. Die Algorithmik der verwendeten Plugins und Funktionen ist Bestandteil der Bachelorarbeit.

7.1 Libhugin–harvest CLI–Tool Geri

Geri ist eine CLI–Anwendung, die zu Demozwecken, aber auch als Testwerkzeug für die *libhugin–harvest* Bibliothek verwendet werden kann.

7.1.1 Übersicht der Optionen

Ein Überblick über die Funktionalität und die möglichen Optionen zeigt die Hilfe des Tools:

```
$ geri -h
Libhugin commandline tool.

Usage:
  geri (-t <title>) [-y <year>] [-a <amount>] [-p <providers>...] [-c <converter>] \
       [-o <path>] [-l <lang>] [-P <pm>]  [-r <processor>] [-f <pfile>] [-L]
  geri (-i <imdbid>) [-p <providers>...] [-c <converter>] [-o <path>] [-l <lang>] \
       [-r <processor>] [-f <pfile>] [-L]
  geri (-n <name>) [--items <num>] [-p <providers>...] [-c <converter>] [-o <path>]
  geri list-provider
  geri list-converter
  geri list-postprocessor
  geri -h | --help
  geri --version

 Options:
   -t, --title=<title>              Movie title.
   -y, --year=<year>                Year of movie release date.
   -n, --name=<name>                Person name.
   -i, --imdbid=<imdbid>            A imdbid prefixed with tt.
   -p, --providers=<providers>      Providers to be used.
   -c, --convert=<converter>        Converter to be used.
   -r, --postprocess=<processor>    Postprocessor to be used.
```

```
   -o, --output=<path>              Output folder for converter result [default: /tmp].
   -a, --amount=<amount>            Amount of items to retrieve.
   -l, --language=<lang>            Language in ISO 639-1 [default: de]
   -P, --predator-mode              The magic 'fuzzy search' mode.
   -L, --lookup-mode                Does a title -> imdbid lookup.
   -f, --profile-file=<pfile>       User specified profile.
   -v, --version                    Show version.
   -h, --help                       Show this screen.
```

Das Tool eignet sich neben dem Einsatz als Testwerkzeug für die Bibliothek auch gut für Skripte und somit für automatische Verarbeitung großer Datenmengen, siehe auch *Scripting Tasks* 7.4.

7.1.2 Filmsuche

Ein Film kann über den Titel oder über die *IMDb–ID* gesucht werden. Hier gibt es die Möglichkeit, *Geri* auch bestimmte Provider, Converter, Sprache und Postprocessor–Plugins anzugeben.

Um das Ausgabeformat zu konfigurieren, gibt es im *Geri*–Ordner eine `movie.mask`– und `person.mask`–Datei. Über diese Dateien kann das Ausgabeformat definiert werden. Die Syntax ist einfach. Um Attribute darzustellen, werden diese einfach in geschweifte Klammern geschrieben. Das *num*–Attribut gibt Geri die Möglichkeit, die Resultate durchzunummerieren.

Folgend die Definition vom Ausgabeformat für die `movie.mask`:

```
$ echo "{num}) {title} ({year}), IMDBid: {imdbid} Provider: {provider} \
\nInhalt: {plot}" > tools/geri/movie.mask
```

Standardsuche nach Titel mit der Begrenzung auf fünf Ergebnisse:

```
$ geri --title "sin city" --amount=5
1) Sin City (2005), IMDBid: tt0401792, Provider: TMDBMovie <picture, movie>
Inhalt: Basin City, genannt Sin City, ist ein düsteres Metropolis, in dem nichts
und niemand wirklich sicher ist, in dem die Gewalt allgegenwärtig ist [...]

2) Sin City (2005), IMDBid: tt0401792, Provider: OFDBMovie <movie>
Inhalt: Basin City, genannt Sin City, ist ein düsteres Metropolis, in dem nichts
und niemand wirklich sicher ist, in dem die Gewalt allgegenwärtig ist [...]

3) Sin City (2005), IMDBid: None, Provider: VIDEOBUSTERMovie <movie>
Inhalt: Willkommen in Sin City.  Diese Stadt begrüßt die Harten, die Korrupten,
die mit den gebrochenen Herzen.  Einer von ihnen ist Marv [...]

4) Sin City (2005), IMDBid: tt0401792, Provider: OMDBMovie <movie>
Inhalt: Four tales of crime adapted from Frank Millers popular comics focusing
```

```
around a muscular brute whos looking for the person responsible for the [...]

5) Sin City (2005), IMDBid: None, Provider: FILMSTARTSMovie <movie>
Inhalt: "Sin City" enthält drei lose verbundene und ineinander verschachtelt
erzählte Episoden: Los geht es mit Hartigan (Bruce Willis) - einem Cop [...]
```

Die Suche kann, wie die Optionen des Tools zeigen, feingranularer konfiguriert werden. Es würde jedoch den Rahmen sprengen alle Optionen zu zeigen.

Unschärfesuche: Ein weiteres nennenswertes Feature ist die Unschärfesuche. Die getesteten Tools (siehe Abbildung 3.7) sind nicht in der Lage, Filme zu finden, wenn der Titel nicht exakt geschrieben ist. Das trifft auch in der Standardkonfiguration für *libhugin* zu, weil hier die Onlinequellen, auf die zugegriffen wird, exakte Suchbegriffe erwarten.

Folgendes findet keine Ergebnisse, weil hier „Matrix" falsch geschrieben ist:

```
$ geri --title "the marix" --amount=2
```

Mit dem aktivierten *Predator–Mode* findet *Geri* „providerübergreifend" den gesuchten Film.

```
$ geri --title "the marix" --amount=2 --predator-mode
1) Matrix (1999), IMDBid: tt0133093, Provider: TMDBMovie <movie, picture>
Inhalt: Der Hacker Neo wird übers Internet von einer geheimnisvollen Untergrund-
Organisation kontaktiert.  Der Kopf der Gruppe - der gesuchte Terrorist [...]

2) Matrix (1999), IMDBid: tt0133093, Provider: OFDBMovie <movie>
Inhalt: Was ist die Matrix?  Diese Frage quält den Hacker Neo seit Jahren.  Er
führt ein Doppelleben - tagsüber ist er Thomas Anderson und arbeitet in [...]
```

Suche über IMDb–ID: Normalerweise kann nur über die *IMDb–ID* gesucht werden, wenn es die jeweilige Onlinequelle unterstützt. Deswegen funktioniert standardmäßig die Suche bei Providern wie *Filmstarts* oder *Videobuster* nicht. *Libhugin* schafft hier Abhilfe mit einer providerübergreifenden *IMDb–ID*–Suche.

Im folgenden Beispiel findet der Provider *videobustermovie* keine Ergebnisse, weil die Onlinequelle die Suche über *IMDb–ID* nicht unterstützt:

```
$ geri --imdbid "tt0133093" -p videobustermovie
```

Mit dem *Lookup–Mode* funktioniert auch die Suche über *IMDb–ID* bei Onlinequellen, die eine Suche über die *IMDb–ID* nicht unterstützen:

```
$ geri --imdbid "tt0133093" -p videobustermovie --lookup-mode
1) Matrix (1999), IMDBid: None, Provider: VIDEOBUSTERMovie <movie>

Inhalt: Der Hacker Neo (Keanu Reeves) wird übers Internet von einer
geheimnisvollen Untergrund--Organisation kontaktiert.  Der Kopf der [...]

[...]
```

7.1.3 Einsatz von Plugins

Einsatz von Postprocessor–Plugins: Das *Compose*–Plugin ermöglicht dem Benutzer, das Ergebnis nach seinen Bedürfnissen zu komponieren und besitzt die Fähigkeit, das normalisierte Genre mehrerer Provider zusammenzuführen.

Zuerst wird die `movie.mask` angepasst, damit das Genre und das normalisierte Genre zu sehen ist:

```
$ echo "{num}) {title} ({year}), IMDBid: {imdbid}, Provider: {provider}\
\nGenre: {genre}\nGenre normalisiert: {genre_norm} \nInhalt: {plot}" > movie.mask
```

Des Weiteren wird ein benutzerdefiniertes *userprofile* erstellt, welches dem *Compose*–Plugin mitteilt, wie das Ergebnis zusammengebaut werden soll. In unserem Beispiel wird ein Profil erstellt, welches standardmäßig den TMDb Provider nimmt und die Inhaltsbeschreibung durch die vom OFDb–Provider austauscht.

```
$ echo "{'default':['tmdbmovie'], 'plot':['ofdbmovie']}" > userprofile
```

Suche nach dem Film *„Feuchgebiete (2013)"* mit Einsatz vom *Compose*–Plugin und Beschränkung auf die zwei Provider TMDb und OFDb:

```
$ geri --title "feuchtgebiete" -r compose -f userprofile -p tmdbmovie,ofdbmovie -a2
1) Feuchtgebiete (2013), IMDBid: tt2524674, Provider: TMDBMovie <movie, picture>
Genre: ['Komödie', 'Drama']
Genre normalisiert: ['Komödie', 'Drama']
Inhalt: Helen ist eine Herausforderung für ihre Mutter und ihren Vater, die
getrennt leben und geschieden sind.  Trotzdem wünscht sich Helen nichts [...]

2) Feuchtgebiete (2013), IMDBid: tt2524674, Provider: OFDBMovie <movie>
Genre: ['Erotik']
Genre normalisiert: ['Erotik']
Inhalt: Die 18jährige Helen (Carla Juri) hat schon seit ihrer Kindheit
Hämorrhoiden, hat diesen Fakt aber immer verheimlicht, da sie glaubte [...]
```

```
3) Feuchtgebiete (2013), IMDBid: tt2524674, Provider: Compose
Genre: ['Komödie', 'Drama']
Genre normalisiert: {'Erotik', 'Drama', 'Komödie'}
Inhalt: Die 18jährige Helen (Carla Juri) hat schon seit ihrer Kindheit
Hämorrhoiden, hat diesen Fakt aber immer verheimlicht, da sie glaubte [...]
```

Das dritte Resultat in der Ausgabe wurde vom Provider „Compose" generiert, dies ist das komponierte Ergebnis. Hier wurde das normalisierte Genre verschmolzen. Dieses Feature macht das gepflegte Genre in den Metadaten feingranularer und lässt im Beispiel auch besser vermuten, ob ein Film für Kinder geeignet ist oder nicht.

7.2 Libhugin–analyze CLI–Tool Freki

Freki ist für Demonstrationszwecke und das Testen der *libhugin–analyze* Bibliothek entwickelt worden.

7.2.1 Übersicht der Optionen

Folgend zum Überblick der Funktionalität die Hilfe des Tools Freki:

```
$ freki -h
Libhugin--analyzer commandline testtool.

Usage:
  freki create <database> <datapath>
  freki list <database>
  freki list <database> attr <attr>
  freki list <database> analyzerdata
  freki list-modifier | list-analyzer
  freki (analyze | modify) plugin <plugin> <database>
  freki (analyze | modify) plugin <plugin> pluginattrs <pluginattrs> <database>
  freki export <database>
  freki -h | --help
  freki --version

Options:
  -v, --version                    Show version.
  -h, --help                       Show this screen.
```

7.2.2 Erstellen einer Datenbank

Freki erlaubt dem Benutzer, eine *Datenbank* aus externen Metadaten zu generieren. Auf dieser Datenbank kann man folgend mit den Analyzer- und Modifier–Plugins , die *libhugin–analyze* anbietet, arbeiten, um beispielsweise seine Metadaten zu säubern. Nach der Bearbeitung können die *neuen* Metadaten in die externen Metadaten–Dateien exportiert werden.

Folgend eine kurze Demonstration des CLI–Tools.

Erstellen einer Datenbank: Hierzu wird die Helferfunktion (siehe Anhang *Helferfunktion für NFO–Dateien*) verwendet. Im Ordner *movies* befinden sich zwei Filme, die mit dem XBMC mit Metadaten versorgt wurden.

```
$ freki create mydb.db ./movies
```

Datenbank anzeigen: Mit *list* kann der Inhalt der Datenbank angezeigt werden. Die Inhaltsbeschreibung wurde hier wegen der Übersichtlichkeit gekürzt. Wie die Ausgabe zeigt, wurden die Attribute *title*, *originaltitle*, *genre*, *director*, *year* und *plot* eingelesen.

```
$ freki list mydb.db
0) All Good Things (2010)
{'director': 'Andrew Jarecki',
 'genre': ['Drama', 'Mystery', 'Suspense', 'Thriller'],
 'originaltitle': 'All Good Things',
 'plot': 'Historia ambientada en los años 80 y centrada en un heredero de
 una dinastía de Nueva York que se enamora de una chica de otra clase
 [..]',
 'title': 'All Beauty Must Die',
 'year': '2010'}

1) Alien³ (1992)
{'director': 'David Fincher',
 'genre': ['Action', 'Horror', 'Science Fiction'],
 'originaltitle': 'Alien³',
 'plot': 'Después de huir con Newt y Bishop del planeta Alien, Ripley se
 estrella con su nave en Fiorina 161, un planeta prisión. Desgraciadamente
 [...]',
 'title': 'Alien 3',
 'year': '1992'}
```

Analyzer–Data anzeigen: Auflisten der Analysedaten aller sich in der Datenbank befindlichen Filme:

```
$ freki list mydb.db analyzerdata
0) All Good Things (2010)
```

```
{}
1) Alien³ (1992)
{}
```

Da noch nichts weiter analysiert wurde, sieht man hier nur leere Klammern.

Analyzer und Modifier anzeigen: Anzeigen der vorhandenen Analyzer:

```
$ freki list-analyzer
Name:           MovieFileAnalyzer
Description:    Analayze movie files, extract video or audio information.
Parameters:     {}

Name:           LangIdentify
Description:    Analyzes the language of a given plot.
Parameters:     {'attr_name': <class 'str'>}
```

Anzeigen der vorhandenen Modifier:

```
$ freki list-modifier
Name:           PlotLangChange
Description:    Allows to exchange plot to given language.
Parameters:     {'attr_name': <class 'str'>, 'change_to': <class 'str'>}

Name:           BracketCleaner
Description:    Removes brackets e.g. brakets with actor name from plot.
Parameters:     {'attr_name': <class 'str'>}
```

7.2.3 Einsatz von Plugins

Anwenden von Analyzern: Anwendung des *LangIdentify* Plugins auf der *mydb.db*–Datenbank:

```
$ freki analyze plugin LangIdentify mydb.db
```

Betrachten der Analyzerdaten nach der Analyse:

```
$ freki list mydb.db analyzerdata
0) All Good Things (2010)
{'LangIdentify': 'es'}
1) Alien³ (1992)
{'LangIdentify': 'es'}
```

Wie man sieht, wurde hier die verwendete Sprache der Plots analysiert. Das Plugin hat sich in das Analysedaten–Array mit seinem ermittelten Ergebnis eingetragen. In unserem Beispiel *es (espanol)* für eine spanische Inhaltsbeschreibung.

Anwenden von Modifiern: Anwendung des PlotLangChange Modifier–Plugins um die Sprache der Inhaltsbeschreibung von Spanisch auf Deutsch zu ändern:

```
$ freki modify plugin PlotLangChange pluginattrs attr_name='plot',change_to=de mydb.db
```

Betrachten der Metadaten nach Einsatz des Plugins:

```
$ freki list mydb.db
0) All Good Things (2010)
{'director': 'Andrew Jarecki',
 'genre': ['Drama', 'Mystery', 'Suspense', 'Thriller'],
 'originaltitle': 'All Good Things',
 'plot': 'David Marks, Sohn einer reichen New Yorker Familie, verliebt sich
 in die junge Katie McCarthy, die nicht zu seinen Kreisen gehört. Doch dann [...]',
 'title': 'All Beauty Must Die',
 'year': '2010'}

1) Alien³ (1992)
{'director': 'David Fincher',
 'genre': ['Action', 'Horror', 'Science Fiction'],
 'originaltitle': 'Alien³',
 'plot': 'Nachdem Ellen Ripley, die kleine Newt, Soldat Hicks und der
 Android Bishop von LV 426 entkommen sind und sich mit dem Raumschiff USS [...]','
 'title': 'Alien 3',
 'year': '1992'}
```

Wie in dem Beispiel zu sehen ist, wurde die Inhaltsbeschreibung bei den Filmen von der spanischen Version auf eine deutsche Version geändert.

7.2.4 Exportieren der Daten

Die modifizierten Metadaten können nun ins Produktivsystem zurückgespielt werden. Dies geht bei Freki über die *export*–Funktion, hier wird wieder im Hintergrund die *Helferfunktion* (siehe Anhang *Helferfunktion für NFO–Dateien*) verwendet.

Betrachten der Inhaltsbeschreibung der *nfo*–Dateien vor dem Export (Ausgabe gekürzt):

```
$ cat "movies/All Good Things (2010)/movie.nfo" | grep 'plot'
<plot>Historia ambientada en los años 80 y centrada en un heredero de una
dinastía de Nueva York que se enamora de una chica de otra clase social. [...]</plot>
```

Export der modifizierten Datenbank:

```
$ freki export mydb.db
./movies/All Good Things (2010)/movie.nfo
./movies/Alien³ (1992)/movie.nfo
```

Betrachten der Inhaltsbeschreibung der *nfo*-Dateien nach dem Export (gekürzt):

```
$ cat "movies/All Good Things (2010)/movie.nfo" | grep 'plot'
<plot>David Marks, Sohn einer reichen New Yorker Familie, verliebt sich in
die junge Katie McCarthy, die nicht zu seinen Kreisen gehört. [...]</plot>
```

Betrachtet man nun die *nfo*–Dateien der jeweiligen Filme, so sieht man, dass sich hier die Sprache von Spanisch auf Deutsch geändert hat.

7.3 XBMC Plugin Integration

Neben den Kommandozeilentools Geri und Freki wurde konzeptionell ein Plugin für das XBMC (siehe Abbildung: 7.1) geschrieben, welches *libhugin* als Metadatenquelle nutzen kann.

Das XBMC erlaubt es, sogenannte *Scraper* zu schreiben. Diese arbeiten vom Grundprinzip ähnlich wie die Provider von *libhugin*. Das Problem bei dessen Scrapern ist, dass diese vollständig mittels *Regulärer Ausdrücke* innerhalb von *XML*–Dateien geschrieben sind. Dies ist nach Meinung des Autors fehleranfällig, aufwändig und nur schwer lesbar. Des Weiteren sind hier die Möglichkeiten des Postprocessings nur begrenzt umsetzbar.

Die Referenzimplementierung des offiziellen XBMC TMDb–Scrapers hat insgesamt über 600 *lines of code*, recht kryptischer regulärer Ausdrücke (siehe [Link–45] und [Link–46]).

Die Implementierung des *libhugin* Plugins in das XBMC hat an dieser Stelle nur 23 *lines of code* (siehe *XBMC–Scraper–Plugin*). Das liegt daran, dass der *libhugin*–Proxy hier dem XBMC die Daten bereits im benötigten Format über das *Nfo*–Converter–Plugin liefern kann.

7.3.1 Libhugin–Proxy

Da die direkte Integration in das XBMC aufgrund der begrenzten Zeit der Projektarbeit nicht möglich ist, wurde hier der Ansatz eines „Proxy–Dienstes" angewandt. Für *libhugin* wurde mittels dem Microwebframework Flask (siehe [Link–47]) ein minimaler *RESTful* Webservice geschrieben (siehe *Libhugin XBMC Proxy*), welcher über eine eigens definierte API Metadaten an das XBMC liefert.

Libhugin RESTful–API: Der *Libhugin*–Proxy zeigt konzeptionell die Integration von *libhugin* als Netzwerkdienst, welcher eine RESTful–API bereitstellt. Nach dem der *libhugin*–Proxy gestartet wurde, ist es möglich über den Webbrowser auf die RESTful–API über Port 5000 zuzugreifen.

Folgende Bash–Sitzung zeigt die Suche des Films *„Prometheus (2012)"* über den *libhugin*–Proxy. Der Proxy liefert ein für das XBMC formatiertes *XML* zurück:

```
$ curl http://127.0.0.1:5000/search/prometheus
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<results>
    <entity>
        <title>
            Prometheus - Dunkle Zeichen (2012), [tt1446714], Source: TMDBMovie
        </title>
        <url>http://localhost:5000/movie/0</url>
    </entity>
</results>
```

Die implementierte Test–API bietet die folgenden Schnittstellen:

- `/search/<titlename or imdbid>`: Suche nach Film über Titel oder *IMDb–ID*.
- `/movie/<position>`: Zugriff auf einen bestimmten Film im Proxy Cache.
- `/info`: Server Information, welche zeigt ob Postprocessing aktiviert ist.
- `/toggle_pp`: Postprocessing aktivieren oder deaktivieren.
- `/shutdown`: Server herunterfahren.

Die Implementierung des Proxy zeigt, dass es mit relativ wenig Aufwand möglich ist, *libhugin* als „neuen Dienst" für Multimedia–Anwendungen oder auch Movie–Metadaten–Manager zu verwenden.

Hierbei kommt die Flexibilität und Anpassbarkeit des Systems den bisherigen Tools zu Gute. Auf diese Art und Weise lassen sich alle Features, die *libhugin* bietet, in bereits existierende Tools integrieren.

7.3.2 Unterschiede TMDb XBMC und TMDb libhugin

Im Vergleich zum XBMC TMDb–Scraper bietet der XBMC *libhugin*–Scraper (*libhugin*–Provider wurde zum Testen auf TMDb beschränkt) zusätzliche Features.

- Suche über *IMDb–ID* möglich.

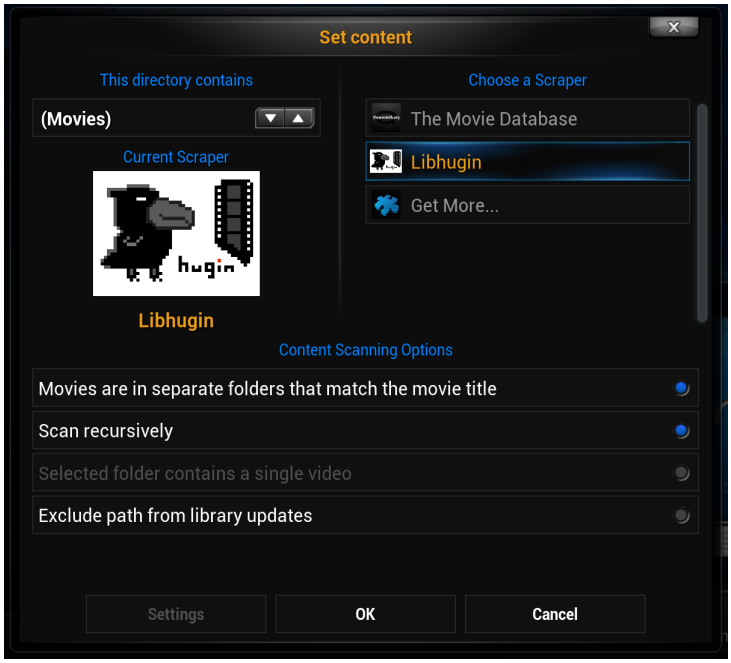

Abbildung 7.1.: Libhugin Scraper–Plugin im XBMC–Scraper–Menü.

- Unschärfesuche möglich, dadurch auch erhöhte Trefferquote.
- Postprocessing, je nach dazugeschalteten Plugin möglich.

Beim Nutzen weiterer Provider sowie Plugins, wie dem Compose Plugin eröffnen sich hier für das XBMC ganz neue Möglichkeiten, seine Metadaten nach den eigenen Wünschen zusammen zu bauen, ohne dabei auf externe Movie–Manager zugreifen zu müssen. Im Prinzip kann *libhugin* hier das komplette Metadatensystem vom XBMC ersetzen.

7.4 Weitere Einsatzmöglichkeiten

Scripting Tasks: Die Einsatzmöglichkeiten sind je nach Szenario anpassbar. Für einfache Anwendungen lassen sich Geri und Freki bereits direkt verwenden.

Ein schönes Beispiel für einen *Scripting–Task* ist das Normalisieren der Ordnerstruktur/Benennung von großen Filmsammlungen.

Hierzu reicht es einfach, die `movie.mask` von Geri anzupassen und ein kleines Bash–Skript zu schreiben. Anpassen der `movie.mask` auf das gewünschte Format:

```
$ echo "{title} ({year}), [{imdbid}]" > tools/geri/movie.mask
```

So schaut das minimalistische rename–Skript aus:

```
#!/bin/bash

for movie in $1/*; do
    old_name=$(basename "$movie")
    new_name=$(geri -t "$old_name" -P --language=en --amount 1 -providers tmdbmovie);
    mv -v "$movie" "$1/$new_name";
done
```

Um eine schlampig gepflegte Filmsammlung zu „simulieren", werden Ordner mit Filmen, die falsch geschrieben sind, erstellt:

```
$ mkdir movies/{"alien1","alien 2","geständnisse","ironman2","iron man3","iron men 1",\
"jung unt schon","marix","oonly good forgives","teh marix 2"}
```

Anschließend wird das Skript auf die schlampig gepflegte Ordnerstruktur angewandt:

```
$ ./rename.sh movies
‘movies/alien1’ -> ‘movies/Alien (1979), [tt0078748]’
‘movies/alien 2’ -> ‘movies/Aliens (1986), [tt0090605]’
‘movies/geständnisse’ -> ‘movies/Confessions (2010), [tt1590089]’
‘movies/ironman2’ -> ‘movies/Iron Man 2 (2010), [tt1228705]’
‘movies/iron man3’ -> ‘movies/Iron Man 3 (2013), [tt1300854]’
‘movies/iron men 1’ -> ‘movies/Iron Man (2008), [tt0371746]’
‘movies/jung unt schon’ -> ‘movies/Young & Beautiful (2013), [tt2752200]’
‘movies/marix’ -> ‘movies/The Matrix (1999), [tt0133093]’
‘movies/oonly good forgives’ -> ‘movies/Only God Forgives (2013), [tt1602613]’
‘movies/teh marix 2’ -> ‘movies/The Matrix Reloaded (2003), [tt0234215]’
```

An diesem Beispiel sieht man, wie gut die Unschärfesuche funktionieren kann. Bei diesem kleinem *Testsample* haben wir eine Trefferwahrscheinlichkeit von 100%.

D–Bus: Eine weitere Möglichkeit, neben dem „Proxyserver–Ansatz", wäre *D–Bus* zu verwenden. *D–Bus* ist ein Framework, das unter Linux zur Interprozesskommunikation verwendet wird. Man kann hier beispielsweise *libhugin* als *D–Bus*–Service laufen lassen und jede andere beliebige Anwendung hätte die Möglichkeit, programmiersprachenunabhängig mit *libhugin* zu kommunizieren.

8 | Zusammenfassung

8.1 Aktueller Stand

Die aktuelle Implementierung zeigt einen modularen Prototypen, der für die gezeigten Anwendungsfälle des Autors gut funktioniert.

8.2 Erfüllung der gesetzten Anforderungen

Die vom Autor gesetzten Anforderungen (siehe Anforderungen an die Bibliothek, 4.1) konnten direkt über die Bibliothek oder durch Schreiben eines Plugins erfüllt werden.

Dennoch gibt es bei einigen Ansätzen Problemstellungen, die nur schwer „gut" umsetzbar sind. Im Fall von *libhugin* wäre das die Normalisierung von Metadaten über mehrere Onlinequellen hinweg. Dies funktioniert im Moment beim Genre mittels statisch gepflegter Abbildungen. Hier wären andere Ansätze, falls möglich, wünschenswert.

Diese Problematik und mögliche andere Ansätze werden in der Bachelorarbeit genauer betrachtet.

8.3 Defizite und Verbesserungen

8.3.1 Erweiterung des aktuellen Pluginsystems

Provider–Plugins: Momentan sind ein multilingualer, ein englischsprachiger und drei deutschsprachige Provider implementiert (siehe Abbildung 6.1). Betrachtet man die Möglichkeiten und Anzahl der Plattformen, ist es wünschenswert weitere Provider zu implementieren.

Die aktuelle Attributstruktur, die von den Providern befüllt wird, richtet sich aktuell an der TMDb–Onlinequelle. Erweiterungen dieser Struktur um neue Attribute sind wünschenswert.

Ein Attribut, das in erster Linie einen Mehrgewinn bringen würde, wäre die „Stimmung". Die Onlinequelle *jinni.com* (siehe [Link–48]) hat ein Attribut „Mood" und noch weitere interessante Attribute

wie „Style", die nach Meinung des Autors einen Mehrgewinn für eine Filmsammlung bringen würden.

Das „Stimmungs"–Attribut könnte man beispielsweise als „tag"–Attribut sogar in die XBMC Metadatenstruktur aufnehmen und hier zusätzlich die Filme nicht nur nach Genre, sondern auch nach Stimmung gruppieren und auswählen.

Postprocessor- und Converter–Plugins: Hier wäre es wünschenswert Converter für allgemein bekannte Metadatenformate zu implementieren, wie beispielsweise für das Windows–Media–Center.

8.3.2 Verbesserungen am Grundsystem

Provider–Priorität: Aktuell wird die Priorität der Provider per Hand gepflegt. Hier wäre ein automatischer Ansatz denkbar und wünschenswert. Eine Idee wäre es, Fehlversuche und Timeouts zu protokollieren und Provider aufgrund dieser zu „bestrafen". Der implementierte OFDb–Provider würde hier wahrscheinlich recht schnell in der Priorität fallen, da dieser sehr oft nicht erreichbar ist. Über diesen Ansatz würde sich zumindest aufgrund der Verfügbarkeit eine Art „Qualität" der Provider bestimmen lassen.

Yapsy: Die aktuell verwendete Bibliothek für das Laden der Plugins wird nur minimal genutzt. Hier wäre es sinnvoll, diese Abhängigkeit komplett aufzulösen und durch einen einfacheren Ansatz auszutauschen.

8.3.3 Weitere mögliche Verbesserungen

Geri und Freki: Die beiden Kommandozeilen–Tools lassen sich noch weiter ausbauen. Das Analysetool Freki beherrscht im aktuellen Zustand noch keine Comparator–Plugins. Weitere denkbare Entwicklungen bei beiden Tools wären automatisierte Analysen der Metadaten und statistische Auswertungen. Des Weiteren wäre ein zusätzliches *ncurses*–Interface wünschenswert, wie es beispielsweise auch beim Mail Client *mutt* genutzt wird. Das würde laut Meinung des Autors die Benutzerfreundlichkeit im Vergleich zum einfachen CLI–Tool erhöhen.

Libhugin–Proxy: Der momentan implementierte Proxy zeigt nur einen konzeptionellen Ansatz und ist aktuell für den Einsatz des XBMC–Plugins geschrieben. Hier wäre eine generische Implementierung als CLI–Tool wünschenswert.

XBMC–Plugin: Das aktuelle XBMC–Plugin kann soweit erweitert werden, dass sich sämtliche *libhugin* Optionen direkt über das Plugin selbst im XBMC konfigurieren lassen.

8.4 Denkbare Weiterentwicklungen

8.4.1 Onlinequellen mit „neuem Wissen" anreichern

Viele Onlinequellen, wie beispielsweise TMDb, haben Schlüsselwörter gepflegt. Diese werden bei TMDb durch die Benutzer der Plattform gepflegt. Oft sind diese jedoch gar nicht vorhanden oder sind zum Teil recht ungenau oder unpassend gepflegt.

Eine Idee wäre hier, die Schlüsselwörter über einen Data–Mining–Algorithmus aus der vorliegenden Inhaltsbeschreibung zu extrahieren. Dies könnte man aufgrund der Architektur von *libhugin* problemlos automatisiert für die ganze Filmsammlung machen und das neu gewonnene „Wissen" in die von der Community gepflegte Plattform zurückfließen lassen.

Ob dies ein möglicher und vom Betreiber der Plattform wünschenswerter Ansatz ist, sollte jedoch vorher mit dem Betreiber der Plattform abgeklärt werden.

8.4.2 Statistische Untersuchung der Metadaten

Der Analyse–Teil der Bibliothek bietet die nicht weiter behandelte experimentelle Comparator–Plugin–Schnittstelle. Die Idee hierzu ist es, Plugins zu entwickeln, die Filmmetadaten verschiedener Quellen untersuchen und miteinander vergleichen. Durch den Vergleich der Metadaten verschiedener Onlinequellen soll die „Qualität" der Metadatenquellen statistisch untersucht werden. Als geeignetes Qualitätsmaß wäre hier beispielsweise die Anzahl der gefundenen Filme oder die Anzahl der in deutscher Sprache gepflegten Metadaten denkbar.

Des Weiteren kann untersucht werden, wie gut sich Filme anhand bestimmter Metadaten mit einander vergleichen lassen und ob man aufgrund von Metadaten Empfehlungen für ähnliche Filme aussprechen kann.

8.4.3 Systemintegration

D–Bus: Neben einem generischen Proxy wäre auch die Implementierung eines *D–Bus*–Service eine gute Idee, um systemweit über eine programmiersprachenunabhängige Schnittstelle auf die Bibliothek zugreifen zu können.

Programmiersprachen–Bindings: Für oft genutzte Sprachen wäre eine Erstellung von Bindings wünschenswert.

8.5 Abschließendes Fazit

Das Projekt zeigt einen Prototyp für die Suche und Analyse von Filmmetadaten. Durch das modulare Konzept lässt sich der Prototyp um verschiedene Onlinequellen und Möglichkeiten der Metadatenaufbereitung erweitern. Der Ansatz mit dem Proxy zeigt, wie sich *libhugin* in bereits existierende Lösungen integrieren lässt. Die beiden Kommandozeilen Tools, *Geri* und *Freki*, eignen sich gut für *Scripting Tasks*. Durch den automatisierbaren Ansatz ist es möglich, sehr große Filmsammlungen von mehreren tausend Filmen in einem vernünftigen Zeitaufwand zu pflegen.

Durch die modulare Erweiterbarkeit lässt sich das System an Bedürfnisse des Benutzers und an zukünftige Anforderungen anpassen.

Zusammenfassend kann gesagt werden, dass das Projekt mit dem „modularen Ansatz" für die vom Autor gestellten Anforderungen erfolgreich war.

A | Helferfunktion für NFO-Dateien

Folgender Anhang zeigt die import/export-Helferfunktion, die von *libhugin-analyze* als Schnittstelle zu den XBMC Metadaten verwendet wird:

```
#!/usr/bin/env python
# encoding: utf-8

import os
import glob
import xmltodict


def attr_mapping():
return {
    'title': 'title', 'originaltitle': 'originaltitle', 'year': 'year',
    'plot': 'plot', 'director': 'director', 'genre': 'genre'
}


################################################################################
# -------------------------- import functions -------------------------------
################################################################################
def attr_import_func(nfo_file, mask):
try:
    with open(nfo_file, 'r') as f:
        xml = xmltodict.parse(f.read())
        attributes = {key: None for key in mask.keys()}
        for key, filekey in mask.items():
            attributes[key] = xml['movie'][filekey]
        return attributes
except Exception as e:
    print('Exception', e)


def data_import(path):
metadata = []
for moviefolder in os.listdir(path):
    full_movie_path = os.path.join(path, moviefolder)
    nfofile = glob.glob1(full_movie_path, '*.nfo')
    if nfofile == []:
        nfofile = full_movie_path
```

```
    else:
        nfofile = os.path.join(full_movie_path, nfofile.pop())
    metadata.append(nfofile)
return metadata


################################################################################
# ------------------------ export functions -------------------------------
################################################################################
def attr_export_func(movie):
mask = attr_mapping()
print(movie.nfo)
with open(movie.nfo, 'r') as f:
    xml = xmltodict.parse(f.read())
    for key, filekey in mask.items():
        xml['movie'][filekey] = movie.attributes[key]
    with open(movie.nfo, 'w') as f:
        f.write(xmltodict.unparse(xml, pretty=True))


def data_export(metadata_dict):
for movie in metadata_dict:
    attr_export_func(movie)
```

B | XBMC-Scraper-Plugin

Folgender Quelltext zeigt die Implementierung des XBMC Plugins. Als externe Schnittstelle wird hier der *libhugin*-Proxy (siehe *Libhugin XBMC Proxy*) verwendet.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<scraper framework="1.1" date="2010-02-22">
<NfoUrl dest="3">
    <RegExp input="$$1" output="\1" dest="3">
        <expression></expression>
    </RegExp>
</NfoUrl>
<CreateSearchUrl dest="3">
<RegExp input="$$1" output="&lt;url&gt;http://localhost:5000/search/\1&lt;/url&gt;" dest="3">
        <expression></expression>
    </RegExp>
</CreateSearchUrl>
<GetSearchResults dest="8">
 <RegExp input="$$5" output="$$1" dest="8">
        <expression></expression>
    </RegExp>
</GetSearchResults>
<GetDetails dest="3">
    <RegExp input="$$1" output="$$1" dest="3">
        <expression></expression>
    </RegExp>
</GetDetails>
</scraper>
```

C | Libhugin XBMC Proxy

Folgender Quelltext zeigt die Implementierung des *libhugin*–Proxy–Servers, welcher das XBMC Plugin mit Daten versorgt.

```
#!/usr/bin/env python
# encoding: utf-8

# stdlib
import re

# 3rd party libs
from flask import Flask
from flask import Response
from flask import request

# hugin
import hugin.harvest.session as HarvestSession
import hugin.analyze.session as AnalyzerSession


SESSION = HarvestSession.Session()
ANALYZER = AnalyzerSession.Session('/tmp/dummydbforanalyzer')

POSTPROCESSING = False
CACHE = {}

app = Flask(__name__)


################################################################################
# ------------------------ flask functions -------------------------------
################################################################################

@app.route('/search/<title>')
def search(title):
    imdbid = re.findall('tt\d+', title)
    # search by imdbid
    if imdbid:
        query = SESSION.create_query(
            imdbid=imdbid.pop(), providers=['tmdbmovie'], language='de'
        )
```

```
    else:
    # search by title
        query = SESSION.create_query(
            title=str(title), fuzzysearch=True,
            providers=['tmdbmovie'], language='de'
        )
    results = SESSION.submit(query)
    template = _read_template('tools/huginproxy/results.xml')
    return Response(
        template.format(results=_build_search_results(results)),
        mimetype='text/xml')

@app.route('/movie/<num>')
def get_movie(num):
    """ Get movie with a specific number. """
    if CACHE:
        result = CACHE[int(num)]
        if POSTPROCESSING:
            postprocess(result)
        nfo_converter = SESSION.converter_plugins('nfo')
        nfo_file = nfo_converter.convert(result)
        return Response(nfo_file, mimetype='text/xml')
    return Response('Cache is empty.', mimetype='text')

@app.route('/stats')
def stats():
    response = 'Postprocessor enabled: {}\nResults in queue: {}'.format(
        POSTPROCESSING,
        len(CACHE)
    )
    return Response(response, mimetype='text')

@app.route('/toggle_pp')
def toggle_pp():
    try:
        global POSTPROCESSING
        POSTPROCESSING = not POSTPROCESSING
    except Exception as e:
        print(e)
    return 'Postprocessor enabled: {}'.format(POSTPROCESSING)

@app.route('/shutdown')
def shutdown():
    print('Shutting down hugin...')
    SESSION.cancel()
    SESSION.clean_up()
    ANALYZER.database_shutdown()
```

```
    print('Shutting down server...')
    shutdown_server()

##############################################################################
# ------------------------ helper functions -------------------------------
##############################################################################

def _build_search_results(results):
    enities = []
    CACHE.clear()
    for num, result in enumerate(results):
        template = _read_template('tools/huginproxy/result_enity.xml')
        enities.append(
            template.format(
                title=result._result_dict['title'],
                year=result._result_dict['year'],
                imdbid=result._result_dict['imdbid'],
                provider=result._provider.name,
                nr=num
            )
        )
        CACHE[num] = result
    return ''.join(enities)

def postprocess(result):
    """ Postprocess example. """
    BracketCleaner = ANALYZER.modifier_plugins('plot')
    result._result_dict['plot'] = ANALYZER.modify_raw(
        BracketCleaner, 'plot', result._result_dict['plot']
    )

def _read_template(template):
    """ Helper for reading templates. """
    with open(template, 'r') as file:
        return file.read()

def shutdown_server():
    func = request.environ.get('werkzeug.server.shutdown')
    if func is None:
        raise RuntimeError('No werkzeug server running.')
    func()

if __name__ == "__main__":
    app.run()
```

D | Projektstatistik (*cloc*)

Folgend eine Projektstatistik erstellt mit dem Tool *cloc*:

```
$ cloc hugin/ tools/
     119 text files.
     117 unique files.
      87 files ignored.

http://cloc.sourceforge.net v 1.60  T=0.51 s (109.5 files/s, 11970.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          49           1220           1171           3540
XML                              5              1              0             57
HTML                             2              9            113             10
-------------------------------------------------------------------------------
SUM:                            56           1230           1284           3607
-------------------------------------------------------------------------------
```

E | Literaturverzeichnis

[1] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[2] Mark Lutz. *Learning python*. O'Reilly Media, Inc., 2013.

[3] James Snell, Doug Tidwell, and Pavel Kulchenko. *Webservice-Programmierung mit SOAP*. O'Reilly Germany, 2002.

[Link–1] http://www.amazon.de/Layer-Cake-Film-Flash-Drive/dp/B001Q3LOTQ. [Stand: 14.4.2014].

[Link–2] http://www.imdb.com/ . [Stand: 15.4.2014].

[Link–3] http://www.xbmc.org. [Stand: 14.4.2014].

[Link–4] http://videoencoding.websmith.de/encoding-praxis/die-richtige-containerwahl.html. [Stand: 20.4.2014].

[Link–5] http://www.gnu.de/documents/gpl-3.0.de.html. [Stand: 20.4.2014].

[Link–6] http://creativecommons.org/licenses/?lang=de. [Stand: 14.4.2014].

[Link–7] http://de.wikipedia.org/wiki/Hugin_und_Munin. [Stand: 20.4.2014].

[Link–8] http://de.wikipedia.org/wiki/Geri_und_Freki. [Stand: 14.4.2014].

[Link–9] https://www.videouniversity.com/articles/metadata-for-video/. [Stand: 14.4.2014].

[Link–10] http://www.cinefacts.de/Filme/Per-Anhalter-durch-die-Galaxis,18240. [Stand: 14.4.2014].

[Link–11] http://www.filmstarts.de/kritiken/37940-Per-Anhalter-durch-die-Galaxis.html. [Stand: 14.4.2014].

[Link–12] http://www.ofdb.de/film/73135,Per-Anhalter-durch-die-Galaxis. [Stand: 14.4.2014].

[Link–13] http://www.themoviedb.org/movie/7453-the-hitchhiker-s-guide-to-the-galaxy?language=de. [Stand: 14.4.2014].

[Link–14] http://www.heise.de/hardware-hacks/artikel/Erste-Schritte-mit-dem-Raspberry-Pi-1573973.html?artikelseite=. [Stand: 14.4.2014].

[Link–15] http://www.popcorn-hour.de. [Stand: 14.4.2014].

[Link–16] http://www.mediaelch.de. [Stand: 14.4.2014].

[Link–17] http://www.junauza.com/2013/01/best-movie-collection-managers-for-linux.html. [Stand: 14.4.2014].

[Link–18] http://wiki.xbmc.org/index.php?title=NFO_files/movies. [Stand: 14.4.2014].

[Link–19] http://dvdxml.com/p/faq/faq.php?0.cat.2.3. [Stand: 14.4.2014].

[Link–20] http://www.imdb.com/genre/ . [Stand: 14.4.2014].

[Link–21] http://www.themoviedb.org/genres. [Stand: 14.4.2014].

[Link–22] http://de.wikipedia.org/wiki/Feuchtgebiete_(Film). [Stand: 14.4.2014].

[Link–23] http://en.wikipedia.org/wiki/Nymphomaniac_(film). [Stand: 14.4.2014].

[Link–24] http://github.com/sahib/glyr. [Stand: 15.4.2014].

[Link–25] http://wiki.xbmc.org/index.php?title=Scraper. [Stand: 14.4.2014].

[Link–26] http://curl.haxx.se. [Stand: 20.4.2014].

[Link–27] http://www.omdbapi.com. [Stand: 20.4.2014].

[Link–28] http://www.filmstarts.de. [Stand: 20.4.2014].

[Link–29] http://libhugin.rtfd.org/en/latest/developer_api/api.html#imovieprovider. [Stand: 20.4.2014].

[Link–30] http://libhugin.rtfd.org/en/latest/user_manual/api.html#creating-a-query. [Stand: 20.4.2014].

[Link–31] http://docs.python.org/3.2/library/concurrent.futures.html. [Stand: 20.4.2014].

[Link–32] http://docs.python.org/3.2/library/shelve.html. [Stand: 20.4.2014].

[Link–33] http://yapsy.sourceforge.net/ . [Stand: 20.4.2014].

[Link–34] http://libhugin.rtfd.org/en/latest/developer_api/api.html#the-build-url-method. [Stand: 20.4.2014].

[Link–35] http://libhugin.rtfd.org/en/latest/developer_api/api.html#the-parse-reponse-method. [Stand: 20.4.2014].

[Link–36] http://libhugin.rtfd.org/en/latest/user_manual/api.html#introduction. [Stand: 14.4.2014].

[Link–37] http://cython.org/. [Stand: 14.4.2014].

[Link–38] http://legacy.python.org/dev/peps/. [Stand: 14.4.2014].

[Link–39] http://github.com. [Stand: 14.4.2014].

[Link–40] https://travis-ci.org. [Stand: 14.4.2014].

[Link–41] http://www-cs-faculty.stanford.edu/ uno/lp.html. [Stand: 14.4.2014].

[Link–42] http://sphinx-doc.org/. [Stand: 14.4.2014].

[Link–43] https://readthedocs.org. [Stand: 14.4.2014].

[Link–44] http://docutils.sourceforge.net/rst.html. [Stand: 14.4.2014].

[Link–45] https://github.com/xbmc/xbmc/blob/master/addons/metadata.common.themoviedb.org/tmdb.xml. [Stand: 14.4.2014].

[Link–46] https://github.com/xbmc/xbmc/blob/master/addons/metadata.themoviedb.org/tmdb.xml. [Stand: 14.4.2014].

[Link–47] http://flask.pocoo.org. [Stand: 14.4.2014].

[Link–48] http://www.jinni.com. [Stand: 14.4.2014].

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Hof, den 29. April 2014 *Christoph Piechula*