Självständigt arbete på grundnivå

*Independent degree project - first cycle*

Datateknik
*Computer Engineering*

Podcast aggregation system
- with cross platform synchronization using Dropbox API

Erik Ström
Abstract

The purpose of this study was to construct an alternative solution to proprietary and licensed products used in the aggregation of podcast information and playback of related audio content. The primary feature of this solution was to offer its users cross platform synchronization of relevant information such as episodic progression and tracking as well as subscriptions in regards to podcasting channels. An application providing podcatching capabilities was developed and its features determined through the process of comparing similar existing solutions. Based on this comparison a Quality Assurance Model (QAM) was created and used as a tool of measuring podcatching capabilities of any media playing software, including the very solution resulting from this study. Questions such as how to find and subscribe to podcast channels was answered through the analysis of syndication feeds, exposing their structure and how its contents may not only be read but also stored to best accommodate requirements deemed to be necessary. The resulting application was subsequently determined, by QAM, to fulfill its main objective of cross platform synchronization. Though, in the end, the application failed to offer enough supporting functionality to be considered as a sufficiently featured podcatching client and thus an adequate alternative to existing products.

Keywords: Aggregation, Podcast, Synchronization, Progression, Tracking, Subscription, Podcatching, Quality Assurance Model, Syndication feeds
Table of Contents

Terminology 6
   Acronyms / Abbreviations 6

1 Introduction 7
   1.1 Background and Problem Motivation 7
   1.2 Overall Aim 7
   1.3 Scope 8
   1.4 Detailed Problem Statement 9
   1.5 Outline 10
   1.6 Contributions 10

2 Theory 11
   2.1 Podcasts and Podcatchers 11
   2.2 Features of Importance 12
   2.3 Syndication of Web Feeds 14
   2.4 Synchronization 16
   2.5 Java Technologies
      2.5.1 Frameworks for Creating Graphical User Interfaces 17
      2.5.2 Working with XML 18
   2.6 Platform Independence 18
   2.7 Dropbox 19

3 Methodology 20
   3.1 Procedures for Analysis 20
   3.2 Procedures for Implementation 22

4 Analysis 23
   4.1 Comparison of Podcast Client Software 23
   4.2 Design of Quality Assurance Model 26
   4.3 Application Requirements
      4.3.1 Functionality and Appearance Requirements 29
      4.3.2 Data Files and Synchronization Requirements 30
   4.4 Tools Selection
      4.4.1 Tools Regarding Syndication Feeds 31
      4.4.2 Tools Regarding OPML Documents 31

5 Implementation 32
   5.1 Working With OPML Documents Using StAX 32
   3
5.2 Parsing RSS 2.0 Feeds Using ROME 33
5.3 Streaming and Downloading Episodes 34
5.4 Progression and Tracking Information 34
5.5 Synchronization using Dropbox API v.2 35
5.6 JavaFX Components 37

6 Results 38
6.1 GUI 39
   6.1.1 Channels List 40
   6.1.2 Channel Information 40
   6.1.3 Episodes 41
   6.1.4 Media Player 42
6.2 Features and QAM Score 43

7 Discussion 45
7.1 Compliance to Application Requirements 45
7.2 Fulfillment of Problem Statements 46
   7.2.1 Main Problem Statements 46
   7.2.2 Supporting Problem Statements 46
7.3 Conclusions 47
7.4 Supplements and Additions 47
7.5 Ethical Considerations 48

References 49

Appendix A: Feature Source Material 51
   PodcatcherMatrix 51
   Podcast Client Feature Comparison Matrix 52

Appendix B: OPML & StAX 54
   Parsing OPML Documents 54
   Building OPML Documents 55

Appendix C: Parsing RSS 2.0 Feeds 56

Appendix D: DownloadTask 58

Appendix E: DropboxManager 59

Appendix F: Project Files 61
Terminology

Acronyms / Abbreviations

API  Application Programming Interface
DOM  Document Object Model
GUI  Graphical User Interface
IETF  The Internet Engineering Task Force
JAXB  Java Architecture for XML Binding
JDK  Java Development Kit
JVM  Java Virtual Machine
OPML  Outline Processor Markup Language
QAI  Quality Assurance Index
QAM  Quality Assurance Model
QAV  Quality Assurance Value
RSS  Rich Site Summary (or Really Simple Syndication)
SDK  Software Development Kit
SAX  Simple API for XML
StAX  Streaming API for XML
UI  User Interface
XML  eXtensible Markup Language
1 Introduction

1.1 Background and Problem Motivation

Podcasts has in a short period of time become one of the most popular mediums of delivering both entertainment and news. It’s free and easily obtainable on all main platforms through various means of distribution, most of which offering a wide selection of podcasting channels of different topics and categories.

The typical consumer of podcasts has access to multiple devices, each used for specific purposes, and most of them supports the necessary capabilities of playing media files associated with podcasts. If a user, having progressed halfway through an episode on a certain device, later wishes to continue its playback, but now on a different device, there are but a few approaches to make this possible. The user could either do this manually by remembering the playback position on the first device and simply skipping the corresponding content on the second device, or he could use some kind of service through which the two devices may communicate and exchange such information. This latter alternative utilizes some kind of data synchronization in order to achieve needed conformation of progressional data, and in the event these two devices run on different platforms such services would offer cross platform synchronization.

While there are no shortage of podcast aggregation software specifk to certain platforms, there doesn’t seem to exist as wide selection of those offering cross platform synchronization. Those that do often relies on proprietary software revolving around a centralized service provided by its creator. Besides often requiring payment fees, these products are usually also accompanied by more or less restrictive licensing which may conceal some, if not most, of its underlying mechanics. Simple features may also be absent from these services, such as the possibility of exporting subscription channels and progression data to a common file format. Should there ever come a time when the user wishes to migrate to another / competing podcatching service, he may thus find himself all too dependent on the current service to warrant the manual work involved in making the switch.

The underlying motivation for this thesis is to determine whether it would be feasible to substitute above mentioned services with a less intrusive alternative which is more in line with the non-commercial spirit of the medium. In order for the author to demonstrate such values, a proof of concept will be made and used as foundation for relevant conclusions.

1.2 Overall Aim

The purpose of this study is to explore the possibilities of performing platform agnostic synchronization in relations to audible podcasts using Dropbox API. The aim is to achieve content synchronization to the extent that not only subscribed channels and finished episodes is up to date, but also the exact progression of unfinished episodes will be retained across systems. A proof of concept will be developed after its comprising features are selected through the analysis and comparison of existing solutions.
1.3 Scope

This study was limited to cover podcatching features and synchronization capabilities of aggregation software based solely on their relation to audio sources, mainly mp3 files, and no effort was made to illustrate how discovered techniques could be used in conformation with other media types. However, as the same rules governing audio should also apply to video files it could be argued that its underlying principles are applicable across many media formats.

Another limitation pertain to the study’s research and the subjects used as its foundation. All research regarding the qualifying features of podcatching products will be restricted to only include pure software solutions, disregarding those dependent upon belonging hardware to fulfill such capabilities. In other words, if any special equipment besides the obviously needed (computer, phone, tablet, etc) is required for either content playback or data management its related product will not be included in this study.

The files associated with podcasts uses ordinary media formats and since just about any media supporting device is capable of playing its contents, further limitations was needed regarding what constitutes as a podcatching client. For the purposes of this study it was determined that the most essential features regarding podcatching capabilities should be comprised by the means of managing channel subscriptions and/or the tracking and progression of belonging episodes.

During the following research it will be assumed that a, not so insignificant, portion of consumers requires a complimentary and non-licensed aggregation system for their podcast consumption. Another assumption is that these users will expect podcatching features equivalent to their currently used solution, in order to even consider another system as an alternative. The capability of synchronizing data across multiple platforms will be determined as the primary feature to which all users will both expect and desire.

In order to demonstrate a proof of concept, by which the main goal of cross platform synchronization, is sufficiently illustrated, at least two applications running on separate platforms was needed. Besides supporting media capabilities, both of these applications would need some way of exchanging information over shared resources in order to synchronize relevant data. For the purposes of this study Java served as implementation language, while the Dropbox API was utilized to accomplish the requirements of synchronization.
1.4 Detailed Problem Statement

There are multiple ways by which a user may consume podcast content as well as keeping track of related subscriptions, tracking and progression. Most of these solutions rely on some form of aggregation software, usually proprietary and restricted by various degrees of licensing. However, there are alternatives to these and other means of available technologies could be utilized to achieve much the same result.

For example, should the main goal be to just consume the contents of each podcast’s episodes across multiple devices one could imagine the user storing relevant media files on his or her personal computer and use some type of distribution service to provide access to these files through data streams. However, such a solution does not necessarily take into account some of the surrounding requirements the user may have, thus involving the inclusion of other means regarding management of episodic data and channel subscriptions.

This study aims to provide the user with an alternative to proprietary cross platform podcast clients that may be used to substitute current solutions as well as liberating the user from the confines often imposed by licensed products. The created solution will need to offer a list of features equivalent to what the user would expect and all aspects regarding synchronization of data will be resolved using Dropbox API. The concrete problems are stated as follows;

Main problem statements:

• How can Dropbox API be utilized in order to achieve cross platform synchronization in regards of aspects such as…
  ▪ ... progression of audio playback?
  ▪ ... subscriptions of podcast channels?
  ▪ ... tracking of episodes (which are finished)?

Supporting problem statements:

• What distinguishing features constitutes podcast client software?
• How can podcasts be found and accessed?
1.5 Outline

**Chapter 1 - Introduction**

Presents an overview of the project, its intended scope and limitations as well as underlying motivation.

**Chapter 2 - Theory**

Brief presentation of the underlying fields of podcasting and synchronization, other related concepts and definitions to be used as base for the rest of the study.

**Chapter 3 - Methodology**

Describes specific approaches for completing the assigned objectives, primarily procedures both regarding the analysis and implementations.

**Chapter 4 - Analysis**

Comparison between select podcast client software based on key features. Design of model to be used as quality assurance for the study's solution. Identification and analysis of the application’s requirements, and the selection of tools needed for their fulfilment.

**Chapter 5 - Implementation**

Ways and means for how solutions is implemented, in relation to chosen tools and frameworks.

**Chapter 6 - Results**

Effective outcomes of implementations.

**Chapter 7 - Discussion**

Evaluation of the resulting outcome and its conformance with requirements satisfaction.

1.6 Contributions

Alice Sveng, fellow student and opponent of this thesis, made valuable contributions regarding the application’s conformance to the Linux platform.
2 Theory

2.1 Podcasts and Podcatchers

The word podcast is derived from the words iPod (media player) and broadcast (distribution of media or messages) [19], and typically refers to audio or video contents which may be consumed using any compatible media player, such as smartphones or computers. Generally, each podcast represents a single part in a larger episodic series aggregated into specific channels to which new content is added periodically by its publisher. The consumer may subscribe to these channels using certain podcast clients, or podcatchers, which pulls relevant data from centralized web feeds, and either downloads or streams the channel’s episodes from its source directories.

These feeds are usually maintained by the distributor of the podcast and stored as Rich Site Summary (RSS) files [21], a derivation of regular eXtensible Markup Language (XML) [7], which contains both general information and metadata regarding the main channel itself as well as its episodes. Updates to such a feed is propagated through the process of web syndication [18], in which changes are pushed to subscribing listeners. Normally a consumer of podcasts wouldn’t subscribe directly to the publisher’s feed, but instead utilize a centralized repository that consolidates and provides access to many of these channels.

Often times, the podcatcher software also provides a repository comprised of many thousands of available channels, and are either maintained directly by the developer or pulled from other sources. Examples of popular podcast clients include; iTunes, Juice and Stitcher.

There are no dedicated file extensions to distinguish podcast episode files from conventional media files. This makes it a bit more complicated to define the exact properties which constitutes podcatching software, since most media players support simple playback of episodes. The characteristics of a podcast client should therefore lie within its managing capabilities in regards to channel subscriptions and tracking of episodes.

A typical Podcatcher would provide functionalities by which the user can subscribe / unsubscribe to channels and, in relation to each channel, track which episodes has been consumed and the progression of not yet finished ones. A common method of managing this information is through the use of OPML (Outline Processor Markup Language) [22] files which is derived from XML and uses outlines to show the hierarchical relationships between its elements.

As its specification reveals [16], the main purpose of OPML is to standardize the structure of documents in order to more easily share subscription information between feed readers, such as Podcatchers, that supports OPML files.

```xml
<opml version="2.0">
  <head>
  </head>
  <body>
  </body>
</opml>
```

**Code fragment 2.1** Bare minimum of an OPML subscription document
In order for the *OPML* document to be deemed as valid it first needs an `<opml>` element as its root, with a required attribute detailing to which version it should conform, and two additional nodes as children; `<head>` and `<body>`, both of which are also required. The `<head>` element contains data regarding the document itself stored as values inside various predefined nodes made available through the specification, none which are required. **Code Fragment 2.1** shows the bare minimum of information a subscriptions document should have. In this example we can see that the *podcast channel* is stored as an outline inside the `<body>` element, which must not be empty and contain atleast one `<outline>` node.

An *outline* should be of an empty element type; a node which do not explicitly declare its ending using closing tags (i.e. `</outline>`). The reason for this is to enforce the rule which says that outlines should not contain any child nodes, such as text or nested elements, but instead keep all information within its *attributes*. A handfull of these *attributes* are defines by the specification of which only *text* is required, but following the recommendations for storing subscription feeds it should also include the attributes *type* and *xmlUrl*, both of which relating to the feed source file; *type* describes its format while *xmlUrl* reveals its location.

### 2.2 Features of Importance

As described in section **2.1 Podcasts and Podcatchers**; simply being able to play podcast content does not qualify a media player to offer *podcatching* capabilitites, and its defining features should instead be found in the provided support for managing subscriptions and episodic data. Further, even though a player may provide required support it may not necessarily define itself as a *Podcatcher* per sé, it's simply one of many services available. Comprehensive lists of what this study defines as *podcatching software* are, because of these aspects, somewhat hard to find.

One tool the author came across during the research was **PodcatcherMatrix**[^1], which is an online tool dedicated to the comparison of different *podcast clients*. The tool provides a convenient side-by-side comparison and considers many aspects including *OS support*, *synchronization* capabilities and list of *features*. However, accessible as it may be the matrix does not fulfill the first criteria of *activity*, as it's basing its comparison on a list of outdated software - most of which has been abandoned or simply has not been maintained for years.

A more current list of *Podcatchers* can instead be obtained from a community maintained article [^11] on *Wikipedia*, to which the main article on *Podcasts*[^2] refer. The list of *Podcatchers* presented in this article will act as a baseline for which further comparisons will be made, but before that the author would like to mention the spreadsheet which is referenced from one of the article's external links.

The *Google Sheet Podcast Client Feature Comparison Matrix*[^3] offers an extensive list of *podcatching* software and a large amount of features by which they are detailed. However, its value as source material for this study is challenged by the fact that no information is provided regarding the meaning of some features or exactly how its data is acquired. A random sample also suggests some discrepancies, where information is either wrong or completely missing.

[^3]: [https://docs.google.com/spreadsheets/d/1c2L14UVH1xtN4iDG4awheLbMgPCQgaKEamUauWs1gps/edit?pref=2&pli=1#gid=0]
Even though both the spreadsheet and the tool *PodcatcherMatrix* were found to lack in quality as foundation to base this study upon, they do provide a combined effort of determining which requirements a *Podcatcher* should fulfill. The author’s own comparison of *Podcatchers* will be partly based on features presented by these sources.

More details regarding both of these sources can be found in *Appendix A: Feature Source Material*.

Table 2.1 shows a compilation of the features which will be used as a quality of measure of *what makes a good Podcatcher* and act as the guideline for features needed by the solution created during this study. Most features are a combination of those found in above sources but renamed to belong inside their respective section. Features added by the author are marked with an asterisk.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SUBSCRIPTION MANAGEMENT</strong></td>
<td></td>
</tr>
<tr>
<td>Channel subscription</td>
<td>As new shows are added, the episodes list is updated.</td>
</tr>
<tr>
<td>Channel discovery</td>
<td><em>Built in support for channel browsing.</em></td>
</tr>
<tr>
<td>* Channel feed URL</td>
<td>Supports user provided URL’s directly to feed.</td>
</tr>
<tr>
<td>OPML support</td>
<td>Support for import / export of subscriptions.</td>
</tr>
<tr>
<td><strong>EPISODE MANAGEMENT</strong></td>
<td></td>
</tr>
<tr>
<td>Episode streaming</td>
<td><em>Shows may be played without the need of download.</em></td>
</tr>
<tr>
<td>Episode download</td>
<td>Support for offline play.</td>
</tr>
<tr>
<td>Episode tracker</td>
<td>*Which episodes have been listened to?</td>
</tr>
<tr>
<td>* Episode progression</td>
<td>Resume from previous progress.</td>
</tr>
<tr>
<td><strong>METADATA</strong></td>
<td></td>
</tr>
<tr>
<td>Channel image</td>
<td><em>Showing image of podcast channel.</em></td>
</tr>
<tr>
<td>Channel information</td>
<td><em>Showing details about the podcast channel.</em></td>
</tr>
<tr>
<td>Episode information</td>
<td><em>Showing details about each episode.</em></td>
</tr>
<tr>
<td><strong>CONTENT PROTOCOLS</strong></td>
<td></td>
</tr>
<tr>
<td>RSS 2.0</td>
<td>Support for RSS feeds.</td>
</tr>
<tr>
<td>Atom</td>
<td>Support for Atom feeds.</td>
</tr>
<tr>
<td>Paged feeds</td>
<td>Support for feed pagination.</td>
</tr>
<tr>
<td>* Archived feeds</td>
<td>Support for feed archiving.</td>
</tr>
<tr>
<td><strong>MISCELLANEOUS</strong></td>
<td></td>
</tr>
<tr>
<td>Personal playlist</td>
<td>User may add episodes to custom playlist.</td>
</tr>
<tr>
<td>Cross device syncing</td>
<td><em>Subscriptions and progress / tracking are synced across platforms.</em></td>
</tr>
</tbody>
</table>

*Table 2.1 podcatching features*
2.3 Syndication of Web Feeds

_Podcast_ content information is communicated through the utilization of _Web syndication_, usually by providing content files conforming to either specification _RSS 2.0_ [9] or _Atom 1.0_ [13], both of which are defined by _The Internet Engineering Task Force (IETF)_ , who's main goal is to improve the quality of the Internet by standardizing best practices in the field ^4^. Both of these syndication formats derive from _XML_, including them to the growing family of technical formats conforming to the _XML 1.0_ specification [12].

The structure of an _RSS document_ is comprised out of a number of tag elements, some of which are required while others are optional. Every document must have the `<rss>` tag as its root element, within which a required attribute specifying its RSS version must be provided. The next required element is the `<channel>` tag which is inserted as a child to the root and contains metadata about both the channel itself and its contents. Required child elements to `<channel>` are `<title>`, `<link>` and `<description>`, while many more are available as optionals such as `<pubDate>` for the date of publication and `<image>` providing information regarding channel image file.

```xml
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Podcast Name</title>
    <link>http://www.podcastname.com</link>
    <description>Info about podcast channel</description>
  </channel>
</rss>
```

*Code fragment 2.2 Bare minimum of an RSS 2.0 document*

Another important element is the `<item>` tag which, in the case of a podcast channel, would represent an episode and contain metadata regarding _title_, _description_, _file location_ and other information. There are no specified limitation to how many items a channel may contain, but there are best practices which should be followed in order to avoid problems.

The example in _Code fragment 2.2_ showed the use of _RSS_ formatation, but could just as easily have used _Atom_ since their similarities makes them interchangable in most situations. A comparison between the two [14] reveals that most of the tags used in _RSS_ have their equivalents in _Atom_, but that _Atom_ has a stricter approach regarding their inclusion, demanding that each _item (entry) _defines elements for _title_, _id_ and _timestamp_ for last update. Also in _RSS_, element values may be of either plain text or escaped _HTML_ but does not provide any means for distinguish these from each other, demanding more involvement from client readers to make this distinction. _Atom_, on the other hand, uses custom _payload containers_ by which element content has its type explicitly labeled, thus releaving this responsibility from client readers.

There are other differences between the two syndication formats, but to the intents and purposes of _podcasting feeds_ they are in most parts equivalent to each other. Both support the inclusion of custom namespaces, giving content creators greater control over the structure of their feeds. A typical

---

^4^https://www.ietf.org/newcomers.html
example of this is the company Apple which provides a certain namespace with iTunes specific elements and attributes.

All of the elements within the feed document constitutes the specified channel's **logical feed**, which in turn is the keeper of its information. The **IETF specification RFC 5005** [11] defines a **logical feed** as "... the complete set of entries associated with a feed.", which in the case of RSS or Atom would mean all of the elements within the document. It’s through the **logical feed** that the syndication of content is directed, by using an **index** comprised of links to all entry elements.

Since there are no specified limits to the number of entries a **logical feed** may contain, problems could arise as the feed grows in content size. Over time as new content is added, increasing the size of the document, it eventually may pose problems regarding its usability. Client code reading the feed's information does so by **parsing** the document in the manner of traversing all of the nodes pointed to by the **logical index**. If all of these nodes are within the same document it may result in slowdowns and inefficient use of resources by the client machine. This problem is especially prominent when it comes to mobile devices which needs to save battery power and has less overall computing power.

Also, as the document gets bigger, its file size increases and thus putting more strain on the network by which various services provides access to the source feed. To combat this problem, many servers apply size restrictions on individual files passing through its gateway, and simply won’t process requests exceeding these limits.

![Picture 2.1 Syndication Feeds](image)

The **RFC 5005** specification addresses these issues and presents two methods to circumvent problems related to content size. A single document containing all of the **logical feed**'s entries are called a **Complete Feed**, and represents the potential problems of content overgrowth. Instead of using **complete feeds** for growing content such as RSS feeds, the specification recommends separating the contents into several smaller documents. The **logical index** would still handle access to individual elements, but now link to its containing document.

There are two main techniques for carrying out this separation; **Pagination** and **Archiving**. A **Paged Feed** divides its content across a sequence of feed pages linked together by **URI**'s defining **first**, **last**, **previous** and **next** page. Each of these pages represents a section of the main logical feed but keeps
its own index of containing elements. This independance eliminates the need of a centralized index as the pages are responsible for their own contents and read in succession. But this also means that there are no guarantees that the logical feed can be fully reconstructed by the client and because of its sequenced layout, new content will always be pushed to the last page.

An Archived Feed handles this separation a bit differently from the paged feed. Content are still divided across individual documents, but are not internally linked to each other like pages. These documents are called Archives and represents a snapshot in the feeds timeline. A subscription document, which always contains the most recent entries available, keeps an index over these archives, making it possible for the client to load contents specific to the chosen archive as needed.

The main difference between paged and archived feeds is that a paged one needs to be reconstructed in its entirety for the logical index to be accessed, while an archived feed only requires the subscription feed to do the same.

2.4 Synchronization
By its general definition [2], synchronization refers to the coordination of separate events to happen in uniformity with each other, as when the conductor of an orchestra directs each instrument to create a harmonius and elegant symphony or when traffic is being controlled by traffic lights. Synchronization is all about order and structure, whereas its counterpart asynchronicity would instead represent disorder and a comparatively more chaotic state.

![Picture 2.2 Types of synchronization](image)

In computer science, this definition is further divided into two distinct but related concepts [3], which are illustrated in Picture 2.2. Process synchronization refers to the act of synchronizing multiple independant processes at certain points and under specific conditions, in order of fulfilling parts of a multiproccessing sequence or to either join or await the execution of others. Usually, these processes have no knowledge of each other and needs to be managed by a controlling part, who handles coordination / execution and access to shared resources. In other words, this manager acts as the conductor or traffic lights.

The other part of the definition refers to data synchronization [20], which concerns data integrity and conformity in regards to multiple copies of the same dataset. The main consideration here,
and of particular interest to this study, are the various strategies of keeping these files synchronized across multiple locations.

There are two main methods for synchronizing files [20], residing in different locations. In a one-way synchronization updated files are mirrored from the source to their target location, ensuring that all copies are direct replicas of the source file. This method of synchronizing files are very straight-forward, where in all target files are determined to be identical to the one source.

A more sophisticated method of updating files is provided by the two-way synchronization, which differs from the former method in that the files are copied in both directions, from source to target and back to source. For this to work, however, synchronization must be based on certain rules, otherwise it would not be possible to determine which file to use as base for the update. Two common strategies of determining this is to either compare the number of edits to the local files or by comparing their timestamps, and simply use the most recent update.

2.5 Java Technologies
Java is one of the most popular programming languages in the world today, offering a large community of practitioners and a comprehensive infrastructure which supports software development in various environments. The language itself has been designed with general-purposes in mind, focusing on flexibility rather than specialisation, and is strictly object-oriented, meaning its use heavily endorses aspects such as encapsulation, inheritance and polymorphism. Java has a mentality of write once, run anywhere which means developers need not compile program code explicitly in conformation to the system’s underlying architecture, but instead pass dynamically through a virtual machine during execution (just-in-time compilation). [23]

This versatility may come at some costs regarding performance, memory and storage but such aspects are often negligible in relation to the system independance offered. As long as the targeted operating system has the capability of running an instance of the Java Virtual Machine (JVM) a successful execution of belonging applications should be guaranteed. However, this may not always be the case and some restrictions do limit the development pertaining some third party libraries which heavily depends on services only offered by certain systems. As long as the developer is cautious to this regard and only uses libraries which conforms to the standards in the official Java Development Kit (JDK), this should not pose much problems. [8]

2.5.1 Frameworks for Creating Graphical User Interfaces
There are quite a few frameworks available for the creation of rich graphical user interfaces (GUI’s), those included in the JDK and other third party solutions. For desktop applications the main contenders are usually three of the officially included ones; SWT, Swing and JavaFX. The first two alternatives are more or less each others equivalent when it comes to appearance and supported functionality. Swing was initially intended to replace the earlier AWT framework by offering more sophisticated components and an improved appearance. However some found it lacking in functionality, and an alternative was made by IBM, called SWT. Over the years these frameworks have matured and offers much of the same features, though based on todays standards both of them look and feel a bit outdated. As a replacement to Swing, Oracle has developed JavaFX which is a framework more suitable to modern software development, as it encourages good OOP practices to separate
concerns regarding implementation using architectural patterns such as Model View Controller (MVC). Besides having a modern look and feel, JavaFX also provides media playing capabilities through its SDK. In most situations, involving desktop application development, JavaFX should be a valid option. [15]

2.5.2 Working with XML

Provided by the JDK is four libraries by which XML files may be read; DOM and StAX for both parsing and building XML documents, SAX which only supports reading, and the newest addition of JAXB which uses annotations to simplify both reading and writing. There are some slight differences between these, but they all require quite a bit of work in order to guarantee quality parsing of complex RSS 2.0 documents with special tag libraries. The main factor to this consists of malformities of many feed providers, and in some cases inconsistencies within the same document. [1]

For the purposes for both parsing and building content to XML files, StAX (Streaming API for XML) seems very competent. It uses memory efficient tools that pulls needed data from bytestreams and thus scales very well even with larger documents. Further, StAX provides two API’s for handling XML data. First there’s the Iterator API through which all available data is traversed in the order it’s pulled from its source. Each cohesive block of data is considered an event and iterating through these occurrences makes for a very elegant solution, particularly in regards to reading XML content.

However, handling data in cohesive blocks does put some restrictions to the way information may be written. Of particular interest here is that each XML element is created by an EventWriter before it’s transferred into the document, and needs to be closed explicitly using a closing tag. This becomes a problem at the Iterator API cannot be used to fulfill the requirements of valid OPML documents, which states that all outline elements should be of empty type.

Luckily, StAX provides the Cursor API for greater control when writing content. By this API information is not written to the document as prebuilt elements, but instead offers a cursor which moves forward as data is written. This makes it possible to customize all aspects of the document’s content and obviously makes it feasible of fulfilling the requirements stated by the OPML specification. [5]

There are many third party libraries more suitable for the task of parsing RSS 2.0 documents. While utilizing one or more of the above mentioned standards, they often provides elegant solutions to common parsing problems. One such library is ROME, which offers both parsing and building capabilities for not only RSS, but also Atom and OPML documents. Using ROME and some of its belonging modules alleviates much of the headaches in the process of parsing content from even the most complex RSS 2.0 documents. [4]

2.6 Platform Independence

Whether a virtual machine, such as the JVM, truly can offer a platform independent environment, really boils down to the definition of what constitutes a platform. In its common definition computing platform [24] refers to technologies capable of executing other software, which would include hardware, operating systems and runtimes. By this interpretation, each of the operating systems supported by the environment would constitute as a separate platform. However, there are some
problems with this definition in regards to applications running on virtual machines such as the JVM. Regardless of the underlying OS these applications still need certain runtime libraries for their execution. The JVM provides the application with the needed runtime and handles all of the communication between the application and services offered by the OS. By this slightly altered perspective, only the JVM itself would have true platform independence while in turn provide OS agnosticism to its containing application.

2.7 Dropbox

Dropbox is a cloud storage service which can be used as an online backup of personal files, or as a file sharing and collaboration tool. Its services are accessible either by a web browser or through its many native applications supporting nearly all available platforms on both desktop and mobile devices. Each registered user receives 2GB of free storage, with payment plans up to 100GB but also a limited possibility of free extensions through various activities. Any type of files may be stored using the service and is secured by 256-bit encryption using the AES (Advanced Encryption Standard) standard.

By installing the Dropbox application a special Dropbox folder appears on the users computer, into which other files and folders can be added freely, restricted only by the designated storage capacity. The contents of this Dropbox folder is then synchronized with the user’s account on the Dropbox servers and pulled from any other devices on which the user may have Dropbox installed. In this way the user will be less likely to lose data and as an added benefit, always have access to synchronized information across devices. [25]

For the purposes of facilitating the integration of Dropbox services into third party applications, Dropbox provides an extensive API which developers may use freely in their software. It’s very intuitive to use and supports the most popular programming languages, including Java. The API is actively maintained and offers a very large community of developers. [5]

---

3 Methodology

While some of the tools to carry out the implementations are decided solely upon the research accounted for in chapter 2 Theory, others need to be based on more substantial data. The aspects of which programming language to use and by which framework graphical components are constructed have no particular significance to the main study, and may therefore be selected based on less strict requirements. The method by which synchronization of data should be performed is much more relevant to the study and the framework used must conform to both the presented scope and aim of the study. Other tools, however, can only be selected based on further analysis of the related field, which is performed by comparing existing solutions in relation to the importance of relevant features.

3.1 Procedures for Analysis

During the research it became clear that there was some complications regarding the acquisition of requirements needed for the project. An absence of quality source material made it challenging to identify features by which a Podcatcher may be distinguished, and so it was deemed necessary to develop a model in an attempt to fill this void. By using this model it should be possible to identify necessary requirements regarding synchronization, functionality and appearance. The resulting model, Quality Assurance Model, will ultimately be used to determine the resulting application’s fulfillment of such requirements.

The specific tools to carry out some of the implementations will be selected based on these requirements. Tools considered by the analysis pertain to the approaches of working with syndication feeds as well as storing and reading subscriptions and episodic data.

![Diagram of Important Features and Podcatchers Comparison](image)

**Picture 3.1 Analysis - overview**

In order to understand which functionalities the resulting application will need to support, their importance and how they should be implemented, it is important that the model’s values correspond with actual demands. And as the Quality Assurance Model will act as the single point of reference by which requirements are determined and later as a measure of quality for the implementations, it’s also important that its development is based on relevant information. To this end, a proceeding comparison of related software is carried out.
The features accounted for in chapter 2.2 Features of Importance will constitute as the foundation upon which a comparison of existing solutions in the field of study is conducted. To be included in this comparison, each solution must first pass the following two criteria:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Whether the application is relatively maintained. Considerations for this criteria is solely based on the application’s last reported update, with dates preceding three years prior to 2017 determined as being too old.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Independance</td>
<td>Whether the application offer its services on more than one platform. No consideration is taken to the environment for which the software was created. An application running on the JVM would have its platform support determined by which operating systems it can run on.</td>
</tr>
</tbody>
</table>

*Operating systems qualified for this criteria are Android, iOS, Windows, OSX and Linux.*

The results of the comparison will firstly yield the frequency of occurrence of each feature, and by that the inherent weight it brings to this study. Secondly, each system will have a score tied to its evaulation, based on how many of said features it supports. The median of these combined evaluation scores will then serve as a Quality Assurance Index (QAI) by which a media playing application may have its podcatching capabilities evaluated by subtracting the QAI from its evaluation score, producing a Quality Assurance Value (QAV). The resulting application from this study will have its podcatching abilities determined by this QAI.
3.2 Procedures for Implementation

In order of fulfilling the study’s main aim of cross platform synchronization, an application able to run in different environments should illustrate as a proof of concept. As accounted for in chapter 2.5 Java Technologies the virtual machine (JVM) upon which all Java desktop applications are executed is fairly agnostic towards operating systems, wherein Java applications may be run in any environment supported by the JVM (which includes all main desktop OS, such as Windows, OSX and Linux). Utilizing Java technologies for the construction and implementation of the application’s functionalities should therefore provide sufficient platform independence to meet this requirement.

Aside from this, Java remains as a very sophisticated development platform. It has one of the world’s largest communities of developers, with a multitude of third party libraries to be freely utilized. But the JDK provided by Oracle also offers extensive development support through its standard libraries as well as a variety of frameworks for creating feature rich GUI’s. For the purposes of meeting requirements regarding both appearance and media playing capabilities, components from the JavaFX framework will be used.

There will be information, pertaining to subscriptions and episodic data, stored in certain files which will need to be synchronized over shared resources while also being accessible from more than one platform. One solution to this would be to offer a custom distribution service which handles all synchronization in personalized accounts, to which a user would be authenticated before using. As this study aims to only prove a concept, this approach was deemed as too complex.

An alternative to creating a dedicated service of our own, would be to use an existing service with the required functionalities. One API of special interest is gPodder Sync [6], which is more or less solely dedicated to the service of synchronizing Podcatching subscriptions and other belonging content. The API is well documented and integration seems fairly straight-forward.

Yet another alternative is to utilize the services of the popular file hosting service Dropbox, which offers APIs for various programming languages. As one of the largest players in its field Dropbox has assigned much of its resources to provide synchronization services to developers free of charge. Its API [17] is extremely well documented, actively maintained and provides both tutorials and examples to cover most situations. Aside from this, Dropbox also have a large community of developers using its API, adding to the availability of answers to questions which undoubtedly will arise during development.

For the purposes of this project Dropbox will be used for all synchronization needs. Though it seems competent enough, gPodder was outshined by the sheer amount of resources available by the community of Dropbox developers.
4 Analysis
This section provides information regarding the acquisition of requirements, their significance and by what means these should be fulfilled. The most important part of the chapter, and the main reason for the analysis, consists of the design of an model, the Quality Assurance Model. This model is intended to assist in identifying distinguishing features of podcasting clients as well as help determining the requirements needed regarding functionalities, appearance and approaches to managing information specific to podcasts.

Before the design of the model can begin, however, we need to determine its comprising properties, namely which features to include and their significance in regards to this study. The foundation to this is provided by the research accounted for in chapter 2.2 Features of Importance, and we analyze these features further by comparing their importance to other existing podcast clients.

4.1 Comparison of Podcast Client Software

<table>
<thead>
<tr>
<th>Name</th>
<th>Latest Release</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amarok</td>
<td>16/08/2015</td>
<td>FreeBSD, Linux</td>
</tr>
<tr>
<td>AntennaPod</td>
<td>25/02/2016</td>
<td>Android</td>
</tr>
<tr>
<td>Banshee</td>
<td>18/03/2014</td>
<td>Linux, macOS, Windows</td>
</tr>
<tr>
<td>Clementine</td>
<td>19/04/2016</td>
<td>Linux, Windows, macOS</td>
</tr>
<tr>
<td>doubleTwist</td>
<td>01/03/2007</td>
<td>Android</td>
</tr>
<tr>
<td>Feedbooks</td>
<td>01/04/2014</td>
<td>Windows</td>
</tr>
<tr>
<td>Foobar2000</td>
<td>07/04/2017</td>
<td>Windows</td>
</tr>
<tr>
<td>iTunes</td>
<td>21/03/2017</td>
<td>Windows, macOS</td>
</tr>
<tr>
<td>Juice</td>
<td>18/07/2006</td>
<td>Windows, macOS</td>
</tr>
<tr>
<td>Liferea</td>
<td>27/12/2016</td>
<td>Linux</td>
</tr>
<tr>
<td>Media Go</td>
<td></td>
<td>Windows</td>
</tr>
<tr>
<td>MediaMonkey</td>
<td>26/02/2017</td>
<td>Windows</td>
</tr>
<tr>
<td>Miro</td>
<td>16/04/2013</td>
<td>Windows, Linux, macOS</td>
</tr>
<tr>
<td>MusicBee</td>
<td>10/03/2017</td>
<td>Windows</td>
</tr>
<tr>
<td>NetNewsWire</td>
<td></td>
<td>macOS</td>
</tr>
<tr>
<td>NewsFire</td>
<td></td>
<td>macOS</td>
</tr>
<tr>
<td>Overcast</td>
<td>21/04/2017</td>
<td>iOS, watchOS</td>
</tr>
<tr>
<td>Podwalk</td>
<td>01/02/2017</td>
<td>iOS, Web</td>
</tr>
<tr>
<td>Radio UserLand</td>
<td>06/09/2005</td>
<td>Windows, macOS</td>
</tr>
<tr>
<td>Rhythmbox</td>
<td>10/09/2016</td>
<td>Linux</td>
</tr>
<tr>
<td>Spotify</td>
<td></td>
<td>Windows, Symbian, Windows Phone, Linux, BlackBerry OS, Android, iOS, Chrome OS, macOS, Sony PlayStation 3, PlayStation 4, and Web</td>
</tr>
<tr>
<td>Stitcher</td>
<td></td>
<td>Android, iOS, Web</td>
</tr>
<tr>
<td>VLC</td>
<td>24/02/2017</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>Winamp</td>
<td>12/12/2013</td>
<td>Windows</td>
</tr>
<tr>
<td>Zune</td>
<td>12/04/2010</td>
<td>Windows</td>
</tr>
</tbody>
</table>

Table 4.1 Noteworthy Podcatching software
Table 4.1 shows the list presented in the article List of podcatchers [11] and, as mentioned in section 2.2 Features of Importance, will be used as the basis for the comparison of podcatching software, which will serve the purpose of determining the requirements the resulting application should fulfill both in terms of functionality and appearance.

Applying the criteria activity and platform independence, as presented in section 3.1 Procedures for Analysis, to above listing produces the results shown in Table 4.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Latest Release</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banshee</td>
<td>18/03/2014</td>
<td>Linux, macOS, Windows</td>
</tr>
<tr>
<td>Clementine</td>
<td>19/04/2016</td>
<td>Linux, Windows, macOS</td>
</tr>
<tr>
<td>iTunes</td>
<td>21/03/2017</td>
<td>Windows, macOS</td>
</tr>
<tr>
<td>Spotify</td>
<td></td>
<td>Android, iOS</td>
</tr>
<tr>
<td>Stitcher</td>
<td></td>
<td>Android, iOS, Web</td>
</tr>
<tr>
<td>VLC</td>
<td>24/02/2017</td>
<td>Windows, Linux, macOS</td>
</tr>
</tbody>
</table>

Table 4.2 criteria applied to podcatching software

Out of the original 25 applications only six remains after applying the criteria. Spotify’s impressive list of supported operating systems, as was presented in Table 4.1, is here shortened to just two, as it only supports podcatching capabilities for Android and iOS. Also, the empty values for Latest Release in regards to both Spotify and Stitcher are ignored, as they are both very much actively maintained.

Table legend - full support = 1, partial support = 0.5, not supported = 0

<table>
<thead>
<tr>
<th>Feature</th>
<th>Banshee</th>
<th>Clementine</th>
<th>iTunes</th>
<th>Spotify</th>
<th>Stitcher</th>
<th>VLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel subscription</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Channel discovery</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Channel feed URL</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OPML support</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Episode streaming</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Episode download</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>Episode tracker</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Episode progression</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Channel image</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Channel information</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Episode information</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RSS 2.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Atom</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Paged feeds</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Archived feeds</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Personal playlist</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cross platform syncing</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Score 11.5 14 12.5 12 11.5 7

Table 4.3 Podcatchers - comparison of features
Keep in mind that this evaluation is purely based on considerations in relation to *podcatching* features, and should have no impact regarding each application’s overall media playing capabilities. In this regard, it seems pretty clear that the otherwise very competent media player VLC underperforms compared to the alternatives. It has the lowest total score and is the only one to not support the most basic features like tracking of episodes, their progress and metadata information. With this in mind, it’s not surprising that more advanced features are missing as well.

Aside from VLC all of the programs score pretty equally, with a slight advantage to Clementine as it offer full support to almost all of the features listed. However, it should be mentioned that a high score does not necessarily equal the greatest value since there are no more complexity to the gradation besides the three values presented. A value of 1 simply means that the feature is fully supported, but considerations to its implementation would perhaps have a different impact to this evaluation. For example; Clementine utilizes third party solutions in order to provide its cross platform synchronization while iTunes, Spotify and Stitcher all use proprietary solutions for the same feature. This is not to say Clementine brings any less value, but it’s reasonable to assume that it has less control regarding its implementation.

The *channel discovery* feature is another example where iTunes, Spotify and Stitcher are using their own proprietary solutions. All three maintains and offer catalogue’s of handpicked podcasts to their users, but iTunes also offer an API through which external programs can pull this information to provide discovery features of their own. Both Clementine and Banshee utilizes sources such as iTunes for these purposes, but in the case of Banshee this feature is only available through plugins.

The use of proprietary solutions to above mentioned features certainly gives more control to the podcatching application. iTunes, Spotify and Stitcher have all expressed a need for controlling which content is consumed through their services so it’s not surprising that none of them offer channel subscriptions by direct URL linking, limiting the user to the selection found through their discovery feature’s. In the same manner do these programs not support OPML import, as these documents could contain subscriptions to channels not accepted by the service. iTunes do have partial support for OPML as it lets the user export their iTunes subscriptions to be used in other Podcatchers.

All systems offers the ability to directly play episodes by streaming as well as download for offline usage. However, Stitcher does this a bit differently from the other alternatives as it won’t let you download single episodes but only those inside playlists which are marked for Offline Mode, in which all content are kept locally.

The most difficult aspect to this comparison concerns the support for different content protocols. While they all fully support RSS 2.0, which seems to be the standard format for the syndication of podcast feeds, none of them seems to offer any Atom support. It could not be verified whether support exists regarding pagination or archiving either, at least not as they are defined in the RFC 5005 specification. However, they all do seem to archive historical data regarding each channel in some way. In the case of iTunes, Spotify and Stitcher this information is stored within the central repository of each systems catalogue, which means that every episode published through the system is available to all users. Neither Banshee nor Clementine store data in this centralized manner, and channel information is instead saved in relation to each client, available to the user for as long as the channel is subscribed to.
4.2 Design of Quality Assurance Model

Based on the comparison of podcatching software we can see how frequently each feature is supported. If we use this frequency as a measure of importance and sort them using this value, we get a list which could act as a baseline for which features should be prioritized. For the purposes of designing the Quality Assurance Model we use the following three types of priorities with belonging frequency values, and a multiplier based on this value;

<table>
<thead>
<tr>
<th>Feature</th>
<th>Priority</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel subscription</td>
<td>HIGH</td>
<td>x3</td>
</tr>
<tr>
<td>Episode streaming</td>
<td>HIGH</td>
<td>x3</td>
</tr>
<tr>
<td>Channel image</td>
<td>HIGH</td>
<td>x3</td>
</tr>
<tr>
<td>RSS 2.0</td>
<td>HIGH</td>
<td>x3</td>
</tr>
<tr>
<td>Personal playlist</td>
<td>HIGH</td>
<td>x3</td>
</tr>
<tr>
<td>Episode download</td>
<td>HIGH</td>
<td>x3</td>
</tr>
<tr>
<td>Episode tracker</td>
<td>MEDIUM</td>
<td>x2</td>
</tr>
<tr>
<td>Episode progression</td>
<td>MEDIUM</td>
<td>x2</td>
</tr>
<tr>
<td>Channel information</td>
<td>MEDIUM</td>
<td>x2</td>
</tr>
<tr>
<td>Episode information</td>
<td>MEDIUM</td>
<td>x2</td>
</tr>
<tr>
<td>Channel discovery</td>
<td>MEDIUM</td>
<td>x2</td>
</tr>
<tr>
<td>Cross platform syncing</td>
<td>MEDIUM</td>
<td>x2</td>
</tr>
<tr>
<td>Channel feed URL</td>
<td>LOW</td>
<td>x1</td>
</tr>
<tr>
<td>OPML support</td>
<td>LOW</td>
<td>x1</td>
</tr>
<tr>
<td>Atom</td>
<td>LOW</td>
<td>x1</td>
</tr>
<tr>
<td>Paged feeds</td>
<td>LOW</td>
<td>x1</td>
</tr>
<tr>
<td>Archived feeds</td>
<td>LOW</td>
<td>x1</td>
</tr>
</tbody>
</table>

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
Feature                  & Priority & Multiplier \\
\hline
Channel subscription     & HIGH     & x3         \\
Episode streaming        & HIGH     & x3         \\
Channel image            & HIGH     & x3         \\
RSS 2.0                  & HIGH     & x3         \\
Personal playlist        & HIGH     & x3         \\
Episode download         & HIGH     & x3         \\
Episode tracker          & MEDIUM   & x2         \\
Episode progression      & MEDIUM   & x2         \\
Channel information      & MEDIUM   & x2         \\
Episode information      & MEDIUM   & x2         \\
Channel discovery        & MEDIUM   & x2         \\
Cross platform syncing   & MEDIUM   & x2         \\
Channel feed URL         & LOW      & x1         \\
OPML support             & LOW      & x1         \\
Atom                     & LOW      & x1         \\
Paged feeds              & LOW      & x1         \\
 Archived feeds          & LOW      & x1         \\
\hline
\end{tabular}
\caption{QAM, feature control values}
\end{table}

If we now apply this formula to the results of Table 4.3, we get the control values upon which the Quality Assurance Model will be based;

We will apply these control values to the evaluation of Table 4.3, but first we need to strip out features deemed absent from all applications. This means that the features Atom, Paged feeds and Archived feeds will not be included in the model.

We may now apply the control values on the remaining features, together with their designated multiplier;
Table legend - full support = 1, partial support = 0.5, not supported = 0

<table>
<thead>
<tr>
<th>Feature</th>
<th>Banshee</th>
<th>Clementine</th>
<th>iTunes</th>
<th>Spotify</th>
<th>Stitcher</th>
<th>VLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel subscription</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Channel discovery</td>
<td>x2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Channel feed URL</td>
<td>x1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OPML support</td>
<td>x1</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Episode streaming</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Episode download</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1.5</td>
<td>3</td>
</tr>
<tr>
<td>Episode tracker</td>
<td>x2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Episode progression</td>
<td>x2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Channel image</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Channel information</td>
<td>x2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Episode information</td>
<td>x2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>RSS 2.0</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Personal playlist</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Cross platform syncing</td>
<td>x2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>QAM score</strong></td>
<td></td>
<td><strong>28</strong></td>
<td><strong>32</strong></td>
<td><strong>30.5</strong></td>
<td><strong>30</strong></td>
<td><strong>28.5</strong></td>
</tr>
</tbody>
</table>

Table 4.6 Podcatchers - feature support by QAM

We now have the means of measuring podcatching capabilities of media applications. The calculated Quality Assurance Index (QAI) is 29.25, the median of all evaluated QAM scores, and will serve as an indication of how well these requirements are fulfilled. Using this assessment as a threshold, only three of the evaluated applications gets a passing grade;

<table>
<thead>
<tr>
<th></th>
<th>Banshee</th>
<th>Clementine</th>
<th>iTunes</th>
<th>Spotify</th>
<th>Stitcher</th>
<th>VLC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QAM score</strong></td>
<td></td>
<td><strong>28</strong></td>
<td><strong>32</strong></td>
<td><strong>30.5</strong></td>
<td><strong>30</strong></td>
<td><strong>28.5</strong></td>
</tr>
<tr>
<td><strong>QAV score</strong></td>
<td></td>
<td>-1.25</td>
<td>2.75</td>
<td>1.25</td>
<td>0.75</td>
<td>-0.75</td>
</tr>
</tbody>
</table>

Table 4.7 Podcatchers - QAV scores

The author would like to take a moment to address the feature cross platform synchronization and its significance to this study. As the study's main aim is to implement a solution to synchronize subscription and episodic data across platforms, this feature obviously holds a greater priority than what it's been evaluated to by QAM. However, this fact should not influence the interpretation during research and does not hold any particular significance regarding the development of this measuring model. The feature will own the highest priority for the entirety of this study, but its QAM control value will remain at its estimation.
4.3 Application Requirements

For the purposes of structuring requirements, they are divided into three sections; Appearance, Functionality and Synchronization, each with their own responsibilities. Appearance should act as a supporter to Functionality, providing visual means by which the user may reach the application’s services. The Synchronization section will manage all file synchronization needed in the application.

![Requirements - overview](image)

**Picture 4.1 Requirements - overview**

Before each section’s requirements are specified, we need to consider the list of features which was produced by the Quality Assurance Model.

![Features & Priority](image)

**Picture 4.2 Features & Priority**

In order of fulfilling this study’s aim of cross platform synchronization, it’s obvious by inspecting Picture 4.2 that the priorities declared cannot have the final saying in regards to which features will be implemented, and in what order. However, this was not the intention of the priority weights to begin with, as its primary purpose was to support the creation of the QAM and its control values. The features listed will still serve as the repository from which our requirements will be determined, and their priorities will be considered as deemed appropriate.
4.3.1 Functionality and Appearance Requirements

Two separate applications are to be created for the purposes of illustrating true platform independant synchronization; one for desktop environments and the other an mobile application. Both applications will need to offer similar functionalities regarding **media playing** capabilities and **Graphical User Interface**. The **media player** needs to support audio playback (**mp3**) and offer means by which start time can be set to values stored from previous progress. The **GUI** should provide the visual means by which the user may choose **Podcast** channel and episodes to play.

![Diagram showing media player and playlist functionalities](image)

**Picture 4.3 Requirements - functionality**

**Picture 4.3** shows an overview of the needed features presented as functionalities in relation to the section to which they belong. **Subscriptions** will be managed by the user, thus the means for both removal of existing and addition of new **channels** are needed. Adding **channels by URL** should be relatively simple to implement, while a **channel discovery** feature would provide more value to the user experience but at the same time adding more complexity.

**Channels** and **episodes** needs to have relevant information presented to the user. Aside from **title** and **description**, **channels** should show an image which’s specified in its feed, while **episodes** should reveal **date** of publication and **duration**. Information regarding **progress** cannot be fetched from the **web feed**, and is instead gathered from a stored location used by the client application.

The user should also have the ability to use at least one **custom playlist**, to which **episodes** from either **channel** may be **added** and **removed**. This list should also support **sorting** or a play order changeable by the user.

Besides these explicitly formulated requirements, there’s also some implicit ones which is dictated by every modern software application. Of particular interest here are aspects which could affect usability and performance. One such aspect would be intrusive **downloads** that's blocking the main thread and thus hindering playback of media. All **downloads** must therefore execute in dedicated threads, preferably using some kind of pool manager to allow multiple **downloads** at the same time.
4.3.2 Data Files and Synchronization Requirements

Picture 4.4 illustrates the most basic requirement needed for file synchronization. It shows a file which is represented by both a local version and its equivalent in the shared resources. At the very least synchronization should be performed at two points during the application’s lifecycle; once at startup and at a later time coinciding with the application’s shutdown.

At these syncing events it will be determined whether a local version exists, and if so request to synchronize over the shared resources and update whichever location has the oldest version. Should a local file not be present, a copy of the shared version will be inquired.

A Synchronization service will be needed for synchronizing local content with its equivalent located in some shared resources. Files will be synchronized based on a last modified timestamp and concurrent access will not be required, as it would add much more complexity to a solution which prime purpose is to simply prove the concept of platform independent synchronization.

In Picture 4.5 we can see which files are going to be needed and how these relates to each other as well as the Synchronization service. Beginning with the subscriptions file, which will be the starting point from which all syndication feeds are loaded.

Within the subscriptions file each subscribed channel will be stored with at least the name of the podcast channel and an URL to its remote syndication feed. XML will be used to store this information.
and by its derivation OPML it will be possible to share this file with other Podcatchers supporting this format. The subscription file uses the Synchronization service to sync with its shared equivalent.

Next we have the syndication feed file, whom's location is known by loading the subscriptions file. Two location points are needed for each feed; one for a local file and one for the remote web feed. It’s from the local file all metadata regarding channel and episodes are parsed. The remote feed is used to update the local file as new contents are detected.

Next we have the channels and episodes, which are all created from the metadata parsed from its relevant syndication feed. Some kind of database will be needed in relation to each channel, in which episodic progression and tracking can be stored. The database would be synced with its equivalent through the Synchronization service.

Lastly we have the Playlist, which also should be synchronized through the main service. This playlist will be created and modified by the user, and by storing its content in the OPML format it would, in theory, be possible for other media players to import it.

4.4 Tools Selection
As the necessary application requirements are identified, it is now possible to determine which specific tools to be used in relation to the implementations of syndication feeds and the means by which subscriptions and episodic data will be handled.

4.4.1 Tools Regarding Syndication Feeds
Based on the features gathered during the development of QAM it’s clear that the standard format for Web syndication feeds are RSS 2.0, atleast in relation to the subjects compared. Supporting both formats would certainly add to the complexity of parsing feed contents, but shouldn’t pose much trouble since both Atom and RSS 2.0 are similar in structure. The only requirement however, is to have a functional parsing system of RSS 2.0 feeds, and for that end there are a couple of alternative solutions.

As accounted for in section 2.5.2 Working with XML the third party library ROME provides elegant solutions in the handling of complex RSS documents, as it offers dedicated modules to aid in parsing tag-specific feeds. Since most channel feeds do include tags specific to itunes much work is alleviated by ROME and the use of its itunes module.

4.4.2 Tools Regarding OPML Documents
For the purposes of reading and writing OPML contents the Streaming API for XML (StAX) will be used, which offers memory efficient tools that pulls needed data from bytestreams and thus scales very well even with larger documents. Because StAX supports both parsing and building XML documents by its API’s and offers memory efficient bytestreams it will scale very well even with larger documents.

As mentioned in section 2.5.2 Working with XML there are alternatives to StAX, provided by the JDK. SAX resembles StAX in that it also uses streams but is not bi-directional, and may only be used to parse content not the building of new documents. DOM could be useful as it enables Random Access to individual parts of a document tree, but does not scale well with large content since it needs
to be loaded in its entirety in memory before anything may be accessed. And lastly, while the XML binding powers of JAXB is nice it also demands a bit more in regards of setting up the prerequisites needed to fully utilize its mapping abilities through annotations. The OPML requirements of this project are simple, and StAX strikes a good balance between potency and efficiency.

5 Implementation

5.1 Working With OPML Documents Using StAX

Two types of OPML files are used by this application, one to contain channel subscriptions and the other for playlist information. These are both very similar in structure and in the procedure in which their contents are parsed and built. As for illustrating its implementation, the subscriptions file will be used as an example, but know that the playlist uses the same type of implementations. This section provides a brief overview, for its detailed implementation see Appendix B: OPML & StAX.

![Picture 5.1 Working with OPML using StAX](image)

The contents of the subscriptions.opml file is structured in conformation to the standards defined by the OPML 2.0 specification [16] and detailed in section 2.1.1 Structure of an OPML Document. As can be seen, the head section is comprised of a single node, `<title>` which is used to differentiate between a subscriptions document from its playlist counterpart. The body section contains information relevant to subscribed podcast channels, each stored as an `<outline>` element together with needed attributes.

Parsing content from the subscriptions file is pretty straight-forward using the Iterator API. An EventReader traverses all of the nodes inside the document and for each occurrence of a detected outline it pulls attribute values related to the event and store these in memory. As the end the outline is detected the EventReader invokes the services of an subscription database, which adds the channel to its list.
As the validity of OPML documents depends on the self-closing properties of each outline element, an event iterator cannot be used since it requires all elements to be explicitly ended with closing tags. Instead we'll utilize StAX’s Cursor API to build each element as needed. This is done by opening an OutputStream to the file, through which related data is added.

### 5.2 Parsing RSS 2.0 Feeds Using ROME

This section provides an high-level overview regarding the construction of channel and episodic objects, by parsing its relevant syndication feeds utilizing the ROME library. Details regarding its low-level implementation are available in Appendix C: Parsing RSS 2.0 Feeds.

Upon application startup, each channel and their belonging episodes is built by parsing the needed information from the channel’s syndication feed, whom location is stored within the subscriptions.opml file.

![Diagram](image_url)

**Picture 5.2 Working with RSS 2.0 using ROME**

A FeedReader object is created and the first task it performs is to peek into the structure of the syndication feed and determines if any special tags are present and which modules are needed to read it. In the event an iTunes tags are detected, it will thus add a module which will help reading tags such as `<itunes:img>` and `<itunes:duration>`. The reader transforms (casts) itself to an instance of FeedInformation as soon as it has pulled necessary data from the feed. This object is then handed to the application, who performs the needed parsing of content and use this to build both Channel and Episode objects together with related information. However, information pertaining to episodic tracking and progression are not included with this FeedInformation object, since these details are specific to each client. Instead each episode retrieves this data from an Episode Database, which is a binary file stored in relation to each channel.
5.3 Streaming and Downloading Episodes

As stated in section 4.3.1 Functionality and Appearance Requirements, the downloading of an episode should be done in a non-intrusive manner using background threads separated from the main event thread, on which the graphical user interface is updated. This is an important aspect to the user's experience, because downloading of large files could otherwise block the main event thread, creating delays as the user navigates the GUI. In order to accommodate this requirement, each download is processed in separate threads, managed by an ExecutorService allowing up to five concurrent downloads. Should the amount of download requests exceed this limit, the service will put overflowing requests into a queue where they will await their turn to process as other downloads finishes and threads are returned to the service pool. Detailed implementations related to this is available in Appendix D: DownloadTask.

A note regarding downloads and streaming. A limitation to the media player provided by JavaFX is that media files stored at secure URL’s cannot be streamed, and must first be downloaded before being played. In the event an audio file cannot be streamed, the user will be notified regarding this through an Alert Message prompting the user to download the audio file to local storage before being able to play it.

Regardless of whether an episode's audio file is available locally or remotely, it does not affect values concerning tracking or progress. Files may be freely downloaded to, and deleted from local storage, without risking any intruding factors upon these aspects. Upon the playing of an episode, but before loading the Media Player with a new audio file, the application will determine whether a local copy of the file exists and if so provide this file to the player, or else use its remote source location.

5.4 Progression and Tracking Information

Progression and tracking refers to podcatching features which are tied to individual episodes and are only relevant to the specific user. Progression relates to the current position of playback time and is stored within its related episode as an instance of java.util.Duration. Tracking is also stored within the episode object but as a boolean value, revealing whether the client has consumed all of its contents.

The data which is represented by both of these values needs to be available throughout the application lifecycle. The presence of such information is determined during startup, and its content loaded together with relevant episodic information pulled from the channel feed. As contents are consumed by using the application, these values will change and needs to be persisted upon application shutdown. For these purposes a database is provided in relation to each channel, into which any and all episodic data will be stored.
**Picture 5.6 Channel relations**

**Picture 5.6** provides an overview of the inner relations between the relevant types of databases and the concrete *channel* objects to which they relate. A central *DatabaseManager* handles all transactional communication and is responsible of both saving and loading persistent information. Two different approaches for storing information can be seen; as a *OPML* document regarding *playlist* information and a binary file relating to *channel* data. Only the implementation of the latter type is detailed in this section, and the concepts of *OPML* implementations are instead described in section 5.1 Working With *OPML* Documents.

*ChannelDB* is the model representation of *serializable* data stored in a *binary file*, and contains a single *HashMap* into which episodic information are kept as key-value pairs of type *Set<String, EpisodeTracking>* for easy accessability. An object of type *EpisodeTracking* contains *progression* and *tracking* information regarding a specific *episode* and is added into the Map with a unique identifier (*GUID*) as key, all of which is provided by the *channel* object. If this map should not be empty at application shutdown, all of its content will be stored in a *binary file* and kept in the same location as other channel specific data, such as *channel image* and the local copy of the *syndication feed*. The contents of any available *channel database* will be included in the process of building *channels* and *episodes* during application startup.

### 5.5 Synchronization using Dropbox API v.2

This section provides an overview of the procedures used to implement synchronization using Dropbox API. For details regarding its implementations see Appendix E: DropboxManager.

In order to use any functionality of the *Dropbox* API, the application must first be registered through an *App Console* interface connected to the developers *Dropbox account*. Through the *App Console* we can set permissions, restricting access to both the application and which folders the application may access on the client's *Dropbox account*. To be able to identify the application through our code, we also need to set up an *App Key*, as well as generating an *Access Token*, which will be used to authenticate the user.
Using the **App Console** a new application is added with the name **PodRunner**, its folder permission is set to **App Folder** which restricts the application to the root folder of **PodRunner** and may not access any files beyond this within the client's **Dropbox account**. **Status** is set to **Development** and only the developer’s **Dropbox account** may be used by the application. Both the **App Key** and **Access Token** are automatically generated for us.

Before any calls to the **API** can be made, an instance of **Dropbox** needs to be present. To do this, the application’s **DropboxManager** creates a **configuration request** by providing a **client identifier** and setting which **language** should be used for **HTTP** responses. What's important here is not the values of these parameters, but that the **configuration** itself will be used to request the main Dropbox instance. **Language** is set to english and as **client identifier** we use the stock identifier from **Dropbox official Java tutorial**. This configuration is passed, together with the **Access Token** which was created in the App Console, as parameters to request a **client Dropbox instance**.

The application’s **DropboxManager** uses a publicly accessible **syncFile(File)** method to determine which file should be updated during synchronization. The local version of the file is passed to the method and a comparison to its shared equivalent is done using each file's last date of **modification** as base value. The location of the shared file is known by parsing the relevant information from the local file and stored in the **path** variable. If either the remote file cannot be found or its date of **modification** is older than that of the local version, the local file will be pushed to the shared folder by invoking the method **uploadFile(...)**. Should instead the local file be determined as older it will be replaced by its remote equivalent by invoking **downloadFile(...)**.

The application uses the services provided by the **DropboxManager** for all of its **synchronization** needs. Both the **subscriptions** and **playlist document** as well as any **channel database** is sent to the **syncFile(File)** method at two occasions during the application lifecycle; once at startup and later as the application closes.

---

6 https://www.dropbox.com/developers/documentation/java#tutorial
5.6 JavaFX Components

JavaFX’s graphic libraries are utilized for building the application’s GUI, and components from its media framework is used to provide media playback capabilities and media control functions. The essence of these components are illustrated in Picture 5.8, in where the main UI is comprised out of four distinct sub sections. In the centre of all this is the MediaControl object which extends upon the layout component BorderPane acting as both the main container for all sub sections, by placing these into distinct areas of the available screen space, and the intermediary of communication between them.

The left section of the MediaControl layout is dedicated to a list of ImageView components, each representing a specific podcast channel except for the top-most item which instead represents the custom playlist. The user may navigate between these channels (and playlist) by simply clicking on its relevant image, upon which the MediaControl object receives a channel selection event and informs both the ChannelInfoPane at its top and the Episode list in its center of what channel is currently selected. The ChannelInfoPane also uses a BorderPane layout to place its components into specific sections, all of which are related to the selected channel. At its top and right sections, the ChannelInfoPane presents information regarding the channel’s title, description and amount of available episodes. At its far left section there are control buttons for both adding and removing channel subscriptions.

The ListView of episodes in the MediaControl’s center section, contains all episodes relevant to the currently selected channel. A BorderPane within each episode determines into which sections related components such as labels and buttons should be placed.
6 Results

![Application Lifecycle Diagram](image)

**Picture 6.1 Application Lifecycle**

**Picture 6.1** illustrates the application’s basic lifecycle, with a primary focus on events occurring during startup and closing. Before the main GUI may be loaded, it needs to have its content both constructed and provided by an preceding **preloader**. This loader is responsible for building object models from information gathered by parsing **channel feeds** and loading data from **database files**. The **subscriptions** file acts as the foundation for this construction, and provides the location to the **syndication feeds** which are used to create each **channel** and their belonging **episodes**. **Tracking** and **progression** data for relevant **episodes** is gathered from the **channel’s database file**, if available.

The **Persistence module** is responsible for providing all of the stored information to the **preloader**, but before doing so it will determine whether any one of these files needs to be updated by its equivalent in the client’s **shared resources**. The reverse to this is performed upon closing the application, where the **persistence module** will determine whether any changes to local files needs to be pushed to its shared counterpart.
6.1 GUI

The preloader uses the JavaFX stock image as background, as can be seen in Picture 6.2, below which is a progress indicator to show activity and a footer conveying that channels are currently being loaded.

In the above illustration of Picture 6.3 is the application's main window, showing its division into four distinct sections. The left part contains a list of every channel currently subscribed to, each represented by its official channel image. Above this list, at the top, resides information regarding the currently selected podcast channel, and in the center section another list presents each episode of said channel. The last section, at the very bottom of the window, is the Media Player, which provides controls for media playback.
6.1.1 Channels List

![Channels List - overview](image)

**Picture 6.4 Channels List - overview**

The channels list actually only consists of the two parts *channels* and *selector*, despite what's conveyed by Picture 6.4. Though it's not a conventional *channel*, as it's not created from the information found in a syndication feed, the *playlist* is by all intents and purposes regarded as a regular *channel* by the application, because it implements the shared interface *ChannelInterface*. The purpose of the *selector* is to show which channel is currently selected.

6.1.2 Channel Information

![Channel Information - default view](image)

**Picture 6.5 Channels Information - default view**

![Channel Information - expanded view](image)

**Picture 6.6 Channels Information - expanded view**

*Pictures 6.5 and 6.6* illustrates the two possible states of which the channel information may be in. The first shows its default view in where only *channel title* and amount of episodes are shown, together with a button by which new *channels* may be added. In the expanded view, this information is extended with *channel description* and an option of removing the channel from subscriptions. In order to toggle this detailed view, the user simply presses the arrow buttons just to the right of the *title*. 
Pressing either the **add** or **remove** buttons triggers the appearance of an alert window, as can be seen in **Picture 6.7**. In the case of adding a new **channel**, the user is asked to provide an **URL** to the relevant feed, and if this is deemed correct the new **channel** will be added to the current **subscriptions**. If the **remove** button is pressed, the user is asked to confirm that the selected **channel** should be removed from the **subscriptions**.

### 6.1.3 Episodes

Each **episode** in the episodes list consists of the five sections illustrated by **Picture 6.8**. A **header** part contains information regarding the episode’s **title**, **date of publication** and **duration** in the format *h:mm:ss*. To the left of the **header** is the **progress** and **tracking** part, represented by an image to convey related value. The **star** icon, to the right of the **header**, is used to indicate whether the **episode** should be included in the **playlist**, and further to the right is an icon showing the current **state** of the **episode** - with regards to its audio file. The last section is located below the others and contains the **episode’s describing text**. As this text may vary in length between different **episodes** it’s restricted to show a single line up until the point where the **episode** is selected and at which point it’s expanded to show its full content.
Illustrated in Picture 6.9 are the possible values for the episode's dynamic sections. Progress and tracking refers to whether the episode is unplayed, has begun or is finished, each of which are represented by a related icon. This value may be changed by right-clicking its icon and choosing a new value from a dropdown list.

The episode state refers to the audio file and whether it's currently loaded in the media player, available for download or, if previously downloaded, may be deleted from local storage. By pressing the icon for playlist inclusion, the episode is either added to the playlist (full star) or removed (star).

6.1.4 Media Player

As shown in Picture 6.10, the Media Player offers a few simple controls for audio playback. The user may play or pause an episode, set its progress by changing the time slider, see current progress and total duration and, finally, control the system's audio volume.
6.2 Features and QAM Score

Out of all the features deemed as necessary by the Quality Assurance Model channel discovery is the only one with no attempt of implementation, which leaves channel feed URL as the only alternative to add new channel subscriptions. Both subscriptions and playlist information are stored as OPML documents, making it theoretically possible to import these into other podcatching software. The playlist is however only partly supported, as it does not provide the means to customize sorting of episodes.

![Podcast & channel files](image)

Each channel has a folder associated with it, as shown in Picture 6.11, in which files are stored for local availability. Inside this folder there will always be a copy of the channel’s RSS 2.0 feed, which is updated if needed upon application start, and a copy of the channel’s official image, whom’s source location is known from the feed. These folders also acts as residence for any downloaded audio files, making it possible to listen to these episodes offline.

The final type of file which may occur in the channel specific folder is the channel database which consists of information relating to episode tracking and progression. This database file, together with the subscriptions and playlist documents, utilizes the application’s services to synchronize over shared resources.
The resulting application of this thesis is called **PodRunner**. Adding **PodRunner** to the list of podcatching software compared in the **Quality Assurance Model**, will produce the result shown in **Table 6.1**.

**Table legend - full support = 1, partial support = 0.5, not supported = 0**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Banshee</th>
<th>Clementine</th>
<th>iTunes</th>
<th>Spotify</th>
<th>Stitcher</th>
<th>VLC</th>
<th>PodRunner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel subscription</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Channel discovery</td>
<td>x2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Channel feed URL</td>
<td>x1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OPML support</td>
<td>x1</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Episode streaming</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Episode download</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Episode tracker</td>
<td>x2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Episode progression</td>
<td>x2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Channel image</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Channel information</td>
<td>x2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>RSS 2.0</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Personal playlist</td>
<td>x3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td>Cross platform syncing</td>
<td>x2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 6.1 Podcatchers - feature support based on QAM**

Based on the **QAM score** provided by the comparison of feature support in **Table 6.1**, we can see that **PodRunner** scores the same as **Stitcher**, well above **VLC** and only slightly higher than **Banshee**. In the other end it’s clear that **Clementine** beats **PodRunner** in this feature score, as does **Spotify** and **iTunes** but not with as much of a margin.

In order to determine whether **PodRunner** provides enough feature support it needs to pass **QAM’s control value QAI (Quality Assurance Index)** which was calculated in section 4.2 Design of Quality Assurance Model to the value 29.25.

<table>
<thead>
<tr>
<th></th>
<th>Banshee</th>
<th>Clementine</th>
<th>iTunes</th>
<th>Spotify</th>
<th>Stitcher</th>
<th>VLC</th>
<th>PodRunner</th>
</tr>
</thead>
<tbody>
<tr>
<td>QAM score</td>
<td>28</td>
<td>32</td>
<td>30.5</td>
<td>30</td>
<td>28.5</td>
<td>19</td>
<td>28.5</td>
</tr>
</tbody>
</table>

**Table 6.2 Podcatchers - QAV scores**

As illustrated from **Table 6.2**, **PodRunner**’s calculated **QAV (Quality Assurance Value)** is a negative value of -0.75 (28.5 - 29.25). As a passing grade would require a positive value, this means that **PodRunner** in its current state does not provide enough quality as required by the model.
7 Discussion

The aim of this study was to research the possibilities of cross platform synchronization using Dropbox API in regards to podcatching properties such as channel subscriptions and episodic tracking and progression. Supporting questions to be answered along the way included means of finding and accessing podcast channels, identifying the properties distinguishing podcast client software, and whether there are particular challenges regarding cross platform synchronization which could be resolved utilizing Dropbox API.

In order to answer these questions and base further conclusions a concept application needed to be developed. The initial research was conducted to account for relevant concepts and defining important terminology. An analysis, supported by this research, was then performed, mainly with the purpose of determining the importance of distinguishing features regarding podcatching clients. The results of this analysis provided enough substance to design the Quality Assurance Model which was used as the foundation to formulate application requirements, both in terms of appearance and functionalities.

Both the construction of the application and the implementation of its features was accomplished using Java technologies with the main intention that its virtual environment would provide sufficient platform independence, as demanded by the thesis primary aim.

7.1 Compliance to Application Requirements

As presented in chapter 4.3 Application Requirements the needed requirements for the application was divided into the three sections appearance, functionality and synchronization. The appearance requirements was aimed to support the functionalities offered by the application by providing suitable GUI controls. The functionality requirements dictated the need for audio playback and the ability of playing files from any time-frame. And lastly, requirements regarding synchronization demanded that file content be synchronized over shared resources and performed at two points during the application’s lifecycle; once at startup and later coinciding with its shutdown.

All of these base requirements was met. Both appearance and functionalities was implemented utilizing the JavaFX framework, which provided the needed graphical components as well as a media player. In regards to synchronization, its requirements was met using the Dropbox API.

The Quality Assurance Model was designed with the main purpose of identifying which features would be needed and how these should be prioritized. And even though every single feature except for channel discovery was successfully implemented during the development of PodRunner, it ultimately failed to provide sufficient podcatching capabilities as required by the model.
7.2 Fulfillment of Problem Statements

All of the main problem statements and its supporting ones have been answered sufficiently within the limits of the present study.

7.2.1 Main Problem Statements

_How can Dropbox API be utilized in order to achieve cross platform synchronization in regards of aspects such as…_

- …progression of audio playback?
- …subscription of podcast channels?
- …tracking of episodes?

These questions were all answered throughout the study, and found to best be fulfilled by means of _file synchronization_ of relevant content using the application’s services to synchronize over shared resources. As the requirements regarding _subscriptions_ deemed it neccessary to provide means for importing and exporting its content, it needed to be stored in a common format readable by other _podcast clients_. By the analysis and comparison of related solutions it became obvious that _OPML_ was the leading standard for both storing and sharing content regarding _podcast subscriptions_.

The storing of _episodic tracking_ and _progression_ was determined to be bound to each client, and thus only relevant to the individual user. The only requirement was that the needed information be saved to retain values, for which purpose a binary format was choosen.

Both of these files is synchronized with their shared equivalent using the application’s dedicated services which are performed at two separate occasions during its lifecycle; once at startup, and later coinciding with the shutdown.

7.2.2 Supporting Problem Statements

_What features distinguishes podcast client software?_ was answered as a result of designing the _Quality Assurance Model_. The features contained withni the model and their deemed importance, constitutes as a sufficient tool to distinguish _podcatchers_ from other media playing software.

_How can podcasts be found and accessed?_ was answered as a result of analysing and comparing existing podcast client software in section _4.1 Comparison of Podcast Client Software_. It was determined that clients often provided some kind of distribution service to discover _podcast channels_. Subscribing to these channels could be done either using these services, or manually by providing an url to its _syndication feed_.

...
7.3 Conclusions
Whichever definition one uses to define a platform, it’s important to consider its relation to the subject matter at hand to determine whether it holds any significant importance to cross platform synchronization. Perhaps the issue lies within this phrasing, and a more suitable wording would instead be; OS agnostic synchronization. However, as stated in section 2.6 Platform Independence, it’s commonly understood that different operating systems constitutes as separate platforms, regardless the runtime environment used by individual applications. It could therefore be safe to assume that the requirement of platform independance is achieved since the solution is executable in all main desktop operating systems; Windows, Linux and OSX.

The QAM score of 28.5 could easily be elevated through some small changes and thus improve this evaluation. As an example, the feature playlist was determined by QAM to have the highest possible weight with a multiplier of x3 towards scoring. PodRunner’s current implementation of this feature was deemed partial, as it’s lacking the capability to custom sort its items, and thus only scores half its maximum value of 3. Simply by adding the mentioned mechanism of sorting would accordingly increase this value by 1.5 resulting in a QAM score of 30, and by that exceed the boundary of 29.25 stated by the QAII leaving a final QAV of 0.75.

7.4 Supplements and Additions
The channel database uses serialized objects to store information in binary format, making it quick and easy to both read and write data. However, this is not an ideal solution as it doesn’t offer any flexibility should any part of the data become corrupted. Furthermore, should the need for stored details change in the future it would require some intermediate adapter to account for older data files - adding complexity where none is actually needed.

A more suitable solution would be to use one of the multiple formats based on XML technologies, JSON or even an embedded relational database such as SQLite. There are many alternatives, but the main aspect to account for is that data moving through remote connections are more susceptible to corruption. A sufficiently satisfying solution would therefore need to be able to detect and handle these defects in a way which does not allow small occurences of corruption to render all of its data useless. Binary files does not fulfill this requirement, since it only offers two possible outcomes; either all data is intact resulting in a correctly reconstructed object, or none of the information is restored because of some defect - no matter the size of its severity.

The application’s preloader is currently underutilized as it has the potential of becoming much more efficient in performing its assigned purpose of loading channels and their belonging episodes. Each podcast channel is built by parsing the information gathered from its related syndication feed, but this is done iteratively by traversing the list of available channels pulled from the subscriptions file. By allowing separate threads to process their own channels, much of this work could be done in parallel to better utilize available resources and decrease startup time which in turn would add more value to the user experience. A pooling service like the one used for downloading episodic audio files would suffice.

Each of these operations would be atomic and isolated up until the event of them being inserted into a shared container, at which point some form of synchronization would be required. This could either be done manually by the use of semaphores to ensure mutual exclusion and subdue the
threat of race conditions, or by using one of the containers offered by the JDK providing such properties (i.e. SynchronizedList).

A similar problem to this exists at the reverse end of the application’s lifecycle. As it’s now, there’s a slight freeze of the main thread upon closing the application, as information is being stored into files and synchronization is performed over shared resources. Having some kind of designated post-loader, these tasks could be processed in a background thread and instead of freezing the main thread it could update the GUI and convey some information regarding its activities.

Both of these potential weaknesses will become more apparent as the amount of channels and/or episodes increases, and could have a significant impact on the overall user experience.

7.5 Ethical Considerations
The author has had great influence regarding the assessment of some portions of the present study, particularly in relation to its analysis. Considerable freedom was taken in regards of estimations during the research of related software, which may have brought presumptions or negative impacts upon developing the Quality Assurance Model. However, because of the non-existence of corresponding studies it was deemed necessary to perform such comparative research. While the author was aware of these aspects, it would be false to assume that such awareness would negate all negative impact.
References

[Accessed: 09 September 2017]

Available: http://www.businessdictionary.com/definition/synchronization.html
[Accessed: 5 September 2017]

Available: https://www.techopedia.com/definition/13390/synchronization-dot-net
[Accessed: 5 September 2017]

Available: https://github.com/rometools/rome
[Accessed: 5 September 2017]

Available: https://docs.oracle.com/javase/tutorial/jaxp/stax/why.html
[Accessed: 5 September 2017]

[Accessed: 01 September 2017]

Available: https://www.w3.org/TR/xml-c14n
[Accessed: 26 May 2017]

Available: https://docs.oracle.com/javase/specs/jls/se8/docs/toh/index.html
[Accessed: 09 September 2017]

Available: https://cyber.harvard.edu/rss/rss.html
[Accessed: 16 April 2017]

[Accessed: 16 April 2017]

[Accessed: 5 May 2017]

Available: https://www.w3.org/TR/REC-xml
[Accessed: 5 May 2017]

[Accessed: 8 May 2017]

[14] InterTwingly, "RSS 2.0 And Atom 1.0 Compared", 2008. [Online]
Available: http://www.intertwingly.net/wiki/pie/Rss20AndAtom10Compared
[Accessed: 8 May 2017]

[Accessed: 09 September 2017]


Appendix A: Feature Source Material

PodcatcherMatrix
This online tool\(^7\) offers the ability of comparing 21 different podcast clients, as seen in Table A-1, based on 52 distinct features, showcased in Table A-2.

<table>
<thead>
<tr>
<th>Armangil</th>
<th>Azureus</th>
<th>Doppler</th>
<th>FireAnt</th>
<th>gPodder</th>
</tr>
</thead>
<tbody>
<tr>
<td>iTunes</td>
<td>Juice</td>
<td>MyPodder</td>
<td>NewsFire</td>
<td>Nimiq</td>
</tr>
<tr>
<td>PlayPod</td>
<td>PPR</td>
<td>QuickNews</td>
<td>RadioTracker</td>
<td>Raggle</td>
</tr>
<tr>
<td>rcFeedMe</td>
<td>RSSRadio</td>
<td>SmartFeed</td>
<td>Synclosure</td>
<td>The Spellcatcher</td>
</tr>
<tr>
<td>Ziepod</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table A-1 PodcatcherMatrix - client software**

<table>
<thead>
<tr>
<th>General Features</th>
<th>Operating Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Windows 98</td>
</tr>
<tr>
<td>Last Release</td>
<td>Windows XP</td>
</tr>
<tr>
<td>Author</td>
<td>Mac OS X</td>
</tr>
<tr>
<td>URL</td>
<td>Linux</td>
</tr>
<tr>
<td>Free and Open Source</td>
<td>Windows Mobile</td>
</tr>
<tr>
<td>License</td>
<td>Palm OS</td>
</tr>
<tr>
<td>License Cost / Fee</td>
<td>Programming Language</td>
</tr>
<tr>
<td>Languages</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Common Features</th>
<th>Sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>Sync to iTunes</td>
</tr>
<tr>
<td>Notifications</td>
<td>Sync to PSP</td>
</tr>
<tr>
<td>OPML Import</td>
<td>Sync to MediaPlayer</td>
</tr>
<tr>
<td>OPML Export</td>
<td>Sync to iPod</td>
</tr>
<tr>
<td>Playlist Formats</td>
<td>Integrated Burning to CD</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Podcatching</th>
<th>Media</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule Downloads</td>
<td>MP3</td>
</tr>
<tr>
<td>Resume Partial Downloads</td>
<td>MPEG-4</td>
</tr>
<tr>
<td>Limit Download Size</td>
<td>Real</td>
</tr>
<tr>
<td>ID3 Editing</td>
<td>QuickTime</td>
</tr>
<tr>
<td></td>
<td>Windows Media</td>
</tr>
<tr>
<td></td>
<td>DivX</td>
</tr>
<tr>
<td></td>
<td>Xvid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transport Protocols</th>
<th>Content Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>http</td>
<td>RSS 2.0</td>
</tr>
<tr>
<td>https</td>
<td>Atom</td>
</tr>
<tr>
<td>http auth</td>
<td></td>
</tr>
<tr>
<td>bittorrent</td>
<td></td>
</tr>
<tr>
<td>Proxy Server</td>
<td></td>
</tr>
<tr>
<td>feed://</td>
<td></td>
</tr>
<tr>
<td>podcast://</td>
<td></td>
</tr>
<tr>
<td>pcast://</td>
<td></td>
</tr>
<tr>
<td>itpc://</td>
<td></td>
</tr>
<tr>
<td>itms://</td>
<td></td>
</tr>
</tbody>
</table>

**Table A-2 PodcatcherMatrix - feature list**

\(^7\) http://www.podcatchermatrix.org/
Podcast Client Feature Comparison Matrix

This spreadsheet offers a comparison of 65 different podcast clients, as seen in **Table A-3**, based on 62 distinct features, showcased in **Table A-4**.

The spreadsheet is locked to View only mode, so for those wanting to determine its value for themselves the code presented in **Code fragment A-1** will import the sheet into a new Google Sheet;

```excel
=IMPORTRANGE("1c2L14UVH1xtN4iDG4awheLbMgPCQgaKEamJauWs1gps","'Podcast Client Feature Comparison Matrix'!A2:BV")
```

**Code fragment A-1** importing data into new google sheet

**Code fragment A-2** shows how the resulting sheet may be queried for information;

```excel
=QUERY(Podcatchers!A:BV, "SELECT* WHERE (G = 'Cross-platform' AND L > date '2013-12-31') ORDER BY L"),1)
```

**Code fragment A-2** query for data based on criteria

<table>
<thead>
<tr>
<th>ACast</th>
<th>Amarok</th>
<th>AntennaPod</th>
<th>Apple TV</th>
<th>Banshee</th>
</tr>
</thead>
<tbody>
<tr>
<td>BashPodder</td>
<td>BeyondPod</td>
<td>bPod</td>
<td>BringCast</td>
<td>Castro</td>
</tr>
<tr>
<td>Chakouat</td>
<td>Clementine</td>
<td>DoggCatcher</td>
<td>Downcast</td>
<td>Dreambox</td>
</tr>
<tr>
<td>foobar2000</td>
<td>FRITZ!Fon MT-F</td>
<td>gPodder</td>
<td>GuttenPodder</td>
<td>hpodder</td>
</tr>
<tr>
<td>i Podcast</td>
<td>iCatcher!</td>
<td>Instacast</td>
<td>Instacast 4</td>
<td>Instacast Cloud</td>
</tr>
<tr>
<td>Juice</td>
<td>MediaMonkey</td>
<td>Miro</td>
<td>Mocast</td>
<td>Musicbee</td>
</tr>
<tr>
<td>Overcast</td>
<td>Overcast.fm</td>
<td>P</td>
<td>Cast</td>
<td>Player.FM</td>
</tr>
<tr>
<td>Pocket Casts</td>
<td>PocketCasts</td>
<td>Pod Wrangler</td>
<td>Podcast Addict</td>
<td>Podcast Bandit</td>
</tr>
<tr>
<td>Podcast Brain</td>
<td>Podcast Lounge</td>
<td>Podcast Picker</td>
<td>Podcast+ Pro</td>
<td>Podcaster 7</td>
</tr>
<tr>
<td>Podcasts</td>
<td>PODCASTS! PRO</td>
<td>Podcast</td>
<td>Podcatcher</td>
<td>Podcatchr</td>
</tr>
<tr>
<td>Poddi Podcatcher</td>
<td>Poddie</td>
<td>PodGrasp</td>
<td>Podkicker</td>
<td>Podkicker Pro</td>
</tr>
<tr>
<td>podscout</td>
<td>Podsnatcher</td>
<td>Rhythmbox</td>
<td>RSSRadio Premium</td>
<td>SlapDash Podcasts</td>
</tr>
<tr>
<td>SleekCast</td>
<td>Squeezebox</td>
<td>uPod Podcast</td>
<td>Volksempfänger</td>
<td>WPodder</td>
</tr>
</tbody>
</table>

**Table A-3 Podcast Client Comparison Matrix - client software**

---

Footnote:

8 [https://docs.google.com/spreadsheets/d/1c2L14UVH1xtN4iDG4awheLbMgPCQgaKEamJauWs1gps/edit?pref=2&pli=1#gid=0](https://docs.google.com/spreadsheets/d/1c2L14UVH1xtN4iDG4awheLbMgPCQgaKEamJauWs1gps/edit?pref=2&pli=1#gid=0)
### Availability
- Last Update
- Latest Version
- Minimum OS Version
- UI Language
- Download Link
- Free
- Light Version
- Open Source
- Price

### Features Support 1
- Sleep Timer
- Radio Mode
- Playback Speed
- Gapless Playback
- Volume Boost
- Bookmarks
- Personal Playlist
- Personal Smart Playlist
- Predefined Smart Playlist
- Support for Video Podcasts
- Opus Support
- AirPlay Support

### Platform
- Tablet Version
- Landscape Support
- Inline Browser
- Push for new Episodes
- Custom Subs. Sorting
- Alphabetically Subs. Sorting
- Touch Controls
- OS-Built in Play/Pause Controls

### Miscellaneous
- Developer aware of Podlove
- Individual Podcast Settings
- Add Ep. to Playlist w/o sub
- App-specific subs. URI scheme

### Common Features
- Paged Feeds
- Chapter Mark Support
- MP3 Chapter Mark Support
- Auto-Chapter
- Chapter Duration
- Chapter Start Timecode
- Chapter End Timecode
- Chapter Pictures
- Shownotes for downloaded episodes
- Shownotes w/o download / playback

### Features Support 2
- BitTorrent
- Skip Into
- Play Currently Downloading Episode
- Multiple Parallel Downloads
- Episode Streaming
- Password protected Podcasts
- Automatic Downloading Options
- Disk Space Mgmt
- Atom Feed Support

### Syncing and Import / Export
- Cross Device Syncing
- Sync Technology
- Import from Music.app Library
- OPML Export
- OPML Import

### Social Services
- Flattr (auto)
- Podcast Directory
- Comment Episodes via AudioBoo
- Comment Episodes via Text
- Social Sharing

---

**Table A-4 Podcast Client Comparison Matrix - feature list**
Appendix B: OPML & StAX

```xml
<?xml version="1.0" encoding="UTF-8"?>
<opml version="2.0">
<head>
  <title>Podcast Subscriptions</title>
</head>
<body>
<outline title="StarTalk Radio" text="StarTalk Radio" type="rss"
<outline title="The Nerdist" text="The Nerdist" type="rss"
  xmlUrl="http://nerdist.libsyn.com/rss"/>
<outline title="The Bugle" text="The Bugle" type="rss"
  xmlUrl="http://feeds.thebuglepodcast.com/thebuglefeed"/>
</body>
</opml>
```

*Code fragment B-1 Structure of a subscriptions document*

Parsing OPML Documents

```java
// Setup a new event reader
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLEventReader eventReader = inputFactory.createXMLEventReader(new FileInputStream(file));

// Read the opml document
String title = null, source_rss = null;
while (eventReader.hasNext()) {
    XMLEvent event = eventReader.nextEvent();

    if (event.isStartElement()) {
       StartElement startElement = event.asStartElement();

        // If we have an outline element, we create a new channel
        if (startElement.getName().getLocalPart().equals("OUTLINE")) {
            title = event.asStartElement().getAttributeByName(new QName(TITLE)).getValue();
            source_rss = event.asStartElement().getAttributeByName(new QName(URL)).getValue();
        }
    }

    // If we reach the end of an outline element, we add it to the list
    else if (event.isEndElement()) {
       EndElement endElement = event.asEndElement();
        if (endElement.getName().getLocalPart().equals("OUTLINE"))
            subscriptionsDB.addSubscription(title, source_rss);
    }
}
eventReader.close(); // close the stream
```

*Code fragment B-2 Parsing subscriptions file using StAX Iterator API*
Building OPML Documents

```java
// Create stream writer
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(new FileOutputStream(file), "UTF-8");

// Insert xml version and encoding
writer.writeStartDocument("UTF-8", "1.0");
writer.writeStartElement("opml");
writer.writeAttribute("version", "2.0");

// Write head
writer.writeStartElement("head");
writer.writeStartElement("title");
writer.writeCharacters("Podcast Subscriptions");
writer.writeEndElement();

// Write body
writer.writeStartElement("body");

// Write outlines
Map<String, String> subs = subscriptionsDB.getSubscriptions();
String title = null, url = null;
for (Map.Entry<String, String> entry : subs.entrySet()) {
    title = entry.getKey();
    url = entry.getValue();
    writer.writeEmptyElement("outline");
    writer.writeAttribute("title", title);
    writer.writeAttribute("text", title);
    writer.writeAttribute("type", "rss");
    writer.writeAttribute("xmlUrl", url);
}

writer.writeEndElement(); // end body

// End document
writer.writeEndElement(); // end OPML
writer.writeEndDocument();

writer.flush();
writer.close(); // close stream
```

*Code fragment 5.2 Building OPML file using StAX Cursor API*
Appendix C: Parsing RSS 2.0 Feeds

```java
// Get URL to source feed from file
URL feedSource = channelFile.toURI().toURL();

// Create InputReader
SyndFeedInput input = new SyndFeedInput();
SyndFeed feed = input.build(new XmlReader(feedSource));
Module module = feed.getModule("http://www.itunes.com/dtds/podcast-1.0.dtd");
FeedInformation feedInfo = (FeedInformation) module;

// Read channel metadata
Channel channel = new Channel();
channel.setTitle(feed.getTitle()); // title of Podcast
String formattedDescription = feed.getDescription()
    .replaceAll("\<.*?\>\", "").trim();
channel.setDescription(formattedDescription);
channel.setDate(feed.getPublishedDate().toString());
channel.setImage(feedInfo.getImage().toString());
channel.setLanguage(feedInfo.getLanguage());

// Get a reference to the relevant channel database
ChannelDB channelDB = DatabaseManager.getInstance().
    getChannelDatabase(feed.getTitle());

// Create episodes list
List<Episode> episodes = new ArrayList<>();
for (Object entry : feed.getEntries()) {
    Episode tmp = parseItem((SyndEntry) entry);
    String progress = db.getProgressOfID(tmp.getGuid());
    tmp.setProgress(progress);
    episodes.add(tmp);
}
channel.setEpisodes(episodes);
```

**Code fragment C-1 Parsing channel metadata using ROME**
// Reading episodic metadata
private Episode parseItem(SyndEntry entry) {
    Episode episode = new Episode();
    episode.setGuid(entry.getUri());
    episode.setTitle(entry.getTitle());
    String formattedDescription = entry.getDescription().getValue().replaceAll("\<\/*\?>", "").trim() // remove html tags
        .split(LINE_SEPARATOR)[0] // remove redundant information
    ;
    episode.setDescription(formattedDescription);
    episode.setLink(entry.getPublishedDate());
    Duration duration = EntryInformation.getModule(entry.getEnclosures().get(0)).getDuration();
    String durValue = (duration == null) ? "00:00:00" : duration.toString();
    episode.setDuration(durValue);
    return episode;
}
Appendix D: DownloadTask

```java
URL url = new URL(sourcePath);
URLConnection connection = url.openConnection();

in = new BufferedInputStream(url.openStream());
out = new FileOutputStream(targetPath);

int fileSize = connection.getContentLength();
byte[] data = new byte[1024];
int count;
double sumCount = 0.0;

while((count = in.read(data, 0, 1024)) != -1) {
    out.write(data, 0, count);
    sumCount += count;
    if(fileSize > 0)
        episode.updateDownloadProgress(sumCount / fileSize);
}
```

*Code fragment D-1 base code*
Appendix E: DropboxManager

```java
DbxRequestConfig config = new DbxRequestConfig("dropbox/java-tutorial", "en_US");
DbxClientV2 client = new DbxClientV2(config, ACCESS_TOKEN);

// Code fragment E-1 creating dropbox client

public void syncFile(File localFile) {
    String path = "/"+localFile.getParent().substring(2)+"/"+localFile.getName();
    try {
        Metadata meta = client.files().getMetadata(path);
        FileMetadata fileMeta = (FileMetadata) meta;
        // Should file in dropbox be updated?
        if(fileMeta.getClientModified().before(new Date(localFile.lastModified()))) {
            uploadFile(localFile, path);
            // .. or the local one?
        } else {
            downloadFile(localFile, path);
        }
    } catch (GetMetadataErrorException e) {
        if (e.errorValue.isPath() && e.errorValue.getPathValue().isNotFound()) {
            // File not found, upload local one
            uploadFile(localFile, path);
        } else {
            e.printStackTrace();
        }
    } catch (DbxException | IOException ex) {
        ex.printStackTrace();
    }
}
```

**Code fragment E-2 Synchronize files**

```java
// Update local file
private void downloadFile(File targetFile, String sourcePath) {
    try (OutputStream out = new FileOutputStream(targetFile)) {
        FileMetadata metadata = client.files().downloadBuilder(sourcePath)
            .download(out);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

**Code fragment E-3 Update local file**
// Update remote file
private void uploadFile(File sourceFile, String targetPath) {
    try (InputStream in = new FileInputStream(sourceFile)) {
        FileMetadata metadata = client.files().uploadBuilder(targetPath)
            .withMode(WriteMode.OVERWRITE).uploadAndFinish(in);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
Appendix F: Project Files

The project’s source code can be found in the project’s GitHub page. The project files, together with all belonging dependencies, are managed and built using Apache Maven, making it agnostic towards IDE’s. Besides source files, this repository offers links to executable versions of the PodRunner application, both as a runnable jar and as standalone installation packages for both Windows and OSX. The excellent javafx-maven-plugin is used to package these installers and has been configured to include all needed dependencies, even a runtime of the JVM. Thus, by using the installers there is no requirement for JDK to be installed on the underlying system.

---

9 https://github.com/Eirikir/PodcastApp_Desktop
10 https://maven.apache.org/
11 https://github.com/javafx-maven-plugin/javafx-maven-plugin