Bachelor Thesis

December 19, 2017

ChangeAdvisor

A tool for Recommending and Localizing Change Requests for Mobile Apps based on User Reviews

Alexander Hofmann

of Zurich, Switzerland (11-916-863)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

ChangeAdvisor

A tool for Recommending and Localizing Change Requests for Mobile Apps based on User Reviews

Alexander Hofmann

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Alexander Hofmann, alexander.hofmann@uzh.ch

URL: https://bitbucket.org/alexander_hofmann/changeadvisor

Project period: 10.07.2017 - 10.01.2018

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

The success and final outcome of this project, its implementation, and this thesis, has been one of the most rewarding and challenging experiences of my life. It would not have been possible without the help of a few people. In the following I would like to express my gratitude to those that have supported me, in many different ways, in order to reach the conclusion of the Bachelor thesis. First, I wish to present my special thanks to all those remarkable people who have contributed directly to the realization of my thesis. After that, I wish to express my appreciation and gratitude, to those who indirectly allowed me to reach such an important milestone in my life: the Bachelor's degree.

First and foremost, I owe a debt of gratitude to my advisor, Dr. Sebastiano Panichella, for offering me this interesting and challenging assignment, for his enthusiasm and trust placed in me, giving me the freedom to tackle this project as I saw fit. The door to his office was always open, whenever I needed his help or guidance. He consistently allowed this paper to be my own work, but steered me in the right direction whenever I needed it. I could not have imaged having a better advisor and mentor for my Bachelor's project.

Besides my advisor, I am sincerely grateful to Prof. Dr. Harald C. Gall for giving me the opportunity to work together with the Software Evolution & Architecture Lab at the University of Zurich.

Finally, I must spend a few words to express my very profound gratitude to those people, who provided me with unfailing support and continuous encouragement throughout my entire academic career. To my wonderful girlfriend, Tramy, whose love and support has helped me more than she will ever realize. And to my mother, Elizabeth, for always believing in me and supporting me in my endeavor, even through the difficulties and setbacks, since the beginning of my career.

This accomplishment would not have been possible without all of them.

Abstract

User feedback plays a paramount role in the development and maintenance of mobile applications. The experience an end-user has with an app, is a key concern when creating and maintaining a successful product. Consequently, developer teams need to incorporate opinions and feedback of end-users in the evolutionary process of their software, in order to meet market requirements. However, existing app distribution platforms provide limited support for developers to systematically filter, aggregate, and classify user feedback to derive requirements. Moreover, manually reading each user review to gather useful feedback is not feasible, considering the sheer amount of reviews popular apps have received and continue to receive day after day. Even then, the gathered information is restricted to user reviews, and no systematic way exists to link user feedback to the related source code components to be changed, a task that requires an enormous manual effort and is highly error-prone.

To fill this void, Palomba *et al.* [PSC+18] introduced *ChangeAdvisor*, an approach able to cluster user reviews, useful for software maintenance tasks, into topics, in order to recommend developers, which source code entities to change. This already greatly simplifies the work for the developer, as it is not necessary anymore to sift through the reviews, divide them in valuable or valueless feedback, then try to figure out, which source code component is affected from the proposed changes. However *ChangeAdvisor*, until now, existed only as a *Proof of Concept*, which was limited in terms of extensibility and maintainability, as well as in functionality.

Thus, this thesis implements *ChangeAdvisor* as a library, in order to support future extensions of the approach, as well as a client-server application, to allow developers to fully leverage the power of the information contained in user feedback.

Zusammenfassung

Das Benutzerfeedback spielt eine wichtige Rolle bei der Entwicklung und Wartung von Mobile Apps. Die Erfahrung, die ein Endbenutzer mit einer App hat, ist einer der wichtigsten Punkte bei der Entwicklung und Wartung eines erfolgreichen Produkts. Aus diesem Grund müssen Entwicklerteams Meinungen und Feedbacks von End-Usern in den Entwicklungsprozess ihrer Software einfliessen lassen, um den Marktanforderungen gerecht zu werden. Bestehende App Stores bieten Entwicklern jedoch nur begrenzte Unterstützung, um Benutzerfeedbacks systematisch zu filtern, zu aggregieren, und zu klassifizieren. Ebenso fehlt es an Techniken, um aus diesen Feedbacks Anforderungen an das Produkt herzuleiten. Ausserdem ist das Lesen jener Feedbacks nicht praktikabel, wenn man die Menge an täglichen Bewertungen für beliebte Apps berücksichtigt. Selbst wenn die Menge an Bewertungen kein Problem wäre, sind die gesammelten Informationen auf Benutzerbewertungen beschränkt und es gibt keine Möglichkeit die Benutzerfeedbacks mit den zu ändernden Quellcode-Komponenten systematisch zu verlinken. Dies wäre nur mit enormen manuellem Aufwand zu ermöglichen und wäre sehr fehleranfällig.

Um diese Lücke zu füllen, führte Palomba *et al.* [PSC+18] *ChangeAdvisor* ein. Der *ChangeAdvisor* Ansatz erlaubt es, Benutzerfeedbacks, welche für die Softwarewartung nützlich sind, nach Themen zu gruppieren. Dank diesen Themengruppen sehen die Entwickler welche Quellcode-Komponenten verbessert werden sollten. Dadurch wird die Arbeit der Entwickler bereits vereinfacht, da es nicht mehr notwendig ist die Reviews manuell zu durchsuchen, zu unterteilen und den Quellcode-Komponenten zuzuordnen.

Bisher war *ChangeAdvisor* nur ein *Proof of Concept*, welcher in Bezug auf Wartbarkeit, Erweiterbarkeit und Funktionalität sehr eingeschränkt war. In dieser Bachelorarbeit wird *ChangeAdvisor* deshalb als Softwarebibliothek implementiert, um zukünftige Erweiterungen zu unterstützen. Zusätzlich wurde eine Client-Server App implementiert, welche es Entwicklern erlaubt die gesamten Informationen aus Benutzerfeedbacks zu nutzen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The importance of end-user perception of a product cannot be overstated. The experience users have with an application, are, almost entirely, decisive for the success or failure of a project. Modern software development has moved away from a traditional paradigm such as *waterfall*, in favor of *continuous delivery*, were new updates are available every other week and new features are added over time. This is particularly true for mobile apps. Indeed, the advent of mobile apps and app stores, made it easier than ever to manage such release schedules. With app stores, came also the possibility for users to voice their opinion regarding an application. This is, usually, in the form of free text and a numerical score. Behind these two simple points, hides a literal trove of information. Indeed, through text, a user can describe the problems he had, whether he likes the app or not, or express their wishes for new options and features, while the rating concisely and numerically represents the opinion the users has. It also contains other useful information, either through the app store itself, such as review date, and software version, as well as, sometimes, smartphone or tablet used. Developers, can and should take advantage of this information as a backlog for development.

1.1 Motivation

There are difficulties in exploiting this information, however. The various app stores, usually provide only limited support for sorting and aggregating this data. Additionally, it is not always feasible to read through all reviews, considering popular apps may receive hundreds of reviews a day.

In order to solve this problems, recent studies have come up with new approaches to classify user reviews. Most of these, however, do not consider sentence structure and semantics, relying solely on keywords [PSC+18]. Additionally, many of these studies only attempted a classification or summarisation of reviews. It would be of great benefit to developers, if there were a tool to help them in identifying the source code components that need change.

In this respect, Palomba *et al.* [PSC+18] introduced *ChangeAdvisor*, an approach able to cluster user reviews into topics, and then link these topics to the source code components in need of change. Such an approach, greatly reduces the amount of time needed to analyze user feedback, and makes it less error-prone. However, *ChangeAdvisor*, existed only as a *Proof-of-Concept* (PoC), which is not ready yet for continuous use, and lacks many features necessary, in order for it to be adopted by software developers.

Thus in this work, we aim to redesign and implement *ChangeAdvisor* from the ground-up, keeping in mind aspects, such as maintainability and extensibility, in order to lay the foundations, for future work to expand on this tool, integrating newer and more advanced approaches, and functionality.

1.2 Thesis Outline

This thesis is a report on the design and implementation of the `ChangeAdvisor` approach [PSC+18] as a Java application. This Chapter provided an overview of the context and motivation for this work.

Chapter 2 presents the research done in the field of *App Store Review Mining*, which is the field this works is situated in. Also an overview of the original `ChangeAdvisor` paper is given at the end of the Chapter (2.1.2).

In Chapter 3, we review the theory behind the newest iteration of `ChangeAdvisor`. Here the high-level concept of the tool is described, including how each step of the process works.

Chapter 4, goes into detail of the actual implementation of the tool: this includes the software's architecture, as well as how each step is implemented, how scheduling is implemented, and what tools and libraries are used, in order to achieve our goal.

Chapter 5, presents a qualitative evaluation of this work. Given the open-ended nature of this implementation, the review is done by mainly presenting the non-functional requirements for `ChangeAdvisor`, and showing how the various features of this tool, not only achieve the main goal, but they even go beyond, showing how it enables many future prospects for extension.

Finally, Chapter 6, concludes this thesis, briefly reviewing the work done, and most importantly, presenting some of the possibilities for future extension.

Chapter 2

Related Work

In this section, I will briefly summarize the research done in the context of *App Store Review Mining* and *Linking* of information contained in free-form text to source code.

2.1 App Store Mining for Software Engineering

User reviews provide an invaluable source of data regarding user perception of a product. Indeed, Mudambi and Schuff showed in 2010, that there is a strong correlation between user reviews and purchasing decisions of products on amazon.com [MS10]. The advent of App Stores, such as *Google Play Store* and Apple's *App Store* in 2008, made it easier than ever to access software and review apps. This is made evident by the download and review numbers, presented in Table 2.1, of popular applications. We can see that in the brief span of seven years, an app like *Facebook* had between 1 and 5 billions unique installs, considering *Android* alone, and over 72 million users commented on their problems, their praise, and their wishes for new features.

It is no wonder then, that researchers started to come up with ways to capitalize on this wealth of, albeit unstructured, information to extract useful structured data. This brings us to Harman *et al.* [HJZ12] which introduced the concept of *App Store Mining* as a form of *Mining Software Repositories*. In their work, they showed a strong correlation between an app's rating and download rank. Their approach can roughly be described in the following three steps: (i) extract raw data from the App store, (ii) parse data extracting all attributes regarding price, ratings, and description, and (iii) use data mining to extract relevant features from gathered description.

Table 2.1: Release year, number of unique installs and reviews for popular apps on the *Google Play Store* (standings Nov. 2017[2]).

App name	Release Year	Unique Installs (in millions)	Reviews (in millions)
Facebook	2010	1'000 - 5'000	72
Facebook Messenger	2011	1'000 - 5'000	49
Whatsapp	2010	1'000 - 5'000	60
Pokemon GO	2016	100 - 500	9

[2]Google Play Store statistics, as of November 2017:
`https://play.google.com/store/apps/details?id=com.facebook.katana&hl=en`
`https://play.google.com/store/apps/details?id=com.facebook.orca`
`https://play.google.com/store/apps/details?id=com.whatsapp`
`https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo`

The result of which are statistics regarding technical, customer, and business aspects of each app.

Many researchers focused on the analysis of user reviews, as can be seen in the survey by Martin *et al.* [MSJ+16]. In their work, the authors summarized and identified the following seven subfields of *App Store Analysis*.

- API Analysis: studies that extract feature information by analyzing app APKs and/or source code together with non-technical data [MSJ+16].
- Feature Analysis: studies that extract features from sources, other than source code and requirements. In this subfield, information is extracted by analyzing descriptions, API usage, manifest files, decompiled source strings, categories, and permissions [MSJ+16].
- Release Engineering: studies that extract feature information from app releases. It analyzes how the content changes between releases, the effects on user perception, and the releases strategies adopted by developers (e.g. app category diversification, free vs. paid, update rollouts) [MSJ+16].
- Review Analysis: studies that extract feature information by examining user feedback [MSJ+16].
- Security: studies that explore various security aspects of apps, such as faults in code leading to vulnerabilities, malicious apps, permissions, plagiarism, privacy, and update behaviour of users [MSJ+16].
- Store Ecosystem: studies that explore "the ecosystem of each app store and the differences between them" [MSJ+16].
- Size and Effort Prediction: studies that predict size or effort based on the list of functionalities of an app [MSJ+16].

Of particular interest for this work are: (i) *API Analysis*, and (ii) *Review Analysis*.

2.1.1 API Analysis

API analysis refers to techniques that attempt to extract feature information by examining open-sourced source code and combine such features with non-technical data (e.g. ratings). Martin *et al.* [MSJ+16] identified 4 subfields: (i) *API usage*, (ii) *Class Reuse and Inheritance*, (iii) Faults, (iv) *Permissions and Security*.

API Usage. Azad [Aza15] mined apps from Google Play Store and F-droid, in particular, method calls to suggest (i) related calls from the VCS history, (ii) similar calls found in Stack-Overflow posts, (iii) possible copyright and license violations with open-source software. Tian *et al.* [TNLH15] proposed a case study encompassing over 1400 apps, in which they identified 28 possible factors that might determine the rating of apps. Through this study they determined that, in high-rated apps, 17 of the 28 characteristics proposed, deviate significantly from low-rated apps. The most influential factors they found were: size of an app, use of promotional images, and target SDK [TNLH15].

Faults. Many studies have concluded that there is a link between Fault and change-prone APIs and low rated apps [MSJ+16]. Indeed, Linares-Vasquez *et al.* [BLVBC+15] studied how the use of change- and fault-prone APIs impacted the success of apps. Their results indicate that there is an inverse relation between the usage of such APIs and their rating.

Syer *et al.* [SNAH15] proposed to use the degree of platform dependence (or independence) as an indicator of software quality. The authors argue that system specific APIs are rapidly evolving, and thus often introduce defects. Through their study they found, that the more defect-prone source files are those that depend more heavily on the platform, thus degree of platform dependence could be used to prioritize QA resources.

Khalid *et al.* [KNH16] proposed the use of popular static analysis tool `FindBugs` to find possible relations between warnings and user rating. The authors found that the categories "bad practice", "internationalization", and "performance", appeared more frequently in low rated apps, and that seemed to reflect user perception.

2.1.2 Review Analysis

Review analysis refers instead to techniques that attempt to extract features by analyzing user reviews, by employing *Natural Language Processing*, *Sentiment Analysis*, and *Topic Modelling*. The goals of this analysis are multiple, the most important of which are mentioned here:

- Review classification
- Determining factors that affect user feedback
- Extraction of bug reports and feature requests
- Review prioritization
- Review summarisation

These subfields are of particular interest for this work, since they are the foundation it is based on: by taking the work done in both categories and merging it into a tool capable of correlating *reviews* and *code*, as seen in Chapter 3.

In the context of *App Store Analysis*, Pagano *et al.* [PB13] showed the need for automated tools for review analysis. Indeed they proposed a case study which showed, that user feedback contains important information for developers, which helps in software maintenance tasks. However manually analyzing reviews, especially for popular apps, requires a considerable time investment. This is caused by the fact that user feedback is unstructed, and that the quality and content varies greatly between users [PB13]. Additionally, users often do not posses the necessary terminology to precisely describe what they would like and what their problems are. Their results show the necessity for software support to categorize, analyze, and track user feedback [PB13].

Khalid *et al.* [KSNH15] ran a study over a set of over 6000 low-rated user reviews of 20 iOS apps. They discovered 12 types of user complaints, the most frequent of which were *functional errors*, *feature requests*, and *app crashes*. The authors argue that developers should devote particular attention to user feedback, since negative reviews affect sales more than good review [KSNH15]. By categorizing the user feedback, they believe developers can better plan their limited quality assurance (*QA*) resources, by focusing on higher prioritized reviews. Their study results also stress the importance of establishing trust and meeting expectations with their user base [KSNH15].

Review Classification and Summarisation. As mentioned before, *Review classification* is a subfield of *App Store Mining* that concerns itself with automatically being able to classify reviews based on the textual features contained inside feedback. In this regard, Chandy and Gu [CG12] mined over six million reviews from the iOS App Store. After manually labeling a subset of the reviews as spam or not spam, they trained an unsupervised classifier to automatically categorize reviews taking into account average user ratings, and number of apps rated. Continuing from here, Ha *et al.* [HW13] manually examined reviews from the Google Play Store to determine

Table 2.2: ARdoc categories [PDSG+16]

Category	Description	User Feedback Example
Information Giving	Sentences that inform or update users or developers about an aspect related to the app	"This app runs so smoothly and I rarely have issues with it anymore"
Information Seeking	Sentences related to attempts to obtain information or help from other users or developers	"Is there a way of getting the last version back?"
Feature Request	Sentences expressing ideas, suggestions or needs for improving or enhancing the app or its functionalities	"Please restore a way to open links in external browser or let us save photos"
Problem Discovery	Sentences describing issues with the app or unexpected behaviours	"App crashes when new power up notice pops up"
Other	Sentences do not providing any useful feedback to developers	"What a fun app"

whether users were talking about the privacy and security aspects of an app. From their sample, they determined that only 1% of the user reviews mentioned these aspects.

Chen *et al.* [CLH+14] proposed `AR-MINER`, a framework for App Review Mining, capable of discerning informative reviews from the non-informative. Their approach works by (i) filtering out noisy and irrelevant reviews, (ii) using topic modelling to cluster reviews together by topic, (iii) ranking reviews, and (iv) presenting the ranked and clustered reviews to the developers. They argue that their approach can facilitate the work of sifting through user feedback for big review sets.

Panichella *et al.* [PDSG+15] presented a system, capable of analyzing user reviews to support software maintenance and requirements evolution. The authors presented a taxonomy to classify reviews, merging three techniques: (i) *Natural Language Processing*, (ii) *Text Analysis*, and (iii) *Sentiment Analysis*. Later the authors further developed this idea and introduced `ARdoc` [PDSG+16], a review classifier, which this work heavily relies on. The authors divided reviews in 5 categories, presented in Table 2.2. They argue that `ARdoc` could be useful in combination with topic modelling techniques, for example, dividing all reviews into categories before clustering, could provide a higher degree of cohesion inside the clusters.

In recent work, Sorbo *et al.* introduced `SURF` (Summarizer of User Reviews Feedback) [DSPA+17], a tool "able to (i) analyze and classify the information contained in app reviews and (ii) distill actionable change tasks for improving mobile applications" [DSPA+17]. It is capable of summarizing thousands of user reviews to generate a task list of recommended software changes [DSPA+17].

ChangeAdvisor

Finally, Palomba *et al.* introduced `ChangeAdvisor` [PSC+18], a novel approach, built on the work of Panichella *et al* [PDSG+16], able to cluster user reviews into topics, and link these topics to source code entities. Additionally they developed a *Proof of Concept* (PoC) to demonstrate their approach.

The approach works as follows [PSC+18]:

- User feedback is tagged based on predefined categories
- Source code and feedback are preprocessed

- Feedback belonging to the same category is clustered, to group together similar user needs
- Determine code components related to user change requests

User Feedback Classification. In order to classify user feedback, `ChangeAdvisor` leverages `ARdoc` [PDSG+16]. Given user feedback, this classifier is capable of identifying the category the review belongs to. Table 2.2 shows an overview of the various categories.

Source Code and Feedback Preprocessing. In order to remove noise from both source code and feedback, the data set is pre-processed. The result of this step is a bag of words that will be used as input for the following steps.

User Feedback Clustering. Similar user needs are clustered in the following step. This is done for two reasons: (i) linking single reviews does not result in good accuracy results, and (ii) multiple users will probably have the same problems or needs that refer to the same change request. By clustering, this approach can achieve better results, all the while giving the developer a better overview of the users' perception and needs.

Recommending Source Code Changes. Clustered change requests (user feedback referring to the same task) are linked against the preprocessed source code component, to find out which set of classes need to be changed, according to user feedback. Links are computed using a similarity metric. Only pairs of code components and clusters that achieve a minimum similarity threshold are considered as related. The output is then a list of tuples $(cluster, component, similarity)$, *links*, representing the reviews cluster, the source component, and their similarity metric. Concretely, each tuple is the class that needs to be changed in order to fulfil the users' request.

Limitations. The `ChangeAdvisor` tool at the moment exists solely as a PoC. It consists of a mix of Java code and Python Scripts, with some glue code in the form of Shell scripts inside a docker container. It takes the path to the source code and a review set and runs the process. As such, it is not yet flexible enough to be of real use for a team of developers, and is hard to maintain. Ideally, in order to become productive with `ChangeAdvisor`, a developer would need to be able to configure it, according to its need. Such needs would include, at a minimum, the possibility to automatically import user reviews, according to a schedule, and the possibility to import source code, ideally from a *Version Control System* (VCS). With these functions in place, the tools could be further developed to allow a developer to explore the reviews and the links to the source code by playing around with time intervals, running the linking only on certain categories, or even test different clustering methods or similarity metrics.

Combining all these functions with a UI, would be of great help to developers, allowing them to gather more insight into their code, and explore the perception a user has of their work in many different ways.

This work, then, has as a goal, to use the `ChangeAdvisor` [PSC+18] PoC, to lay the ground work for a new implementation of `ChangeAdvisor`, that is first of all configurable and maintainable. This tool should contain all of the features of the original work, plus new features such as a UI for the configuring of the parameters, scheduling of long running tasks, and a way to better explore user reviews through the use of visuals, such as diagrams. In doing so, it will become possible for future work to continue to evolve `ChangeAdvisor` adding new features, in order to better support developers, in the tedious task of gathering and understanding user change requests, by proposing entities that need change, in a fully automatic fashion. This would enable developers to leverage the wealth of information contained in reviews, and better plan their *QA* resources.

Chapter 3

Approach

In this work, we aim to build a tool that is capable of a giving a team of developers insight into user reviews and the change requests contained within by (i) automatically importing user reviews, (ii) analyze these reviews in order to (iii) extract the top topics discussed and sentiments of the users, and finally (iv) link these topics to the source code which is going to need change. This tool is called `ChangeAdvisor` and it is composed of two parallel data processing pipelines. Figure 3.1 shows the two pipelines; (i) the *Review Pipeline* and (ii) *the Code Pipeline*, and the key steps of each pipeline and how they come together to compute the *links* between code and feedback.

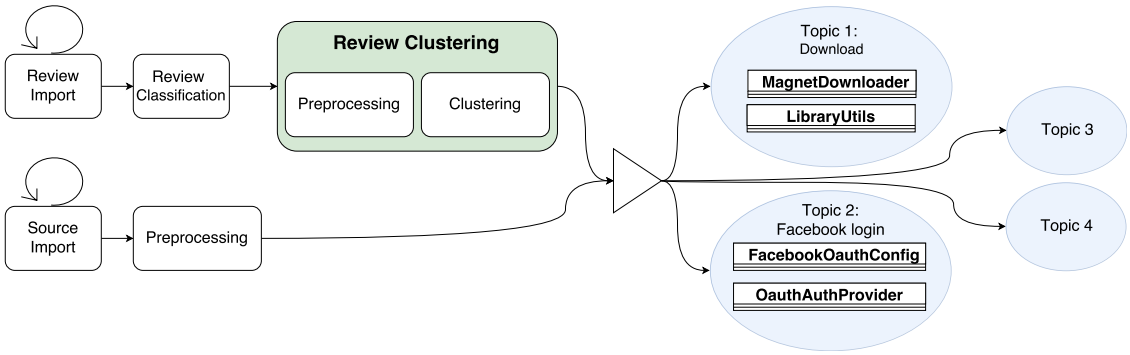

Figure 3.1: ChangeAdvisor Pipeline

This Chapter explains the overall functioning of `ChangeAdvisor` as it was implemented for this thesis, while further implementation details can be found in Chapter 4. Section 3.1 describes the *Review Pipeline*, while Section 3.2 the *Source Code Pipeline*. Section 3.3 explains how the linking algorithm determines links between source code and feedback. Finally, Section 3.4, gives a brief overview of the differences between the *Proof of Concept* and the implementation done as part of this thesis.

3.1 Review Pipeline

The ChangeAdvisor approach requires two inputs: user reviews and code. In this section, I shall describe the first pipeline, the *Review Pipeline*, that contributes to the system by feeding the linking step with reviews.

User feedback come in many shapes and forms. If we consider feedback regarding a broken feature for example, the same problem might be described very differently by different people, ranging from the very generic "ur app is terrible, i cant even login!", to the more specific "I tried going under settings and tapping the blue button and then the app crashed. Happens on a Samsung Galaxy S7" and again to "Clicking on the *log-in with Facebook* button doesn't work". All three reviews are discussing about the same problem they are having and yet are very different under many aspects. In order to be able to correctly identify similar feedback, certain steps are necessary to *normalize* user reviews.

As such a pipeline for reviews was devised that could process reviews into a usable format for the ChangeAdvisor linker. Its main steps, which can be seen in Figure 3.2, are:

- the *Review import* step imports users reviews into ChangeAdvisor. It uses the *Review Crawler Tool* [Gra] implemented in-house at UZH to mine user reviews from the *Google Play Store*.

- the *Review Categorization and Analysis* step, processes user reviews into various categories such as *FEATURE REQUEST* or *BUG REPORT*, so that in the last step we can apply topic modelling more effectively, having grouped together reviews. In order to do so, we leverage ARdoc [PDSG+16], a review classifier which uses Natural Language Processing (NLP), Sentiment Analysis and Text Analysis to classify reviews.

- the *Preprocessing* step handles the *normalization* of user feedback, in order to reduce noise in the reviews. This step is composed of classic NLP techniques such as *stop word* removal, *stemming*, and *Part-of-Speech filtering* in order to clean up user feedback.

- the *Clustering* step groups reviews discussing the same change request together. Two clustering algorithms were implemented for this work: **Hierarchical Dirichlet Process** [TJBB05] (HDP, 3.1.4) and **Term Frequency-inverse Document Frequency** [JON72] (TFIDF, 3.1.4).

3.1.1 Review Import

As mentioned, ChangeAdvisor requires reviews as an input. As such, this work includes the functionality to mine user reviews from the *Google Play Store*. This import function takes the *Google Play Store Id* of an app, and includes the possibility to import reviews either in a scheduled manner or triggered by user action. The review miner was written at the University of Zurich [Gra] and was integrated into ChangeAdvisor. This functionality represents one of the biggest changes in respect to the PoC. With the proof of concept, a developer would have had to separately export all his reviews, maybe writing them to a database or to a file. ChangeAdvisor would then run its computation over the entire data set. If he wanted to regularly use ChangeAdvisor, he would have to regularly append the most recent reviews export to his database, export into a format suitable for the tool, and then re-run the same computation even though only a small part of the ever-growing data set has changed. Thus, providing the possibility for a user of ChangeAdvisor to import reviews directly into the tool, using a predefined schedule, or even just by manually triggering the process, represents a huge step forward in terms of usability.

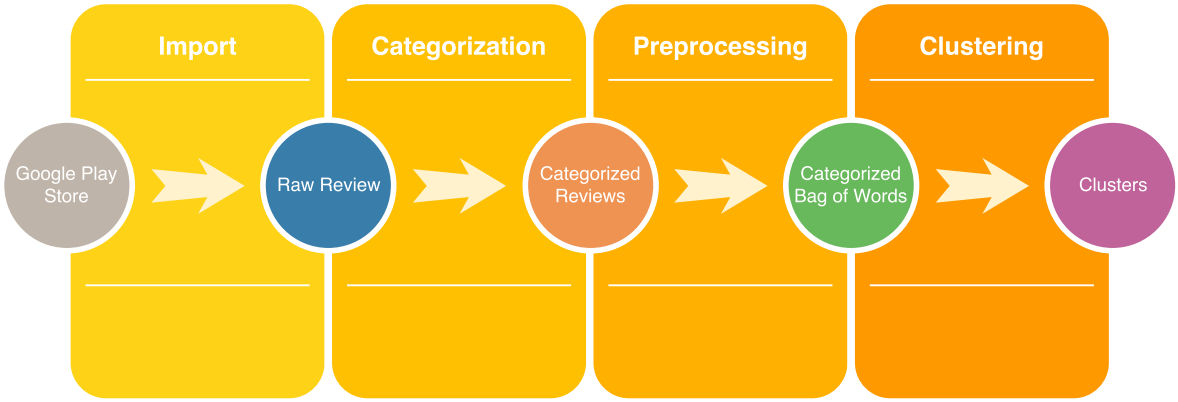

Figure 3.2: Review pipeline. The result of each step, represents the input to the following one.

3.1.2 Review Analysis

The goal of the *Review Analysis* step is to categorize reviews, as a pre-clustering step, to increase the overall accuracy of the system, as mentioned in Section 2.1.2. Reviews present in the database are analyzed using `ARdoc` [PDSG+16], in order to assign to the feedback one of the categories presented in [PDSG+15]. An overview of these categories can be seen in Table 2.2. A review might be composed of multiple sentences, while `ARdoc` analyzes single sentences in isolation. Consider a review such as "Great game very addicting would give it 5 stars. Unfortunately ever since the last update, the game crashes after the fourth turn". This review is composed of two sentences, the first part would be categorized as *Information Giving*, while the second part as *Problem Discovery*. This example shows that a single review might have different categories assigned to each sentence of the review. To keep track of context after processing, the review is split into it's composing sentences and saved separately, but each with a reference to the original sentence. Thus, the example above, would become two separate entities in the review pipeline. This is particularly useful when showing the linking results at the end, where the context of the entire sentence might be of use, instead of just part of it.

3.1.3 Review Preprocessing

Preprocessing is necessary in order to remove the noise contained in user reviews that might hinder the accuracy of the NLP techniques used and to make reviews consistent to each other. For this a series of steps were defined to transform the categorized reviews into input suitable for `ChangeAdvisor`:

Sentence Correction. The feedback's sentences are checked for spelling and grammar mistakes using the **LanguageTool** [lan] API. This tool is capable of parsing sentences and suggesting changes to correct them, but it is only able to make suggestions. Because of this, `ChangeAdvisor` applies each suggestion automatically. Since **LanguageTool** checks both spelling and grammar, it comes with a noticeable performance penalty, when processing thousands of reviews. Future work might want to look into possible alternatives for sentence correction, or limiting the correction to only spelling mistakes.

Contractions Expansion. All English (colloquial) contractions are replaced with their expanded forms (e.g. "it's" becomes "it is"). Contractions are identified by the use of regular expressions for both two parts contractions (e.g. "it's") and three parts contractions (e.g. "wouldn't've"). Once a contraction is identified, its expanded form is found from a dictionary of contractions and it is replaced.

Tokenization and Part of Speech Tagging (PoS). User feedback is split into tokens using the well known **Stanford CoreNLP** library [MSB+14]. Tokenization follows the rules of the Penn Treebank tokenization [MSM93] using Stanford's `PTBTokenizer` and works by enhancing each parsed token with its PoS tag and lemma, which will be useful in the following two steps *Nouns and Verbs filtering* and *Singularization*. Depending on the clustering algorithm used in the following steps of the pipeline, token order may or may not be maintained.

Nouns and Verbs filtering. According to Capobianco *et al.* [CLO+13], nouns and verbs carry the most meaning inside a document and can greatly help in increasing the accuracy of Information-Retrieval (IR) tasks. As such, in the *Preprocessing* step, we filter our document to keep only verbs and nouns. Nouns and verbs are identified in the previous step were each token was tagged with its PoS tag by the tokenizer.

Singularization. Tokens are normalized by means of transforming all plural forms into their singular forms. This is done by using their lemma.

Stop word removal. Stop words are all those words that carry little to no semantic meaning (e.g. "the", "a", etc...). Since these words carry no real meaning, they are discarded at this step. A stop word dictionary and a filter is included together with `ChangeAdvisor`. Future work may want to extend this dictionary or add support for multiple languages by extending the filter.

Stemming. Each token is reduced to its stem utilizing Porter's stemmer [Por80], to reduce the variation, since tokens such as "play", "playing", and "played" all convey the same semantic.

Repetition removal. As mentioned above, depending on the clustering algorithm used next, duplicates may or may not be allowed. As an example, **TF-IDF** [JON72] clustering on N-gram tokens would lose it's significance if duplicates were removed, since it would greatly change the structure of a sentence.

Short tokens removal. As with *repetition removal*, depending on the clustering used, short tokens may or may not be removed. According to Mahmoud *et al.* [MN15], tokens containing less than 3 characters are usually irrelevant for IR purposes.

Short documents removal. In this step, document shorter than three tokens are discarded, since they cannot convey a change request clearly, according to Palomba *et al.* [PSC+18].

Preprocessing Example. Below we can find an example of running our IR preprocessing on a sample review:

> *"One thing that I would really love if this app had is if it lets you create an account (or log in with your email) because whenever I get a new phone, or my phone's been reseted, I need to download the app and music all over again, which can waste a bit of time (especially since I've got lots of music on this app)."*

We now process the review, maintaining duplicates and order, but removing stop words and using a PoS-filter:

"One thing that I would really love if this app had is if it lets you create an account (or log in with your email) because whenever I get a new phone, or my phone's been reseted, I need to download the app and music all over again, which can waste a bit of time (especially since I've got lots of music on this app)."

We then stem each token:

["love", "app", "create", "account", "log", "email", "phone", "phone", "reseted", "download", "app", "music", "waste", "bit", "time", "lots", "music", "app"]

And the resulting Bag-of-Words after the *Preprocessing* step is:

["love", "app", "creat", "account", "log", "email", "phone", "phone", "reset", "download", "app", "music", "wast", "bit", "time", "lot", "music", "app"]

At the end of this step, we have a set of bag-of-words, that will be suitable to be used as input for the clustering and linking steps.

3.1.4 Clustering

Clustering is the process of grouping together a set of objects, which in some way, are more similar to one another, than to those belonging to other groups. Each group is then called a *cluster*. This process works by defining a set of *features* we use to compare one item to another and a metric, in order to quantitatively measure the similarity, in terms of distance between the features of any two items.

Consider the following example, adapted from the `ChangeAdvisor` paper [PSC+18]: *John* is the mobile app developer behind, `FrostWire`, a BitTorrent client for Android, with a presence on Google Play Store. `FrostWire` has quite a user base at the moment. In an attempt to increase his user base even further, *John* recently released an overhaul of the app, redesigning the UI and adding new features. Suddenly, the average ratings for his app are plummeting, and many users are complaining in their reviews. A lot of people seem to be complaining about having problems downloading files, while others seem to be having various usability difficulties. From the sheer amount of reviews alone, it is hard to judge, exactly, how many unique problems there are. So, *John* starts sifting through reviews, trying to isolate the various problems his users are having. Unfortunately, there are too many reviews to go through, so after a while, *John* stops and starts to plan, which issue he wants to tackle first. This is where clustering comes into play. By running a clustering algorithm, we can do this job automatically, without having *John* go through the trouble of reading all the reviews. Now, he has an overview of how many problems the current version of the app has, and can plan each fix and their priority based on gravity and number of people discussing this problem.

The goal of the *Clustering* step is, thus, to group together feedback discussing the same change requests. This step is required and is useful for two reasons: (i) as has been shown in the work of Palomba *et al.* [PSC+18], linking single reviews tends to return poor results, and (ii) clustering allows us to group together feedback referring ideally to the same, or at the very least similar, change request, thus avoiding duplicates and making the system more comprehensible and generally easier to follow for the developer.

The original `ChangeAdvisor` paper experimented with three different methods for clustering textual documents: (i) **Latent Dirichlet Allocation** (LDA) [AAT10], (ii) **genetic algorithms applied to LDA** (LDA-GA) [PDO+13], and (iii) **Hierarchical Dirichlet Process** (HDP) [TJBB05]. While experimenting, they found that **LDA-GA** was not efficient enough, and thus running it against a big data set (in the order of thousands of reviews) wasn't feasible. The original `ChangeAdvisor` PoC chose to implement **HDP**, as the authors found that it provided a good balance between speed of execution and quality of results. Thus for this work, **HDP** was im-

plemented as well (3.1.4). Additionally this work experimented with **Term Frequency-inverse Document Frequency** (TF-IDF) as well, the details of which can be found in Section 3.1.4

Hierarchical Dirichlet Process

Hierarchical Dirichlet Process (HDP) [TJBB05], is a non-parametric Bayesian model for topic extraction. It is an extension of **Latent Dirichlet Allocation** (LDA), with the advantage that it does not require the number of topics to be known a priori. Given that we have thousand of reviews and we attempt to cluster reviews by change requests, there is no way to determine the number of clusters in advance. **HDP** trains a hierarchical model to distinguish the topics, i.e. clusters, over a fixed number of iterations, based on a probability distribution, the **Dirichlet Process** [TJBB05].

Ideally at the end of this clustering step, each resulting cluster, i.e. topic, should contain a set of reviews and a Bag-of-words containing the features extracted from these reviews. Each cluster should identify a unique change requests. The Bag-of-words are the most meaningful words representing a cluster and are then used as a sort of label while presenting the results to the user.

The following shows an example of a topic and two reviews that were assigned to it:

- Topic: One of the topics (in Bag-of-Words format) that was found using **HDP** on a set of reviews for `FrostWire`:

  ["app", "love", "reinstal", "error", "look", "version", "lot", "download", "file", "fix", "phone", "try", "gui", "sai", "time", "wast", "issu", "internet", "card"]

- Two of the reviews that were assigned to this topic:

  1. *"I try to get the plus version and every time I open the file it keeps saying can't open file"*
  2. *"[...] Also when I downloaded some songs and days later I go back to play them, the app would display a message saying "file not available" when it clearly shows on my playlist.*

- And their Bag-of-Words, which are used to link a review to a topic:

  1. *["file", "sai", "try", "time", "version"]*
  2. *["song", "app", "plai", "download", "file", "playlist", "displai", "sai", "messag", "dai"]*

TF-IDF

As can be seen above, **HDP** computes topics, where each topic represents a change request. A topic is essentially composed of a list of tokens, representative of all reviews discussed inside said topic. This list of tokens is then used as a sort of label, to quickly give an idea to the developer about what a set of reviews is talking about. However, as in the example above, a label consisting of 20 or more tokens doesn't convey the information about the topic very well. Because of this, a second approach was experimented for the computation of clusters: **Term Frequency-inverse Document Frequency** (TF-IDF) [JON72]. **TF-IDF** is one of the most popular term-weighting scheme and is a technique often used in Information-Retrieval and text mining. It is commonly used on many websites, to automatically tag posts and articles, by determining the most relevant terms in a document. It works by weighting the frequency of a token inside a document against the number of documents in which the token appears over the entire corpus.

More specifically the **TF-IDF** is the product of *Term Frequency* (TF) and *Inverse Document Frequency* [JON72] (IDF). While there are multiple different formulations of **TF** and **IDF**, the following shows the way it was applied for `ChangeAdvisor`.

Term Frequency. The *Term Frequency* $tf(t, d)$ is the frequency of a token t inside a document d. Simply put, it is the number of occurrences of t in the document d, $n_{t,d}$, divided by the number of tokens in d, $|d|$ [MRS08]:

$$tf(t, d) = \frac{n_{t,d}}{|d|}$$

A high $tf(t, d)$ means that the token t appears many times in respect to the entirety of the document, which might indicate higher relevance. However, **TF** considers all tokens equally as relevant, which is not always the case. Consider for example this thesis: the words *"review"* and *"feedback"* appear many times. The resulting **TF** would probably be higher than any other word (aside from stop words). However, the above mentioned words probably convey little meaning in the context of this work. So in order to compute the actual relevancy of the token, $tf(t, d)$ is weighted by multiplying it with the *Inverse Document Frequency*.

Inverse Document Frequency. The *Inverse Document Frequency* (IDF) is used to attenuate the effect of terms, that appear too often to be relevant in the context of the entire corpus. For this we use the *inverse document frequency* $idf(t, C)$ [MRS08]:

$$idf(t, C) = \frac{N}{|d \in C : t \in d|}$$

where N is the number of documents inside the corpus C and $|d \in C : t \in d|$ is the number of document in which the token t occurs. Thus, a token that occurs in many documents scores lower than one that appears in many documents.

TF-IDF. The *Term Frequency-inverse Document Frequency* metric is simply the product of **TF** and **IDF**, or in other words, the *TF-IDF* score of a token t in a document d belonging to a corpus C is [MRS08]:

$$tf\text{-}idf(t, d, C) = tf(t, d) \times idf(t, C)$$

where a higher *tf-idf* weight for a token t implies a higher relevancy inside a document d, whereas a lower weight implies that t is not as relevant in the context of the document d.

TFIDF and ChangeAdvisor Given that *TF-IDF* computes the relevancy of a token inside a document, the choice of *what is* a document becomes particularly important, i.e. a single review should not be chosen as a document for certain reasons. The problem is that the tf term of *TF-IDF* is almost a binary value in the context of reviews and as such it does not do a good job as a discriminating feature. This is caused by the fact that reviews typically are very short and rarely contain duplicate terms (exception made for stop words). The same problem was mentioned by Naveed *et al.* [NGKA11]. The authors researched new methods to extract features from Twitter posts. Twitter posts because of their nature, are similar to user feedback in that it is kept short (< 140 characters) and informal. Additionally we do not care about the importance of a token inside a review but rather the relevance of a token inside the set of reviews labeled with a given `ARdoc` category. Because of this, the set of all reviews belonging to an `ARdoc` category was chosen as a document d while the corpus C is composed of the set of all reviews for a particular app. Additionally the **TF-IDF** component developed for `ChangeAdvisor` allows the possibility to compute the score using single **tokens** as term t, as well as **N-grams** (further details in Section 4.

The process of clustering using *TF-IDF* then works as follows and can be seen in Figure 3.3.

- For each `ARdoc` category fetch all reviews belonging to said category, this is our document d.
- For each token $t \in d$, compute the *TF-IDF* weight $tf\text{-}idf(t, d, C)$.

- For each document d, fetch the N tokens with the highest *TF-IDF* weight.
- For each of the fetched tokens t_i, fetch all reviews in the same `ARdoc` category that contain said token t_i.

Figure 3.3: TF-IDF clustering approach in `ChangeAdvisor`. 1) The initial set of reviews. 2) The reviews are grouped by `ARdoc` category. 3) We run TF-IDF for each group. 4) For each group we take the top N tokens with the highest TF-IDF score and fetch the reviews containing said token.

The following shows an example of a cluster determined by using **TF-IDF**, for the category *"Problem Discovery"*, with an N-gram size of 1, for the app `FrostWire`:

- Topic: One of the topics found using **TF-IDF**

  "Crash"

- Three reviews, verbatim, that were assigned to this topic and belong to the category *"Problem Discovery"*:

  1. *"I updated it when the description said the crashes we're fixed.
     Literally after I updated and started playing my music it crashed twice. Please fix it.."*
  2. *"It crashes and I lose all of my torrent and downloads stop. Edit: now it turns itself on. Grrrr"*
  3. *"It was okay at first but then it kept crashing and couldn't download or find hardly any rock bands I looked up. If they fixed that it would be great."*

3.2 Source Code Pipeline

The second input to the `ChangeAdvisor` process is source code. Similarly as with reviews, source code is also plagued by noise, which is why a second pipeline was devised specifically for it. This second pipeline, is much simpler than the first one. This is because source code doesn't have the same nature of reviews: (i) documents do not tend to be short, (ii) they adhere, syntactically, to the strict rules of a programming language, and (iii) even in teams of developers, guidelines are usually enforced helping making the source code more homogeneous. These reasons contribute to a lower overall noise level than online reviews. Thus, this simpler pipeline, which can be seen in Figure 3.4, is divided in two steps:

- Source Code Import: source code is imported from the file system or from version control.
- Source Code Processing: source code is processed into a suitable format for input into the `ChangeAdvisor` linker.

3.2.1 Source Code Import

The goal of this step is to import the source code into `ChangeAdvisor`. Source code that is imported, is *parsed* and divided into its components. We define a *source code element* as the basic building block of an application, i.e. in Java a *class*. It is important to note that we use Java *classes* as *elements* and not compilation units (i.e. .java files) which might be composed of multiple classes, e.g. nested classes. Furthermore, we exclude *interfaces* from being *elements* as they do not contain any implemented logic and as such would rarely be targets of change requests. Although Java 8 introduced default methods which add implementation to an interface, this is not used as often. Future work might want to look into this, especially considering the recent release of Java 9 and the increased adoption of Java 8. As such, candidates for being picked up as *source code element* are classes (normal, nested, and static nested) and enums.

As in the PoC, for each *source code element*, we parse its public API and related comments. The terms of each *element* are collected for processing in the following step.

Here, again, the biggest newest feature is the possibility to automatically import source code from *git*, which increases the overall usability of the tool. Other VCS options, can easily be added in future. Further details can be found in Chapter 4.

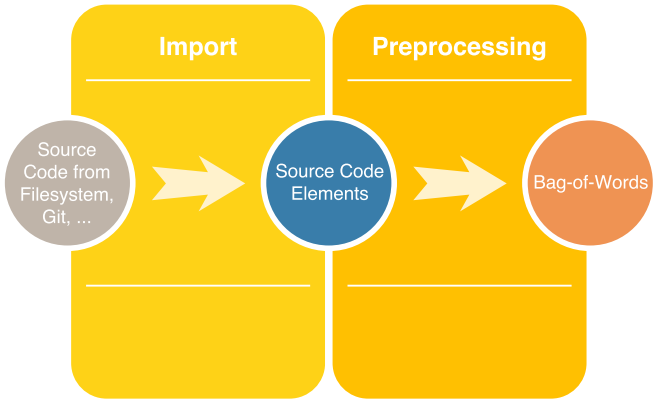

Figure 3.4: Source Code pipeline. Again, the results of a step, are passed to the following one as inputs.

3.2.2 Source Code Preprocessing

In the preprocessing step, we normalize the terms of each *source code element* to lower the noise level and to bring the *elements* into a usable format for linking. The preprocessing steps are similar to those of the *Review pipeline* seen in Section 3.1.3:

Separation of composed identifiers. We split composed identifiers, commonly found in many programming languages, such as camelCase, snake_case and digit separated text. To do this, we employ regular expressions.

Removal of special characters. Special characters such as brackets are removed from the document as they do not convey any semantics.

Removal of stop words. In addition to normal english stop words, source code also includes programming stop words, i.e. terms such as *private*, *public*, *void*, etc. These terms are also removed as they are not usable as discerning features.

Stemming. We normalize each token by replacing it with its stem, computed using Porter's stemmer [Por80].

Removal of short tokens. As a last step, we remove all tokens shorter than a threshold, for the same reason we removed them during the *Review Preprocessing* step (3.1.3).

At the end of this step, we should, ideally, have a set of Bag-of-words, each representing a *source code element*, suitable to be used for linking with reviews.

3.3 Linking

Regardless of the clustering algorithm used for reviews, we now have a set of clusters, each containing an `ARdoc` category [PDSG+16] and a set of reviews as Bag-of-Words, ideally representing

the same, or at least similar, change requests, and a set of *source code elements*, also in Bag-of-words format.

Our goal is to find links between our feedback clusters and source code, each link representing a set of classes that are probable candidates for modification, in order to fulfill the change request represented by the cluster. In order to compute links, we must first define how we measure the *vicinity* of two documents, i.e. a *metric*.

3.3.1 Metric

As mentioned, we must first define a *metric*. As it was the case in the `ChangeAdvisor` PoC, we compute the similarity using the Asymmetric Dice coefficient, the definition of which can be found below [PSC+18]:

$$\text{similarity}(cluster_i, source_j) = \frac{|W_{cluster_i} \cap W_{source_j}|}{min(|W_{cluster_i}|, |W_{source_j}|)}$$

where $W_{cluster_i}$ represents the set of words contained in cluster i. W_{source_j} represents the set of words contained in the *source code element* j. The min function normalizes the similarity score with respect to the shortest document between i and j. We normalize with respect to the shortest document, because user feedback is, in most cases, considerably shorter than our *source code elements*. Due to the normalization step, the Dice coefficient range is $[0, 1]$.

3.3.2 ChangeAdvisor Linking

Having defined the inputs, and a suitable metric for document similarity, we now define the steps and conditions needed to compute links between a *review cluster* and a *source code element*. Given a set E_{source} which represents the set of all *source code elements*, $C_{cluster}$ the set of all clusters, and T the threshold to reach in order for a cluster and element to be considered a link:

- define a bucket b_i for each cluster $c_i \in C_{clusters}$.
- for each cluster c_i compute the Dice coefficient score with each code element $e_j \in E_{source}$.
- if the Dice score reaches the threshold T, add the *element* e_j to the bucket b_i

The linking process can be seen in Figure 3.5. At the end of these simple steps, we shall have a set of buckets, where each bucket b_i contains the set of all links found for the cluster c_i, or an empty set, meaning that no links were found for a given change request.

3.4 ChangeAdvisor Proof of Concept Limitations

This chapter gave an overview of the approach implemented in `ChangeAdvisor`. This is mostly unaltered from the approach applied in the Proof of Concept [Cha17]. During the development of this thesis, it became apparent that the PoC deviates slightly from the process as explained in the work of Palomba *et al.* [PSC+18]. Thus for the implementation, we chose to mostly concentrate on the PoC interpretation of `ChangeAdvisor`.

However, the PoC is limited under many aspects. Under the usability aspect, the PoC is run as a command line program. It outputs its results in a CSV (*comma separated values*) file. It is not possible, to visually explore the results using charts and diagrams for example, which could help in understanding and interpreting the results. Additionally a user is required to dump his

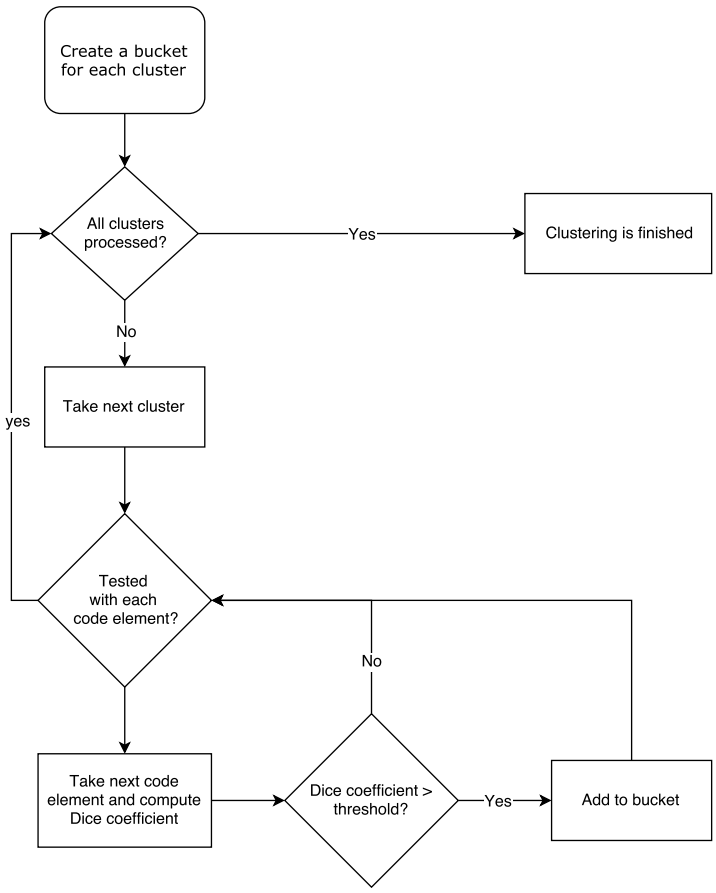

Figure 3.5: `ChangeAdvisor` linking.

reviews in a plain text file. Source code is imported by providing the path to the root folder of an app. The PoC doesn't persist any values between runs so, every time we want to re-run `ChangeAdvisor` we need to create an updated version of the review file, which in turn, greatly impacts its efficiency: the PoC will then recompute all values from scratch, which means that as the code base grows and, in particular, the number of reviews increases, `ChangeAdvisor` will become more and more slow. There is also no support for stopping and restarting a job at a later time. Once the PoC has started, it either has to go through the entire process, which might take hours for a relatively small data set, or if we want to interrupt it, we will have to restart the process from the beginning at a later time.

The PoC is also not configurable in any way. All of its steps are hard-coded, so it isn't possible to, say, swap out a clustering algorithm for another one, or modify the preprocessing steps, at run-time. This kind of configurability would require a substantial change in the tool's design and a complete rewrite of the source code.

On the maintainability side of things, it is very hard to maintain a code base, consisting of various scripts written in different languages and brought together by glue code. A maintainer of the tool might find himself tracking down a bug through different applications.

Thus, one of the main contributions of this work, is the complete rewrite of `ChangeAdvisor`

as a cohesive Java application, including a GUI in the form of a web application. This all translates to:

- Increased usability for the developer: features such as the automatic import of reviews based on a schedule, import of code through VCS, visual representation of reviews, ratings distributions in the form of diagrams and charts, all greatly help in making the best use of a developer's time when dealing with feedback. Not to mention, that the GUI greatly simplifies the use of `ChangeAdvisor` under many key aspects:
  - Configuration: e.g. creation of a project, setting a schedule, setting the remote path to a repository, etc.
  - Exploration of results: e.g. the usage of labels computed through **TFIDF** clustering, for example, can more quickly transmit to the developer, what the main topics discussed in the reviews are.
  - Starting long-running jobs: e.g. even with a schedule, it is still possible to manually start a review import.
- Increased performance: we leverage the database, in order to not have to recompute values that can be persisted. Given, that the database contains data of a previous pipeline step, a job can resume from the following step. Additionally, we can use the database for costly operations such as search and aggregation operations.
- Increased configurability for the researcher, testing possible research questions and links between reviews and code (e.g. *Release Engineering*), allowing them to run `ChangeAdvisor` with different processing steps and/or clustering algorithms.
- Increased maintainability and extensibility, since `ChangeAdvisor` itself is an application, which undergoes evolution, in much the same way, as the apps it analyzes.

The concrete details of all changes can be found in Chapter 4.

Chapter 4

ChangeAdvisor - the Tool

This section presents the architectural considerations and details, that went into the implementation of `ChangeAdvisor`. First, we describe the underlying architecture, such as how the application is divided in its components and how these interact. Afterwards, we will look at how each component was designed, its architecture, and its technical challenges. We shall follow a top-down approach in our explanation, each section expanding more in detail. Thus, this chapter is divided as follows:

- In Section 4.1, the overall architecture is described.
- In Section 4.2, the core engine of `ChangeAdvisor`, e.g. how scheduled tasks work, how are the pipelines orchestrated, etc., is explained.
- Starting from Section 4.2.3, the core business logic is reviewed. Specifically the review (4.2.3) and source code pipeline (4.2.4) implementations are explained.
- In Section 4.2.6 and Section 4.3, we shall briefly discuss persistence in `ChangeAdvisor` and the included thin client.
- Finally, in Section 4.4, the installation and usage notes can be found.

4.1 Architecture

At the beginning of this work, `ChangeAdvisor` was conceived as a plug-in for an *Integrated Development Environment* (IDE). The basic concept was that, it would be of great use to a developer, if there were an application that could show a view of change requests and the code that is a candidate for change, directly from within the IDE. This way, all information could be contained into a single application, avoiding the mental overhead of having multiple applications. During development, however, it became apparent that `ChangeAdvisor` might be better suited as a standalone application, because of performance considerations: e.g. mining user reviews requires a noticeable amount of RAM. Most IDEs are, as-is, already heavy-weight applications, requiring a considerable amount of resources. `ChangeAdvisor` might risk slowing down the entire IDE and as such be more of a nuisance, rather than being helpful. Additionally, long running tasks and scheduled jobs are better suited as a background service, decoupled from the editor they are working in: e.g. a developer might not keep his/her IDE open during the entire work day. Indeed, a considerable amount of the time of a developer is spent away from code: brainstorming, design meetings, administrative tasks, etc. In the context of a company, it could prove to be more useful to host `ChangeAdvisor` on a server in the intranet, in order to handle all of the

firm's applications. This would free up a lot of resources on each developers computer, while maintaining the same level of usability. Indeed, it would translate in increased performance, since we could dedicate a machine for the sole purpose of running `ChangeAdvisor`.

In order to keep open the possibility of developing a plug-in in future, it was decided that `ChangeAdvisor` should become a standalone application, providing an API with which all functions can be exerted and a decoupled UI. This would allow for future work to develop any sort of UI, from IDE plug-in to mobile app, to a web application. Much in the same way, as how the popular static analysis tool `SonarQube` [Son17] works.

Thus, `ChangeAdvisor` was, in the end, designed as a client-server application, with the UI being a thin client. This client can then be hosted on any static web server. While the back-end, which does all the heavy lifting, would be installed as a service, ideally, on a dedicated machine, or less optimally, on the developer's machine, providing an HTTP API to which the client can connect.

Below follows the description of the back-end service (4.2), while the client, is described in Section 4.3.

4.2 ChangeAdvisor Server

As mentioned previously, the back-end contains the core functionality of `ChangeAdvisor`. It is the engine that does all of the processes discussed in Chapter 3, and as such, it is the most complicated part of this work. The *Java application* was developed following the layered architecture pattern, and is composed of three layers:

- REST API (top-layer, 4.2.2)
- `ChangeAdvisor` business logic (mid-layer, 4.2.3, 4.2.4, 4.2.5)
- persistence (bottom-layer, 4.2.6)

The main purpose of the back-end, is to handle all long running jobs, such as the scheduled import of reviews, review processing, etc. Because of this, most of its logic, does not require user intervention, rather it is triggered by events, such as a schedule. Thus, most of the `ChangeAdvisor` pipeline is run offline, with only the last step of the pipeline, the linking, triggered by the user, due, unfortunately, to time constraints.

In order to support fully automatic, long running jobs, the `ChangeAdvisor`'s pipelines were implemented following a **batch processing** approach.

4.2.1 Batch Processing

Batch processing is the execution of a series of non-interactive job, i.e. with minimal or no user intervention. Each job is defined as a list of steps, each step taking something as input and outputting something for the following step. Each steps is triggered automatically once the previous step is done, and uses the output of the previous step as input. This approach allows to maximize the efficiency of a process, since it removes the biggest cause of low resource usage: user input. Additionally, it creates a clear separation of concerns, as each step has a simple interface and depends only on the input from the previous step and doesn't concern itself with any other details.

ItemReader, ItemProcessor, ItemWriter.

Batch processing was implemented using **Spring Batch** [Piv17a], which provides reusable building blocks to define jobs, extensive support for scheduling, retry logic, monitoring, parallelization, chunk-oriented processing and transaction management.

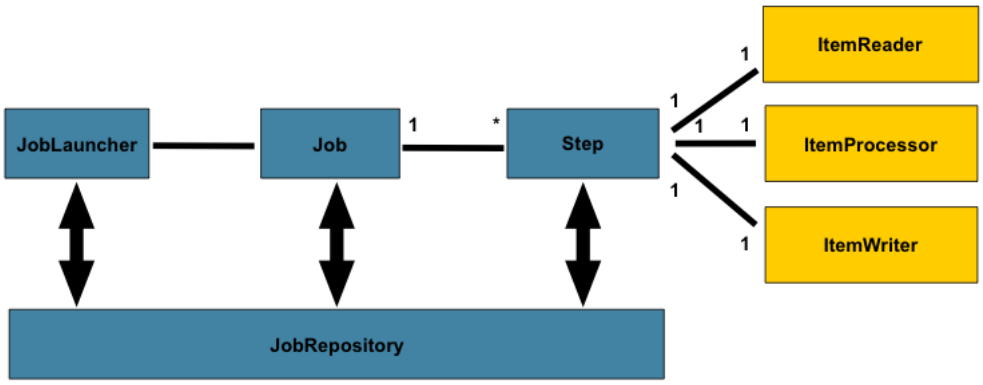

Figure 4.1: Key concepts of the Spring Batch domain language [Piv17a].

Figure 4.1 highlights the key concepts of a batch job. A job consists of steps, each step having exactly one `ItemReader`, one `ItemProcessor`, and one `ItemWriter`. A job has a `JobLauncher` which concerns itself with starting the process, and metadata about the currently running process which is stored using the `JobRepository`. Finally, an `ItemReader` concerns itself only with reading items from a source, an `ItemProcessor` runs a certain process on said items, and an `ItemWriter` writes the results of the processor to a destination. With this simple definition almost the entirety of the `ChangeAdvisor` process was implemented. Indeed, most steps in the pipelines can be seen as (i) read an item (e.g. get reviews), (ii) process an item (e.g. compute ARdoc [PDSG+16] category), and (iii) write an item (e.g. pass processed review to the next step). These interface, allow for extensive flexibility: thanks to their simplicity, it is particularly simple to swap one processor out for another one, for example, making it easier for future works to test new approaches. Listing 4.1 through Listing 4.3 shows an example of how the ARdoc processing step was implemented in the back-end: each review is read one at a time and passed on to the processor. To avoid continuously opening and closing the database connection, **chunk-oriented processing** was used. With **chunk-oriented processing** we read and process items one at a time. Before sending them to the writer, however, we wait until a certain configurable threshold of items has been processed. Only then are the processed items sent to the writer, which will then write them all in a single transaction, thus lowering the overhead of managing the database connection.

```
1  public class ReviewReader implements ItemReader<Review> {
2
3     ...
4
5     @Override
6     public Review read() throws Exception {
7        return readNext();
8     }
9
10    private Review readNext() {
11       if (isNotYetInitialized()) {
12          List<Review> reviewsSinceLastAnalyzed =
13             service.getReviewsSinceLastAnalyzed(appName);
14          contentIterator = reviewsSinceLastAnalyzed.iterator();
15       }
16
17       if (!contentIterator.hasNext()) {
18          contentIterator = null;
19          return null;
20       }
21
22       return contentIterator.next();
23    }
24 }
```

Listing 4.1: An example of an `ItemReader`: the `ReviewReader` class reads reviews one at a time, starting from the last review analyzed.

```
1  public class ReviewProcessor implements ItemProcessor<Review, ArdocResults> {
2
3     ...
4
5     @Override
6     public ArdocResults process(Review item) throws UnknownCombinationException {
7        List<Result> results = parser.extract(ARDOC_METHODS, item.getReviewText());
8        ArdocResults result = new ArdocResults(item, results);
9        trackProgress();
10       return result;
11    }
12 }
```

Listing 4.2: An example of an `ItemProcessor`: the `ReviewProcessor` class processes each review using ARdoc.

```
1  public class ArdocResultsWriter implements ItemWriter<ArdocResults> {
2
3     private MongoItemWriter<ArdocResult> writer = new MongoItemWriter<>();
4
5     ...
6
```

```
7     @Override
8     public void write(List<? extends ArdocResults> items) throws Exception {
9        for (ArdocResults results : items) {
10          writer.write(results.getResults());
11       }
12    }
13 }
```

Listing 4.3: An example of an `ItemWriter`: the `ArdocResultsWriter` class writes the results of the processing step to the database.

Batch Job Configuration

The steps of a batch job are configured through the definition of **Spring Configuration** classes. Listing 4.4 shows the definition of the *ARdoc* step with the previously mentioned classes. Each configuration class decides which implementation of `Item{Reader, Processor, Writer}` it wants to use to implement a step. Additionally, it defines the input and output types of the processor, and the size of the processing chunks.

```
1  @Component
2  public class ArdocStepConfig {
3
4     public Step ardocAnalysis(final String appName) {
5        return stepBuilderFactory.get(STEP_NAME) // "ardoc_step"
6              .<Review, ArdocResults>chunk(100)
7              .reader(reviewReader(appName))
8              .processor(reviewProcessor())
9              .writer(ardocWriter())
10             .build();
11    }
12
13    private ItemReader<Review> reviewReader(String app) {
14       return new ReviewReader(ardocService, app);
15    }
16
17    @Bean
18    public ItemProcessor<Review, ArdocResults> reviewProcessor() {
19       return new ReviewProcessor();
20    }
21
22    @Bean
23    public ItemWriter<ArdocResults>() {
24       return new ArdocResultsWriter();
25    }
26
27    ...
28 }
```

Listing 4.4: An example of the configuration of a step: the `ArdocStepConfig`. Through its fluent API, we define the behavior of a step.

Finally, a job is composed of a list of steps. Each job is configured in the same way as a step, using a configuration class. Listing 4.5 shows the configuration for the entire *Review Pipeline*.

```
1  ...
2  public Job reviewPipelineJob(String googlePlayId, Map<String, Object> params) {
3        return jobBuilderFactory.get(REVIEW_PIPELINE)
4              .incrementer(RUN_ID_INCREMENTER)
5              .flow(reviewImport(googlePlayId, params))
6              .next(ardocConfig.ardocAnalysis(googlePlayId))
7              .next(feedbackProcessingStepConfig.transformFeedback(googlePlayId))
8              .next(documentClusteringStepConfig.documentsClustering(googlePlayId))
9              .next(tfidfStepConfig.computeLabels(googlePlayId))
10             .next(linkingStepConfig.clusterLinking(googlePlayId))
11             .end()
12             .build();
13 }
```

Listing 4.5: Definition of the the *Review Pipeline.*

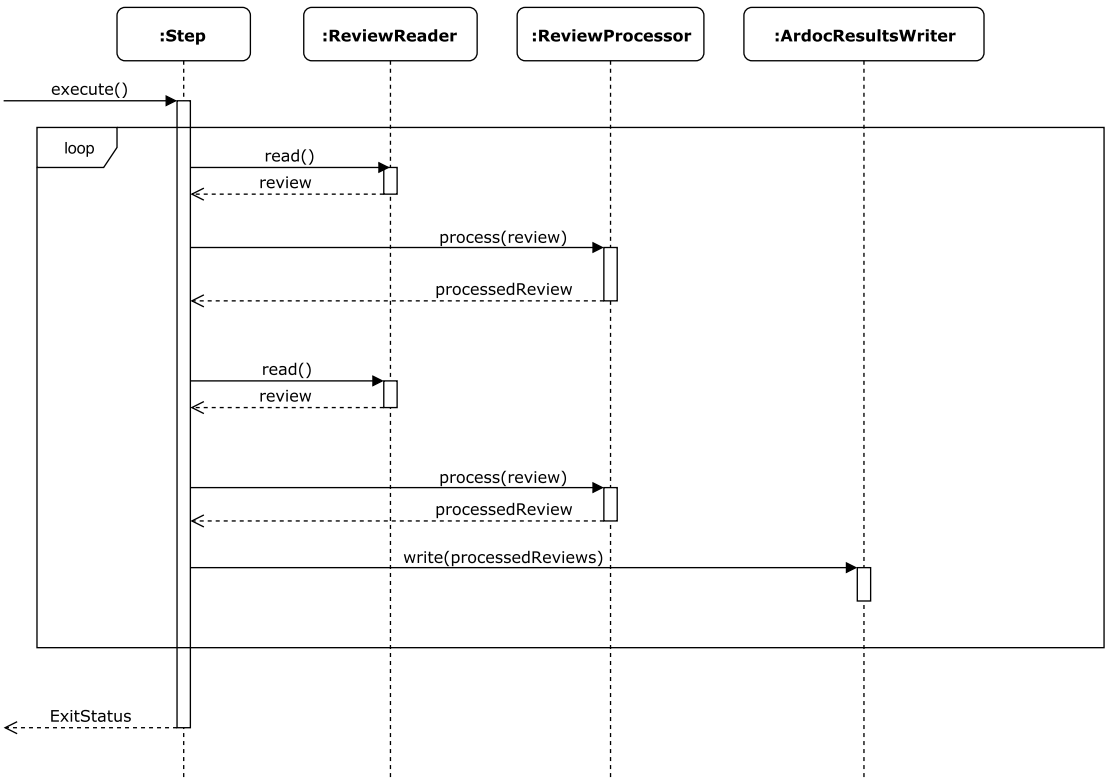

Figure 4.2: Sequence of *read(), process(), write()* calls for the *ARdoc processing* step with a chunk size of 2.

Once a job has been created and executed through the use of a `JobLauncher`, it is entirely handled by **Spring Batch**, which does the actual passing of inputs and outputs, and decides when to send a chunk to the writer. Figure 4.3 shows, as an example, the sequence of actions for the *ARdoc processing* step.

Tracking Progress

Many of the `ChangeAdvisor` jobs are long-running. For example, an initial review import with a limit of 5000 reviews can take between 7 and 8 hours, while the *ARdoc* analysis of a single review can take up to 3 seconds, which means that the first two steps of the *Review pipeline* can take around 12 hours to complete. During this time, it would be useful to have a way to monitor progress. Because of this, monitoring was added. **Spring Batch** already includes tools to check the progress of a job, through its `JobRepository` interface. However, I found that it was a bit hard to use, and the monitoring reports were hard to read and overly verbose. Additionally, the reports included information only, from the **Spring Batch** side of things. It wasn't possible to know, for example, how many reviews have been mined up to a certain point. Thus, I added a simpler way to check for status updates. A `JobService` class was added, which centralizes all access to the `JobLauncher` class, so anytime a new job is executed, it goes through the service class. There, each new job is saved after launch in a `JobHolder` class, using a map, where the *jobId* is the key and the `JobExecution` class is the value. From here, custom reporting logic was implemented, which includes custom reports with more domain-specific information. Each `JobExecution` instance includes a list of `StepExecution` instances, each with its own `StepExecutionContext`, which one can use as a map to write information. This was mostly designed for passing extra data between steps, but can also be used for execution metadata. Figure 4.3 shows the classes involved and their dependencies.

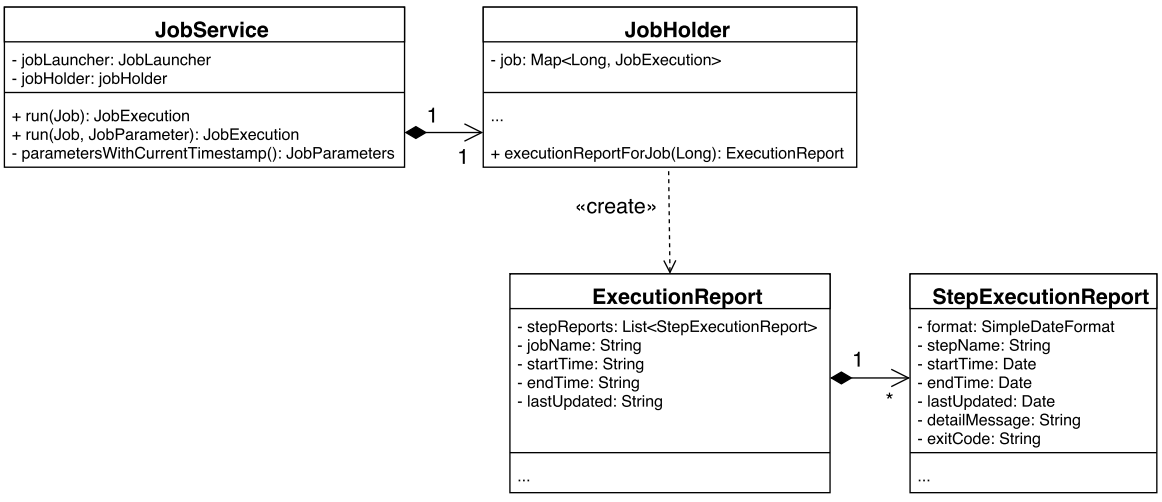

Figure 4.3: Class diagram of the job launching and monitoring part of the tool.

Scheduling Jobs

`ChangeAdvisor` includes the possibility to schedule jobs. At the moment the scheduling is restricted only to review import, while source code import must be triggered manually, through the API. This is because, it would be more appropriate to trigger source code imports by means of VCS hooks, e.g. anytime that someone pushes a set of commits to the master branch.

Through the API (or the client) a user can set a schedule using the *setSchedule()* method, seen in Listing 4.7. In order to be able to cancel scheduled jobs, for example in the case the user changes his mind, e.g. he had the recurrence set for weekly but now wants daily imports, the previous scheduled task must be canceled and replaced by a new schedule. To be able to do this, we keep track of all scheduled tasks inside the `ScheduledReviewImportConfig` using a map of *projectId* and *scheduledTask*.

Finally, a schedule is defined using cron expressions. Cron expressions were, originally, expressions used by the `Cron` job scheduler in Unix-like systems, to define recurring tasks. Nowadays, however, they are often used in many other systems, due to the fact they are easy for machines to parse and powerful enough to define any kind of periodic schedule.

```
1  # ---------------------- minute (0 - 59)
2  # | ---------------------- hour (0 - 23)
3  # | | ---------------------- day of month (1 - 31)
4  # | | | ---------------------- month (1 - 12)
5  # | | | | ---------------------- day of week (0 - 6) (Sunday to Saturday)
6  # | | | | |
7  # | | | | |
8  # | | | | |
9  # * * * * * command to execute
10
11 # Run the backup shell script daily at 12:00 (24h clock)
12 0 0 12 * * ./backup.sh
```

Listing 4.6: Anatomy of a cron expression and an example of a recurring task.

These cron expression are sent to a custom scheduler, that leverages the Spring framework scheduling abstractions, to set the date and time of the following execution. Scheduled jobs are managed by the Spring framework using triggers. A trigger is an abstraction, representing the condition with which a job must be executed, by means of setting a *nextExecutionDate* as can be seen in the return of the *trigger()* method in Listing 4.7. The basic idea is that the execution times may be based on previous runs or based on arbitrary conditions. This information is available inside the `TriggerContext`. As such, anytime the trigger is set off, it computes the following execution time and schedules itself for the next run, creating a recurring job. Additionally, each task receives its own thread from a pool of threads, thus enabling multi-threading and allowing `ChangeAdvisor` to handle multiple applications simultaneously.

```
1  public class ScheduledReviewImportConfig implements SchedulingConfigurer {
2
3     private ScheduledTaskRegistrar taskRegistrar;
4     private Map<String, ScheduledTask> scheduledTasks;
5     ...
6     public void setSchedule(final Project project) {
7        logger.info(String.format("Updating schedule for [%s]",
8              project.getAppName()));
9        final String projectId = project.getId();
10       TriggerTask task = triggerTask(project);
11       cancelScheduledTaskIfAnyExists(projectId);
12       ScheduledTask scheduledTask = taskRegistrar.scheduleTriggerTask(task);
13       scheduledTasks.put(projectId, scheduledTask);
14    }
15
16    private TriggerTask triggerTask(final Project project) {
17       Trigger nextExecutionTrigger = trigger(project.getId());
18       return new TriggerTask(
19             () -> startReviewImport(project),
20             nextExecutionTrigger
21       );
22    }
23
24    private Trigger trigger(final String projectId) {
25       return triggerContext -> {
26          Project project = projectService
27             .findById(projectId).orElseThrow(IllegalArgumentException::new);
28
29          Date next = getNextExecutionTime(project.getCronSchedule());
30          ...
31          logger.info(
32          String.format("Setting next execution time for [%s]: %s",
33             project.getGooglePlayId(),
34             next));
35          return next;
36       };
37    }
38    ...
39
40    private void cancelScheduledTaskIfAnyExists(final String projectId) {
41       if (scheduledTasks.containsKey(projectId)) {
42          ScheduledTask previouslyScheduledTask = scheduledTasks.get(projectId);
43          previouslyScheduledTask.cancel();
44       }
45    }
46 }
```

Listing 4.7: Scheduling of review import.

4.2.2 REST API

The main goal of this component, is to allow communication between the back-end and any front-end. It is a thin layer with close to no logic, its main tasks being:

- Fetching data from the database, such as linking results, or reviews.
- Configuration of long-running jobs, such as the scheduling of the review import.
- Manual triggering of said jobs, mostly to be used for development.

It was decided to develop a REST API rather than other messaging protocols, since it is the most portable, allowing the widest range of clients to be developed. The API was then implemented using **Spring MVC** [Piv17c].

To simplify the usage of the API, `Swagger` [Sof17] was added to the project. `Swagger` handles the documentation of the endpoints, while also adding tools for testing.

4.2.3 Review Pipeline

Having looked at how the *Review Pipeline* is composed and how it is orchestrated, we now look at the implementation details of the various steps.

Review Import

The review import process is the first step in the entire process and at the same time represents one of the biggest step forward in terms of usability of the system. This step, when triggered, start an automatic import of reviews from the *Google Play Store* by crawling the app store in order to mine reviews. The import works by using the *Reviews Crawling Tool* developed at UZH [Gra].

The *Reviews Crawling Tool* relies on `PhantomJS` [Hid17] and `Selenium` [Sel17]. `PhantomJS` is a headless scriptable webkit while `Selenium` is a tool for browser automation. `PhantomJS`, basically, acts as a browser. With selenium, the crawler is, then, scripted into perusing page after page of reviews from the app store and making copies of each review.

The tool works as follows:

- In case no reviews are present in the database for a given app, it starts crawling from the current date backwards until the configured review limit is reached (4.2.3)
- In case reviews are present in the database, it first fetches the last review imported during the previous run, by sort the reviews in ascending order by review date, and then picking the first review. It then starts crawling from the current date backwards until either the configured review limit or the date of the last review is reached, whichever comes first.

Configuration. First of all, the review crawler must be configured. The original library suggests using a properties file. In Listing 4.8 an example of the properties file is shown. Most of these properties do not need to be modified by a user, however, a user might be interested in modifying the following two properties:

- *"to"*: sets the end date for the review import. Reviews after this date are not imported and the process terminates. This can obviously be helpful in case we are not interested in all reviews but only those up to a certain point.
- *"limit"*: represents the maximum number of reviews to import.

```
to=31/12/2013
limit=2000
thread=2
store=google
extractor=reviews
get_reviews_for=newest
export_to=mongodb
phantomJS_path=phantomjs
```

Listing 4.8: Default Review Crawler properties.

Additionally a user might want to be able to set a custom configuration on a per app basis.

The `ReviewCrawler` uses a singleton instance of the class `ConfigurationManager` to manage all properties set in the config file. Thus, in order to allow for a user to customize the crawlers configuration at run-time, a new class was introduced: `ReviewsConfigurationManager`. This class takes the user's input and merges it together with default config values. Through composition, it keeps a reference to a `ConfigurationManager` instance, which is going to be passed on to the review crawler afterwards.

Crawlers and Extractors. The `ReviewCrawler`, defines two interfaces central to its functioning: the `Crawler` and the `Extractor` interface. The `Crawler` interface represents an instance of an app store crawler which can be spawned inside a thread. The `Extractor` is, then, used to launch a crawler. It handles the actual thread creation and launch.

As entry point to the review crawler, `ChangeAdvisor` uses the `GoogleReviewsCrawler` class, which implements `Crawler`, allowing us to, easily, start a crawling process into his own thread. Unfortunately, it is also designed to finish mining all reviews up to his configured limit or end-date (see 4.2.3), before saving the reviews in the database. Additionally, through the use of a webkit for crawling, we are severely bottle-necked by the network's transmission rate, web server response time, and simply from the way the app store is designed. Indeed, at the time this thesis was written, the web page of the *Google Play Store* only shows between 3 and 5 reviews at a time, depending on the amount of content. A few pages of reviews are present at the beginning, with additional pages loaded via *AJAX* upon clicking on the *"next"* button. Because of this, the crawler has a 250 milliseconds sleep time between each page. This all translates to a job spanning over the course of multiple hours, if not days.

The crawler writes to the standard output stream log messages indicating its process every 50 reviews. However, a user of `ChangeAdvisor` wouldn't have access to these logs. In order to allow for a user to access the progress of the crawler through the API/client a custom implementation of the `Extractor` abstract class was added. The *Reviews Crawling Tool* uses the `Extractor` interface as an adapter to simplify starting crawling threads. The custom implementation sees two methods added to its public API. In addition to the *execute()* (Listing 4.9) method, it has a *getProgress()* and *isDone()* method, which work as follows.

```
/**
 * @author giograno
 */
public abstract class Extractor {
    ...
    public abstract void extract();
    ...
}
```

Listing 4.9: Extractor abstract class. Through its simple interface, it simplifies usage of the crawlers. Taken

from the source code of [Gra].

After starting a crawler, we keep a reference to the `Future<?>` returned from the `ExecutorService`, which allows us to cancel an asynchronous operation or to know whether it is done. Thus, to know whether all started crawlers are done, we, simply, iterate over the list of `Future<?>` instances and check whether they are all done (Line 47, Listing 4.10).

As mentioned before, to be able to track the progress done, in the form of number of reviews mined, we keep a reference to the crawler. The *reviewsCount* and *reviews* list which, respectively, hold the number of reviews mined, and the actual reviews are private inside the `GoogleReviewsCrawler`. So, we use the Java **Reflection API**, through the popular **Apache Commons** [Fou17] library, to access the fields (*Line 40, Listing 4.10*).

```
1  public class MonitorableExtractor extends Extractor {
2     ...
3     private Map<String, GoogleReviewsCrawler> crawlers = new HashMap<>();
4
5     private List<Future<?>> crawlersRunning;
6
7     ....
8
9     @Override
10    public void extract() {
11       final int numberOfThreadToUse =
12          this.configurationManager.getNumberOfThreadToUse();
13       ExecutorService executor = Executors
14          .newFixedThreadPool(numberOfThreadToUse);
15
16       for (String app : this.appsToMine) {
17          GoogleReviewsCrawler crawler =
18             new GoogleReviewsCrawler(app, this.configurationManager);
19          this.crawlers.put(app, crawler);
20       }
21
22       this.crawlersRunning = crawlers.values().stream()
23          .map(executor::submit)
24          .collect(Collectors.toList());
25
26       executor.shutdown();
27    }
28
29    public Map<String, Integer> getProgress() {
30       Map<String, Integer> progress = new ConcurrentHashMap<>();
31       crawlers.forEach((key, value) ->
32          progress.put(key, getReviewsCounter(value)));
33       return progress;
34    }
35
36    private Integer getReviewsCounter(GoogleReviewsCrawler crawler) {
37       Integer reviewsCounter = 0;
38       try {
```

```
39            reviewsCounter = (Integer)
40               FieldUtils.readDeclaredField(crawler, "reviewsCounter", true);
41         } catch (IllegalAccessException e) {
42            logger.error(e);
43         }
44         return reviewsCounter;
45      }
46
47      public boolean isDone() {
48         return crawlersRunning.stream().allMatch(Future::isDone);
49      }
50   }
```

Listing 4.10: `MonitorableExtractor`: adds functionality that enables the user to track the progress of the review crawler.

Tasklet. The review import functionality, being the first step of the process, and being that the crawler actually manages all reviews until the mining process is over, is one of the few steps in `ChangeAdvisor` that does not use chunk processing and does not follow the usual *read-process-write* process. Instead, it uses what Spring calls a **Tasklet**. **Tasklets** were designed specifically for use cases that do not conform to chunk-oriented processing: e.g. calling stored procedures, cleanup processes, executing a script, etc. The **Tasklet** is an interface containing a single *execute()* method and can then be integrated as a step into a job. Indeed, the definition of a step allows more then just chunk-oriented processing, but allows also to use an existing job as a step or, as in this case, a **Tasklet**.

As can be seen in Listing 4.11, the `ReviewImportTasklet` handles starting the crawler through the `Extractor`'s interface. Since the tasklet is run on his own thread, there is no need to immediately return to the caller. Instead, after starting the job and before returning, we keep track of its progress by using the custom implementation's methods (*Line 11-13* 4.11). To avoid refreshing the context to often and thus adding a lot of overhead caused by the **Reflection API**, we only check for progress every 2 seconds. This is, at the moment, hard-coded, but can of course, easily be made configurable at a later moment.

```
1  public class ReviewImportTasklet implements Tasklet {
2     ...
3     @Override
4     public RepeatStatus execute(StepContribution contribution,
5                                 ChunkContext chunkContext) {
6
7        MonitorableExtractor extractor =
8           new MonitorableExtractor(this.apps, this.config);
9        extractor.extract();
10
11       while (!extractor.isDone()) {
12          Thread.sleep(2000); // do not refresh context too often.
13          writeIntoExecutionContext(chunkContext, extractor.getProgress());
14       }
15       return RepeatStatus.FINISHED;
16    }
17
```

```
18      private <T> void writeIntoExecutionContext(ChunkContext context, T progress) {
19          context.getStepContext().getStepExecution()
20              .getExecutionContext().put("extractor.progress", progress);
21      }
22  }
```

Listing 4.11: ReviewImportTasklet

Once the custom `Extractor` finishes importing the reviews the **Tasklet** will return the status `FINISHED`, signaling Spring Batch, that his task is over and that the process may continue to the next step. At this point the database has been populated with reviews from the *Google Play Store*, which will be used as input for the *review analysis* step.

Review Analysis

Central to the efficacy and usability of `ChangeAdvisor` is the processing step using `ARdoc` [PDSG+16]. As we've discussed in 2.1.2 running `ARdoc` over the reviews before clustering them, would increase the overall precision of `ChangeAdvisor`. Indeed, the idea behind clustering is to group together similar items. After processing them, we know to which category each review belongs, which means these reviews are already semantically closer to each other. Thus we possibly reduce the occurrence of false-positives. As an added advantage, it is informative for a developer and potential user of `ChangeAdvisor` to know how many reviews are e.g. bug reports vs feature requests, or whether there are increases in bug reports following a new software release.

Review Enhancement. We enhance the user reviews imported in the previous step by wrapping the original review with the `ARdoc` category they belong to and the timestamp at the time of the wrappers creation. This wrapper is called the `ArdocResult` class, which can be seen in Figure 4.4. Additionally, we do not wrap the original review. Rather, we make a copy of this and save the enhanced version in a new table in the database. This redundancy provides the following advantages:

- The *"vanilla"* reviews are kept separately from the processed ones. This can be very useful for experimenting:
  - in case a researcher want to try new approaches for categorization.
  - we want to integrate a new tool into `ChangeAdvisor`, which does not rely on `ARdoc`.
- We might be interested in using `ChangeAdvisor` to import the reviews and to quickly get a graphical overview of the feedback, but then want to export the reviews into another tool and/or format.

Review Classification. In order to assign a category to each review, we run the `ARdoc` parser against every review that we want to analyze. We do not want to process reviews that were already processed once, since `ARdoc` parsing is a costly operation. Thus we start the process, by pulling the last `ArdocResult` that was categorized by sorting the results in the database by their timestamp, which marks when it was processed. With the last `ARdoc` result, we then get the review it holds and its review date. With this information we can, find all reviews in the database that have a *reviewDate* greater than the last processed review. Now, we can finally pull the list of all reviews that have not been processed as of yet, as we have seen in the example listings for **batch processing**, in Section 4.2.1, and run the process as defined in the Listings 4.1, 4.2, and 4.3.

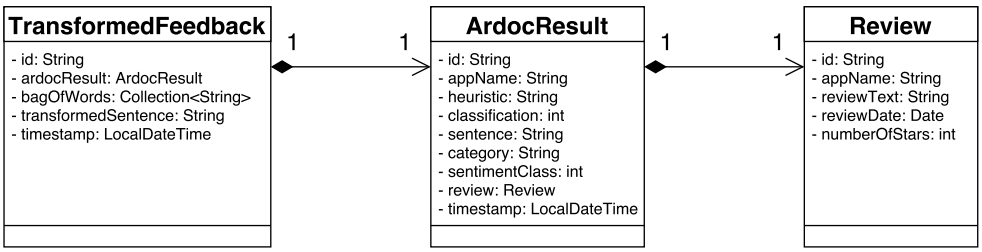

Figure 4.4: Relationship between `TransformedFeedback`, `ARdocResult` and a `Review`. The first half of the *Review Pipeline*, enhances the results of the previous step, by wrapping it and adding new information.

Flattening Results. `ARdoc` processes single reviews in isolation. This means that, for every sentence processed, it produces a separate result object. In the case, a review is composed of multiple sentences, it then returns multiple results, one for each sentence of the review. Because of this, the results must be *flattened* before writing them to the database. However, **Spring Batch** works by taking **one** item as input and returning **one** item as output. Because of this, a simple `ArdocResults` container was added that wraps all the results for a single review, which can be seen in Listing 4.12. A list of items of this class is then passed on to the writer, which flattens them, which can be seen in Listing 4.2, *line 10*.

```
1  public class ArdocResults implements Iterable<ArdocResult> {
2
3      private List<ArdocResult> results;
4
5      public ArdocResults(Review review, List<Result> results) {
6          this.results = results.stream()
7              .map(result -> new ArdocResult(review, result))
8              .collect(Collectors.toList());
9      }
10 }
```

Listing 4.12: ArdocResults. Acts as a wrapper for multiple ArdocResult, each originating from the same review.

Review and Source Code Preprocessing

This section discusses how reviews and source code are pre-processed before being sent to the linker, as we have seen in Section 3.1.3 and Section 3.2.2. Since both inputs undergo similar processing, text goes in as input, and text comes out tokenized, a single text processor, called `CorpusProcessor`, was designed for `ChangeAdvisor`, that could be used by both pipelines.

CorpusProcessor. As mentioned above, the `CorpusProcessor` handles any significant manipulation of text for `ChangeAdvisor`. It was designed to keep the public API as simple as possible, while allowing extensive configurability during creation. Because of this, as can be seen in Listing 4.13, the public API offers only two *process()* methods, with the first being syntactic sugar for the second. All functionality is, then essentially, delegated to its dependencies and is determined at the time of creation.

The processor only returns a very generic `Collection<String>` instance. This is because, the concrete implementation of the collection will depend on whether the client chose to maintain duplicates or not. The duplicate removal is implemented, simply, by using a `Set<String>`. Thus in case the client of the `CorpusProcessor` wishes to maintain duplicates, a `List<String>` is returned, so as to maintain both duplicates, as well as order. Otherwise, a `Set<String>` is used, which doesn't make any guarantees regarding the order of its items, but can very efficiently discard any duplicates.

```
1 /**
2 * @class: CorpusProcessor
3 */
4 public Collection<String> process(Collection<String> bag) {...}
5
6 public Collection<String> process(String text) {...}
```

Listing 4.13: Public API of CorpusProcessor.

CorpusProcessor Builder. `CorpusProcessor` includes an inner class that follows the **Builder** pattern [GHJV95] that handles the creation and definition of each processing step, which allows the client to freely choose how the corpus must be manipulated. Listing 4.14 shows a concrete example of the creation of such a processor, while Figure 4.5 shows the class diagram of the `CorpusProcessor` and its dependencies.

```
1  CorpusProcessor corpusProcessor = new CorpusProcessor.Builder()
2              .escapeSpecialChars()
3              .withAutoCorrect(new EnglishSpellChecker())
4              .withContractionExpander()
5              .withComposedIdentifierSplit()
6              .lowerCase()
7              .removeDuplicates(true)
8              .singularize()
9              .removeStopWords()
10             .posFilter()
11             .stem()
12             .removeTokensShorterThan(3)
13             .build();
14
15 Collection<String> processed = corpusProcessor.transform(text);
```

Listing 4.14: CorpusProcessor. Every possible processing option is shown here. Also shown here is the usage of the created processor.

To provide a concrete example of text processing, we apply processing to the following code snippet, taken from the `CorpusProcessor` public interface:

```
1 /**
2 * Will filter tokens based on their Part-Of-Speech tag.
3 *
4 * @return this builder for chaining.
5 */
6 public Builder posFilter() {}
```

Listing 4.15: Test code snippet.

Which results in:

1. After: (i) filtering special characters, (ii) splitting camel case terms, (iii) lowercasing, (iv) and tokenization.
   ["chaining", "pos", "based", "tokens", "builder", "filter", "speech", "tag"]
2. After 1) and: (i) stopword removal, (ii) stemming, (iii) and removal of short tokens.
   ["filter", "chain", "speech", "builder", "tag", "base", "token"]

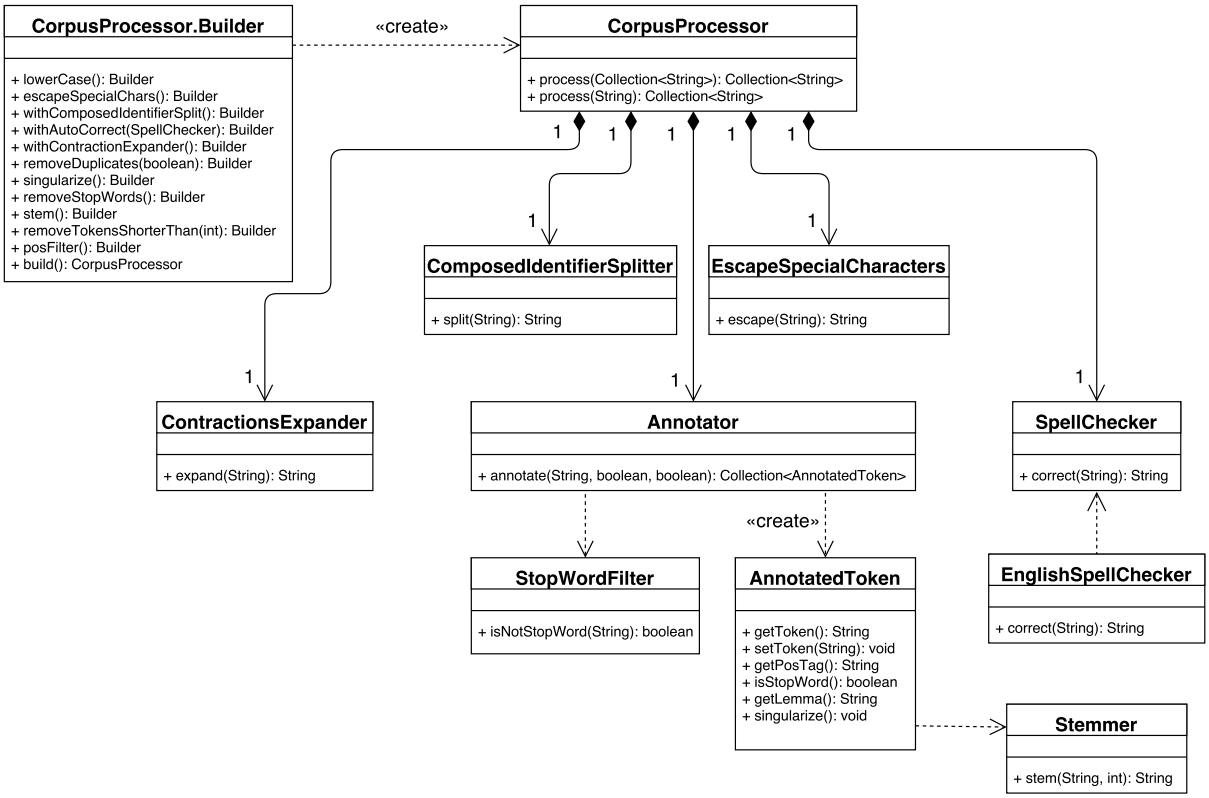

Figure 4.5: Class Diagram for the `Preprocessing` package.

Preprocessing. The actual preprocessing, then, is simply iterating through the `ArdocResult` or the `ClassBean` (see Section 4.2.4) instances computed during the previous step and running a previously configured `CorpusProcessor` on each item. The *Review* and *Source Code* pipelines, each, have a different version of the processor, depending on the preprocessing steps defined in Section 3.1.3 and Section 3.2.2. After processing an item, if the resulting bag contains less terms than a pre-determined threshold, it is discarded, as can be seen in Listing 4.16, *line 6-7*, and the process continues with next item. At the end of this step, the database will have been populated with the processed reviews (`TransformedFeedback`) or source code elements (`CodeElement`).

```
1  public class FeedbackProcessor implements ItemProcessor<ArdocResult, TransformedFeedback> {
2     ...
3     @Override
4     public TransformedFeedback process(ArdocResult item) throws Exception {
5        Collection<String> bag = this.corpusProcessor.process(item.getSentence());
6        if (bag.size() < this.threshold) {
7           return null;
8        }
9        return new TransformedFeedback(item, bag);
10    }
11 }
```

Listing 4.16: `FeedbackProcessor`. Uses the `CorpusProcessor` to process a review. Notice how, in case the processed feedback contains less terms than a pre-determined threshold, the `ItemProcessor` will returns `null`, signaling to **Spring Batch** that we want to discard this item.

Review Clustering

As mentioned in Section 3.1.4, `ChangeAdvisor` implements, two different clustering techniques, **TFIDF** [JON72] and **HDP** [TJBB05]. These implementations are not complementary but are rather offered as alternatives. Regardless of the clustering algorithm chosen, however, both take as input, instances of `TransformedFeedback`, which were computed in the previous preprocessing step. We shall first discuss about **TFIDF**, and afterwards about **HDP**.

HDP. From the operations point of view, the **HDP** clustering approach [TJBB05] works in much the same way as the other steps. Clustering is inherently dependent on the entire data set, as such we, first of all, pull all `TransformedFeedback` items, representing the processed reviews, from the database.

The processing step, then, transforms the set of all reviews into a suitable input for the clusterer, the `Corpus` class. `Corpus` is a data structure, that simplifies random access to any sentence or Bag-of-Words of the corpus, but does no actual manipulation of the data, it is simply a convenience class. Afterwards, the processor calls the *fit()* method on an instance of `DocumentClusterer`, an interface representing some abstract clustering technique, which can be seen in Listing 4.17.

The actual clustering is delegated to the `HierachicalDirichletProcess` class, which can be seen in the UML diagram in Figure 4.6. To simplify the initialization parameters to HDP and to maximize flexibilty, in case we want to test other clustering algorithms in future, and because it was designed at a later time, the `HierachicalDirichletProcess`, does not directly implement the `DocumentClusterer` interface, rather it follows the **adapter pattern** [GHJV95]. The `HierarchicalDirichletProcess` is a 1:1 port of the python code that was used in the `ChangeAdvisor` PoC, that was completely rewritten in Java for this work.

```
1  public interface DocumentClusterer {
2
3     void fit(Corpus corpus, int maxIterations);
4     List<TopicAssignment> assignments();
5     List<Topic> topics();
6  }
7
8  public class TopicClustering
9     implements ItemProcessor<List<TransformedFeedback>, TopicClusteringResult> {
```

```
10
11      private DocumentClusterer documentClusterer;
12      ...
13
14      @Override
15      public TopicClusteringResult process(List<TransformedFeedback> items) {
16          Corpus corpus = Corpus.of(items);
17
18          documentClusterer.fit(corpus, this.maxIterations);
19
20          List<TopicAssignment> assignments = documentClusterer.assignments();
21          List<Topic> topics = documentClusterer.topics();
22
23          return new TopicClusteringResult(topics, assignments);
24      }
25  }
```

Listing 4.17: `DocumentClusterer` interface and `TopicClustering`.

Once the `DocumentClusterer` implementation returns, it will hold the results of the topic modelling. We define, now, a `Topic` as a cluster, containing a Bag-of-Words to represent said cluster, and `TopicAssignment` which represents a review that is assigned to a given cluster. Only the `TopicAssignment` is going to be relevant after this step. Indeed, it is going to be one of the inputs for the `ChangeAdvisorLinker` we shall see in Section 4.2.5.

At the end of this method, both `TopicAssignment` list and `Topic` list, will be wrapped in an instance of `TopicClusteringResult` which will be sent to the `ItemWriter` for this step of the pipeline, which will simply persist both lists to the database, ready to be used, without the need for further processing, for the final step of `ChangeAdvisor`.

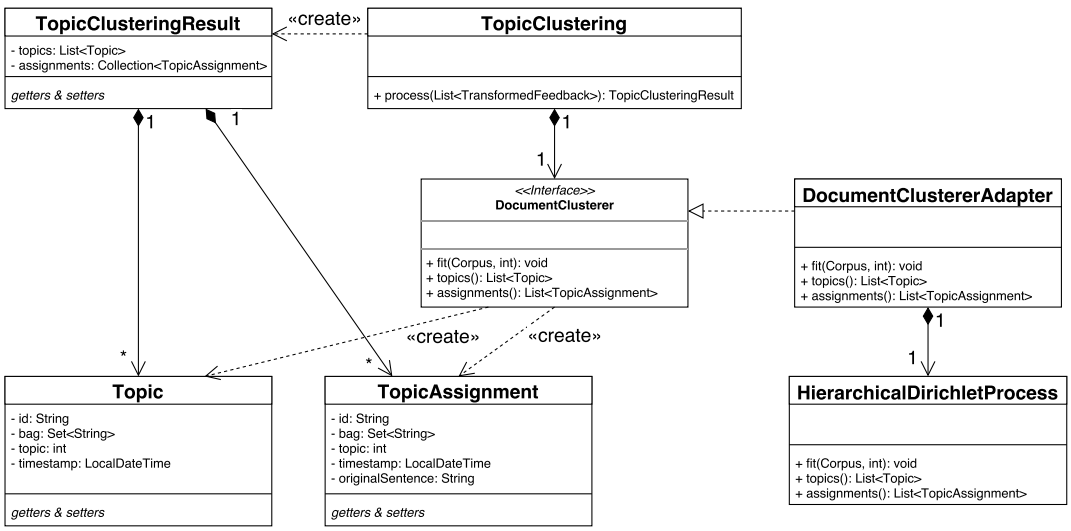

Figure 4.6: Class Diagram for the HDP clustering part of the tool.

TFIDF. As we have seen in Section 3.1.4, `ChangeAdvisor` also implements **TFIDF** clustering [JON72]. The goal is to determine the relevancy of a *term*, defined as an N-gram, inside a *document*, defined as the collection of all reviews with the same *ARdoc category*, in the context of a *corpus*, which is the set of all reviews for a given app.

With this definition set in stone, we use the following approach, which can be seen in Listing 4.18, to compute the **TFIDF** score for all terms of our data set.

This step, is the only other step, aside from the *Review Import* and *Source Code Import*, which was implemented as a **Tasklet**. This is, again, mostly due to time constraints, since this implementation is a bit more complex and less straightforward than the other steps, and it was added to the requirements at a much later date, which left us with little time to better integrate it with the `Item{Reader, Processor, Writer}` approach. Below we describe the steps which are executed in order to compute labels based on the **TFIDF** score. This process can be seen in Figure 4.7.

First of all, we delete all previous clustering results, since the review set might have changed since the last run, we need to compute fresh results. Then, we iterate over each *ARdoc* category in the data set. For each category, we compute the **TFIDF** score for multiple N-gram sizes, ranging from N-grams of size 1, *Unigrams*, to N-grams of size 3. We run this computation for multiple N-gram sizes, to allow for increased exploration possibilities of the results. Additionally, we compute the score for all terms in the document, but only persist the top N N-grams by score, because, a user of the system will hardly be interested in any label that scored too low, as scoring low on the **TFIDF** metric, means that a term has low relevancy. N is defined by the constant `MAX_LABELS_TO_COMPUTE`, which is hard-coded at the moment to 100, but can easily be made configurable at a later time.

Once we have computed the top N terms for a given category, we save the results to the database, and continue with either the next N-gram size, or the next category in case we have already computed all other N-gram size for this category.

```
1  /**
2  * @class: TopLabelTasklet.
3  */
4  public RepeatStatus execute(StepContribution contribution,
5                              ChunkContext chunkContext) {
6        this.repository.deleteAllByAppName(appName);
7        ...
8        for (ReviewCategoryReport category : reviewCategories) {
9           for (int i = 1; i < MAX_NGRAM_SIZE; i++) {
10             ReviewsByTopLabelsDto dto =
11                   new ReviewsByTopLabelsDto(
12                         appName,
13                         category.getCategory(),
14                         MAX_LABELS_TO_COMPUTE, i);
15
16             List<Label> labels = this.service.topNLabels(dto);
17             this.repository.saveAll(labels);
18          }
19       }
20
21       return RepeatStatus.FINISHED;
22    }
```

Listing 4.18: TopLabelTasklet. Manages the computation of the top N labels by TFIDF score for

various N-gram sizes and categories. The actual computation of the labels, is delegated to the `ReviewAggregationService` class.

The `ReviewAggregationService` class then, fetches the reviews, transforms them in a more suitable format for **TFIDF** computation, the `Corpus` and `Document` classes. This transformation step can be seen in the method *getNgramTokensWithScore()* at *line 28*. Both classes are, again, just convenience classes, simplifying the computation of the **TFIDF** score, by containing methods such as `Document#frequency()`, which computes the frequency of a token inside a document, or `Corpus#documentFrequency()` which iterates over the list of documents contained in a corpus, and counts how many document contains a term, through the `Document#contains()` method.

```
1  /**
2  * @class: ReviewAggregationService.
3  /**
4
5  * Retrieves the top N labels for a set of reviews.
6  * A label is an Ngram of tokens that are representative for a group of reviews.
7  *
8  * @param dto object representing the parameters we use to
9  * compute the top N labels (e.g. how many labels and for which app)
10 * @return list of labels with their tfidf score.
11 */
12 public List<Label> topNLabels(ReviewsByTopLabelsDto dto) {
13    ReviewDistributionReport reviewsByCategory = groupByCategories(dto.getApp());
14    final String category = dto.getCategory();
15
16    List<Label> tokensWithScore = getNgramTokensWithScore(reviewsByCategory, dto);
17
18    Collections.sort(tokensWithScore, Collections.reverseOrder());
19
20    final int limit = dto.getLimit();
21    if (limit >= tokensWithScore.size()) {
22       return tokensWithScore;
23    }
24    return tokensWithScore.subList(0, limit);
25 }
26
27 private List<Label> getNgramTokensWithScore(
28          ReviewDistributionReport reviewsByCategory,
29          ReviewsByTopLabelsDto dto) {
30    Map<String, Document> categoryDocumentMap =
31       mapReviewsToDocuments(reviewsByCategory, dto.getNgrams());
32
33    Corpus corpus = new Corpus(categoryDocumentMap.values());
34    Document document = categoryDocumentMap.get(dto.getCategory());
35    List<AbstractNGram> uniqueTokens = document.uniqueTokens();
36
37    return tfidfService
38       .computeTfidfScoreForTokens(dto.getApp(),
```

```
39                                      dto.getCategory(),
40                                      uniqueTokens,
41                                      document,
42                                      corpus);
43  }
```

Listing 4.19: Computation of the top N labels via the `ReviewAggregationService` class.

Now that we have the reviews separated in documents and have constructed the corpus, we send them to the `TfidfService` class, which iterates over each unique token in the document, and computes its score using the `TFiDF` class.

```
1  public class TfidfService {
2
3     private TFiDF tFiDF = new TFiDF();
4
5     public List<Label> computeTfidfScoreForTokens(String appName,
6                                                    String category,
7                                                    List<AbstractNGram> tokens,
8                                                    Document document,
9                                                    Corpus corpus) {
10       return tokens
11             .stream()
12             .map(token -> {
13                double score = tFiDF.compute(token, document, corpus);
14                return new Label(appName, category, token, score);
15             })
16             .collect(Collectors.toList());
17    }
18 }
19
20 public class TFiDF {
21
22    public double compute(AbstractNGram token, Document document, Corpus documents) {
23       return tf(token, document) * idf(token, documents);
24    }
25
26    double tf(AbstractNGram token, Document document) {
27       return document.frequency(token);
28    }
29
30    double idf(AbstractNGram token, Corpus documents) {
31       int documentFrequency = documents.documentFrequency(token);
32       if (documentFrequency == 0) {
33          return 0.0;
34       }
35       return Math.log10(documents.size() / (double) documentFrequency);
36    }
37 }
```

Listing 4.20: Actual computation of the tfidf score for a term.

Each score is then saved using the `Label` class, which contains, the token, the size of the N-gram, its score, and the category and app it belongs to.

The `Label` class is the **TFIDF** equivalent of the **Topic** class for **HDP**, it represents a set of terms representative of a discussed topic in a group of reviews and is used to group different reviews together. The difference here is that with **HDP** the topic and assignments are computed at the same time over the course of many iterations, while with TFIDF, we first compute the labels, and at a later time we simply fetch the related reviews from the database, based on whether the review contains said labels or not. Thus, the fetching of reviews is not done directly after computing the labels offline, but simply when they are needed, since now it only involves a simple database query.

The `Label` class can be seen in Figure 4.8, while the entire process can be seen in the sequence diagram in Figure 4.7. This process might seem complicated, since we keep delegating operations further down the call stack. However, it creates a clear separation of concerns, since every layer of the operation actually handles single aspects of the process: the **tasklet** manages the computations for the various categories and N-gram sizes, the `ReviewAggregationService` fetches and transforms the reviews for the chosen category and N-gram size into `Document` and `Corpus` instances, the `TfidfService` iterates over the tokens of a document, and finally, `TFiDF` computes the actual **TFIDF** score for a given token in a document.

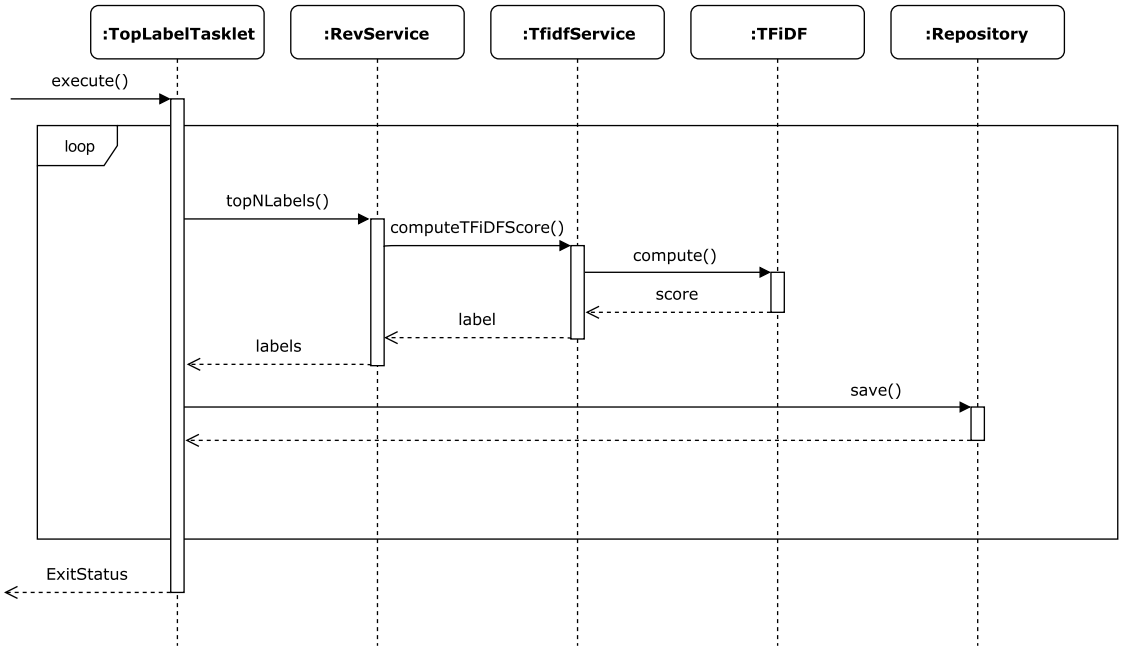

Figure 4.7: Sequence diagram representing the computation of labels.

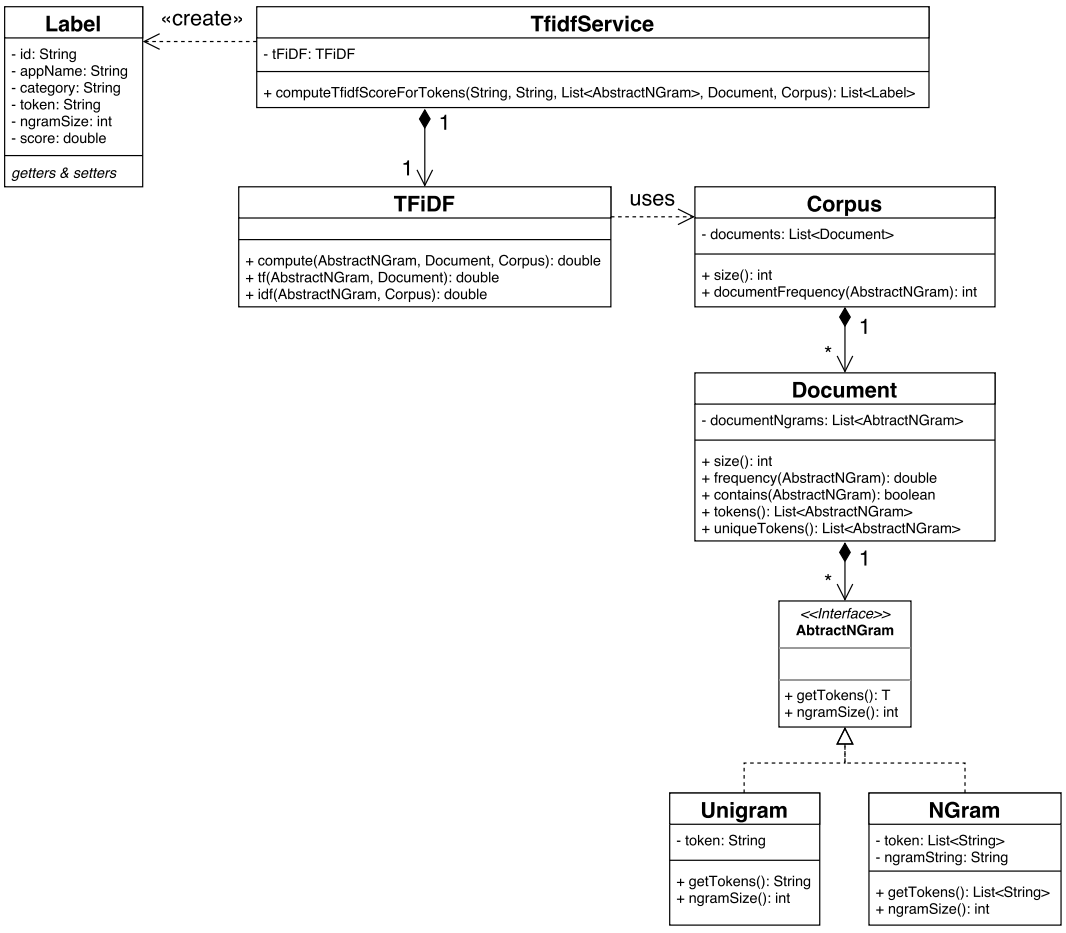

Figure 4.8: Class Diagram for the TFIDF clustering part of the tool. Only includes the part that actually does the computation.

Clustering Step Result. The end result of the *Review Clustering* step depends on the clustering algorithm used. In the case of **HDP**, the end result is a set of `Topic` and a set of `TopicAssignment` instances which represents, respectively, a set of terms representative of a group of reviews, and its associated reviews, which can then be simply fetched from the database for the `ChangeAdvisorLinker`. In the case of **TFIDF**, the end result of this step is a set of `Label` instances, where analogously to **HDP**, each instance contains a term, defined as an N-gram, that is representative for a set of reviews, persisted in the database. When the time comes to start the `ChangeAdvisorLinker`, we shall retrieve the `TransformedFeedback` instances that contain said term and that belong to the same *ARdoc category*, through a simple database query.

4.2.4 Source Code Pipeline

Having seen how the *Review Pipeline* works, we, finally, look at the *Source Code Pipeline*. This pipeline is much shorter and simpler than the *Review Pipeline*, since it only consists of two steps, (i) *source code import* and (ii) *source code preprocessing*. *Source code preprocessing*, however, is nearly identical to *review processing* and as such was already discussed in the previous section (4.2.3). Thus, in this section, we only discuss the *source code import* step.

Source Code Import

One of the steps forward, in term of usability, is the possibility to import source code directly from within `ChangeAdvisor`. Indeed, a user can, now, choose to import source code from either the file system, as it was previously in the PoC, in case the code is already present on the local system, or using the url of the git repository, in which case the code will be cloned into the local file system. Using git, or any type of version control systems (VCS), opens up new possibilities in future. Future works, might want to look into the option of correlating review ratings, source code, and commit messages, even more precisely, pinpoint the exact moment in time when a bug was introduced. Additionally, when using VCS, we can use hooks to automatically trigger code imports when new code is commited to the master branch, for example.

As mentioned previously, at the moment, code import must be triggered by the user. So the process only starts, when the users enters a path to the source code and manually triggers it through the API or client.

Depending on the path that was given, the `SourceCodeImporterFactory` (Figure 4.9) will generate an instance of an implementation of `SourceCodeImporter`, which is an interface containing a single *importSource()* method that returns a `SourceCodeDirectory` object, that contains the name of the app, the path in the file system where the code was saved, and a link to the remote url in case the project was cloned from a VCS.

There exist, at the moment, only two implementations of the `SourceCodeImporter` interface: (i) `FSSourceImporter`, which uses the file system, and (ii) `GitSourceCodeImporter` which clones the code to the file system from git. In future, new ways of importing code can be added, by implementing the interface and adding them to the factory. All parameters necessary for the import of code, are provided to the implementations through the factory: we pass the arguments to the factory, which uses them to decide the concrete implementation to use, and then passes them on to the constructor of each implementation.

The `FSSourceImporter` isn't very interesting, simply checking whether the path that was given from the user actually exists in the local file system, and, in case it found the root directory of the project, it creates the `SourceCodeDirectory` bean. Thus, we shall concentrate on the `GitSourceCodeImporter` implementation.

Git importer. In order to clone code from a remote git repository, we utilize `JGit` [Ecl17], which is a Java library from the Eclipse Foundation, that implements the git VCS, allowing us to clone the repository to the local file system, in particular, it clones it as a sub-directory inside a directory relative to the project called simply, *imported_code*, with the name of the app as the sub-directory's name. In case such a sub-directory already exists, i.e. the project has been imported at least once before, the existing directory is deleted and a new one is created.

Concretely, Listing 4.21 shows how this works, the *cloneRepo()* method, does the actualy cloning of the repository. In case the repository is private, it is possible to provide credentials to the importer, which are going to be saved in the *credentialsProvider* instance variable.

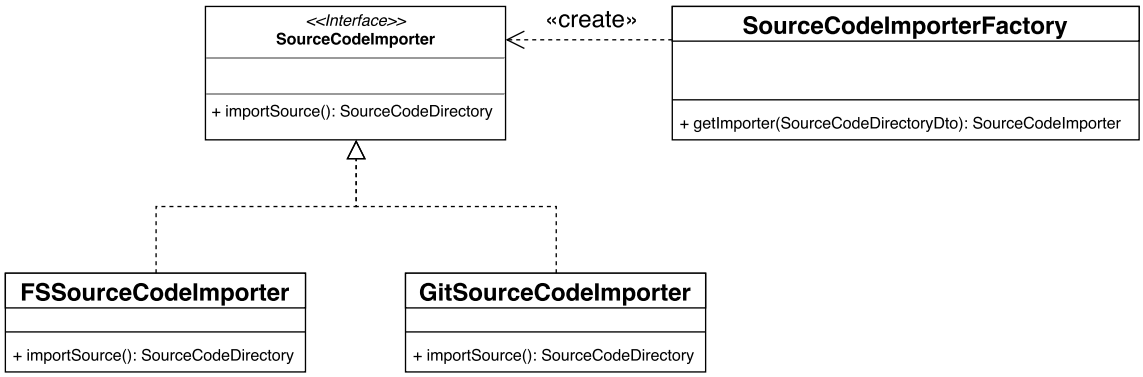

Figure 4.9: Class diagram of `SourceCodeImporterFactory`. This step only includes the download of the data into the local file system.

```
1  public class GitSourceCodeImporter implements SourceCodeImporter {
2     @Override
3     public SourceCodeDirectory importSource() {
4        final String REMOTE_URL = getURLFromPath();
5        final String projectName = StringUtils.isEmpty(this.projectName) ?
6           getProjectNameFromPath() : this.projectName;
7        final File projectPath = new File(this.projectDirectoryPath);
8
9        if (projectPath.exists()) {
10          clearDirectory(projectPath);
11       }
12
13       try (Git result = cloneRepo(REMOTE_URL, projectPath)) {
14          return new SourceCodeDirectory(projectName,
15             projectPath.getAbsolutePath(),
16             REMOTE_URL);
17       } catch (TransportException e) {
18          throw new GitCloneException(String.format(NO_CREDENTIALS_OR_NOT_FOUND,
19                                                    REMOTE_URL), e);
20       } catch (GitAPIException e) {
21          throw new GitCloneException(String.format(FAILED_TO_CLONE, REMOTE_URL), e);
22       }
23    }
24
25    private Git cloneRepo(final String REMOTE_URL, File projectPath) throws GitAPIException {
26       return Git.cloneRepository()
27             .setURI(REMOTE_URL)
28             .setDirectory(projectPath)
29             .setCredentialsProvider(this.credentialsProvider)
30             .call();
```

```
31        }
32        ...
33    }
```

Listing 4.21: `GitSourceCodeImporter`.

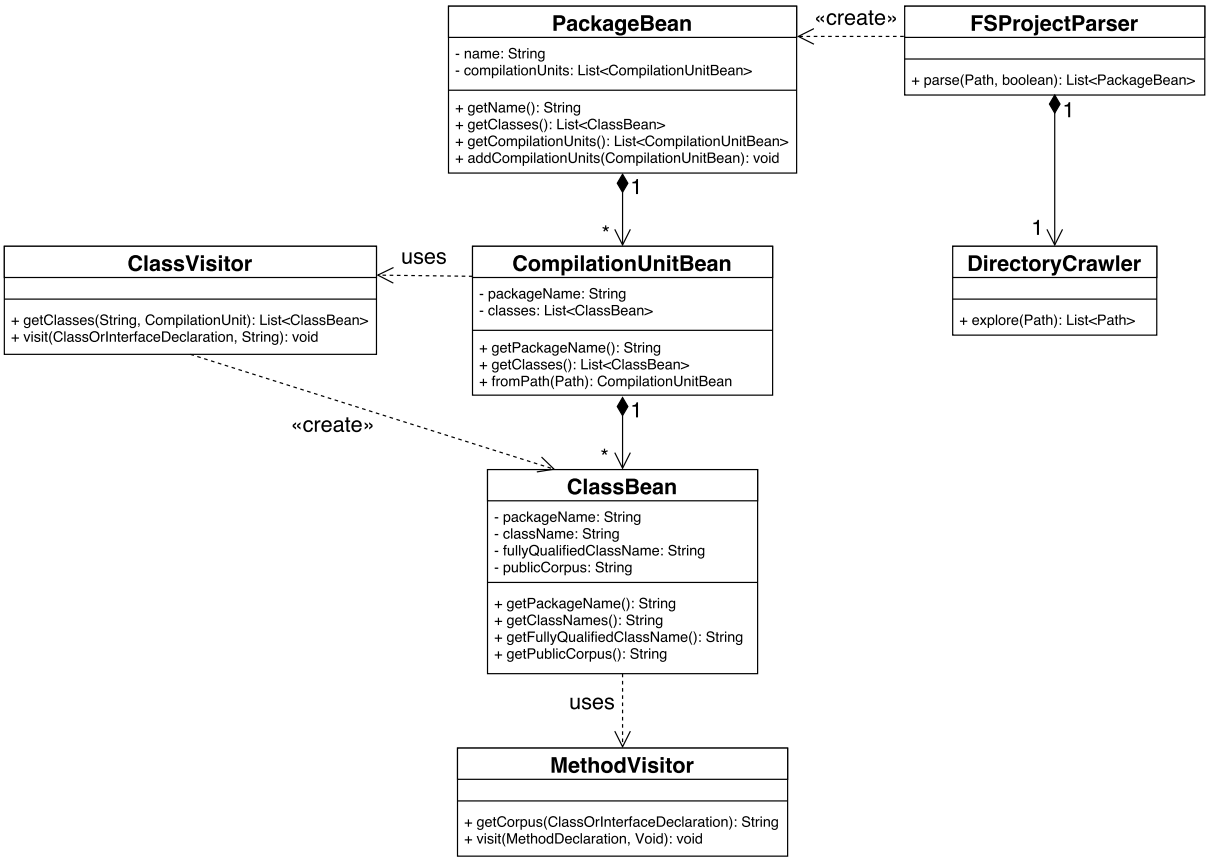

Figure 4.10: Class diagram for the source code parsing part of the tool. This step parses the public API of the source code imported in the previous step, which will be later persisted in the database.

Result of Source Code Import. Regardless, of how the code was imported, the end-goal of this step is the `SourceCodeDirectory` bean, which contains the path to the source code in the local file system. This will be used as an input for the following *code parsing* step.

Source Code Parsing. After the previous step, we have downloaded the code to the local file system, but it is not yet ready for processing. As mentioned in Section 3.2.1, we parse the source code for its public corpus. This will import the source code into a suitable form, to be then pre-processed in the following step, and at the end, linked by the `ChangeAdvisorLinker`. The class diagram for this step can be seen in Figure 4.10.

The starting point is the *parse()* method of the `FSProjectParser` class. This method uses the `DirectoryCrawler#explore()` method to get the path to each Java source code file in the file system, starting from the root folder, found in the previous step. It does so recursively: each time it finds a file, it adds it to the results list, if it finds a directory, it calls the *explore()* method to visit the directory. Once the crawler has crawled its way to all leaf directories, it returns the list of paths to all Java source code files.

With this list of paths, we can parse each file to recreate an in-memory model of the entire project. The classes involved in this are the `PackageBean` class, which represents a package of the project, the `CompilationUnitBean` class, which represents a Java file inside a package, and finally the `ClassBean` class, which represents a single class (normal, nested, static nested, or enum). The `ClassBean` class is the only one that actually contains any data from the files, all other classes, being there just to recreate the relationships in our model.

The actual parsing of files, is done using the `JavaParser` library [Jav17]. The library builds an Abstract Syntax Tree (AST), allowing us to easily parse the methods contained inside a class. To do this, it provides an API based on the visitor pattern [GHJV95]. The user of the class only needs to implement the logic inside the visitors, i.e. what the visitor should do when visiting a node, while the library handles most of the complexity of iterating through the AST, making parsing Java code an almost trivial task. Listing 4.22 shows the implementation of the `MethodVisitor` class, which given the `JavaParser` equivalent of our `ClassBean`, parses each public method, including documentation. The entry point is the static *getCorpus()* method, which handles the creation of the visitor and the execution of the visitor pattern. While visiting the nodes, i.e. methods, we accumulate the text found inside a `StringBuilder`. When the visitor finally returns into the *getCorpus()* method, it means, it is done visiting all nodes, and we can fetch the parsed text from the `StringBuilder`. The `ClassVisitor` class works analogously but builds a list of `ClassBean` instead.

```
1  public class MethodVisitor extends VoidVisitorAdapter<Void> {
2
3      private StringBuilder sb = new StringBuilder();
4
5      public static String getCorpus(ClassOrInterfaceDeclaration node) {
6          MethodVisitor visitor = new MethodVisitor();
7          visitor.visit(node, null);
8          return visitor.getPublicCorpus();
9      }
10
11     @Override
12     public void visit(MethodDeclaration n, Void arg) {
13         if (n.isPublic()) {
14             String methodText = n.toString();
15             sb.append(String.format("%s%n------%n", methodText));
16         }
17         super.visit(n, arg);
18     }
19
20     private String getPublicCorpus() {
21         return sb.toString();
22     }
23 }
```

Listing 4.22: Implementation of the visitor pattern, allowing us to parse the public interface of a class.

Result of Source Code Parsing. The end result of this parsing is, then, a list of `ClassBean` instances, each representing a class of the project, containing, among other things, the corpus representing the public interface, including comments, of the class. These instances are then persisted in the database and will be processed in the *Source Code Preprocessing* step of the pipeline (3.2.2, 4.2.3).

4.2.5 ChangeAdvisor Linking

Finally, our long journey through the pipelines has brought us to the linking. This step is where the two pipelines join and `ChangeAdvisor` can, finally, compute the links between reviews and source code. The `ChangeAdvisorLinker` requires two inputs, in order to compute links: (i) reviews and (ii) source code. For the source code, we fetch the code processed in the previous step. Regarding reviews, both `TransformedFeedback`, in case **TFIDF** was used, or `TopicAssignment`, in case **HDP** was used, can be utilized here. This is by design, because, the linker takes as input a list of items extending `LinkableReview`, an interface, presented in Listing 4.23, that both `TransformedFeedback` and `TopicAssignment` implement. Indeed, the `ChangeAdvisorLinker` itself, implements the `Linker` interface, seen in Listing 4.24.

```
1 public interface LinkableReview {
2
3     Set<String> getBag();
4
5     String getOriginalSentence();
6 }
```

Listing 4.23: `LinkableReview`. The return types of both clustering algorithms implement this interface, allowing the linker to use any of the two.

```
1 public interface Linker {
2     ...
3     List<LinkingResult> link(String topicId,
4                              Collection<? extends LinkableReview> reviews,
5                              Collection<CodeElement> codeElements);
6 }
```

Listing 4.24: `Linker`. Interface for a linking algorithm.

Thus, the process looks as follows: the processor utilizes an implementation of the `Linker` interface, calling the *link()* method, which implements the steps described in Section 3.3.2. The `ChangeAdvisorLinker`, which is the only concrete implementation of the above-mentioned interface, is a refactored port of the original linking algorithm of the Proof-of-Concept. This port is, functionally, unaltered from the original and can be seen in Listing 4.25. The biggest difference being, that the linker doesn't operate on all clusters inside the same method call. The iteration of clusters is handled by the reader for this step. This way, we leverage the practicality of **Spring Batch** and the linker's interface become simpler, needing only the reviews of a single cluster, instead of all clusters and related reviews.

```
1  /**
2  * @class: ChangeAdvisorLinker.
3  */
4  @Override
5  public List<LinkingResult> link(String topicId,
6                                  Collection<? extends LinkableReview> reviews,
7                                  Collection<CodeElement> codeElements) {
8     List<LinkingResult> results = new ArrayList<>(assignments.size());
9     Collection<CodeElement> candidates = new HashSet<>();
10    Set<String> clusterBag = new HashSet<>();
11    Set<String> originalReviews = new HashSet<>();
12
13    this.findCandidates(assignments, codeElements, candidates,
14                        clusterBag, originalReviews);
15
16    final Collection<String> clusterCleanedBag =
17       corpusProcessor.process(clusterBag);
18
19    List<LinkingResult> similarityResults =
20       checkSimilarity(topicId, candidates, clusterCleanedBag, originalReviews);
21    results.addAll(similarityResults);
22
23    return results;
24 }
```

Listing 4.25: `ChangeAdvisorLinker` *link* method. The linker is mostly a cosmetic refactoring of the PoC version, and for the greater part, functionally equivalent.

The *findCadidates()* method, allows us to restrict our search for links. We iterate over both data sets and retain in the candidates collection (*line 9*), only those code elements that have at least one term in common between code and reviews, by computing the intersection, in the mathematical sense of the word. As we find possible candidates, we build a Bag-of-Words, the *clusterBag* set, containing only those words found in reviews, that had a possible code element candidate. Additionally, we also set aside the original sentence of the review, in order to present this to the user at the end of the process, together with the results.

We then pass our *candidate* code elements, together with the new *clusterBag* to the *checkSimilarity()* method, which can be seen in Listing 4.26, which will compute the similarity, using a `SimilarityMetric`, which will compute the vicinity of a code element and the cluster of reviews. In case the similarity scores a value over a certain *threshold*, the candidate is promoted to *link* and will be returned to the user as a result of the process.

The `SimilarityMetric` is, yet another interface, that acts as an abstraction layer for a concrete metric, allowing maximum flexibility, in future, for new ways of computing similarity.

At the end, the *checkSimilarity()* method will return a list of results for this cluster, which will be passed on to the `ClusterWriter`, which is the `ItemWriter` implementation for the linking step. Before writing the items to the database, however, it will first do one last pass over all items, setting the id of the application and the type of clustering that was executed on each result. This final step can be seen in Listing 4.27, while a class diagram of the main part of the linking step can be seen in Figure 4.11.

```
/** @class: ChangeAdvisorLinker. */
private List<LinkingResult> checkSimilarity(
                    String topicId, Collection<CodeElement> candidates,
                    Collection<String> clusterBag, Collection<String> reviews) {
   List<LinkingResult> results = new ArrayList<>();
   for (CodeElement candidate : candidates) {
      checkSimilarity(topicId, candidate, clusterBag, reviews)
         .ifPresent(results::add);
   }
   return results;
}

private Optional<LinkingResult> checkSimilarity(
                    String topicId, CodeElement candidate,
                    Collection<String> clusterBag, Collection<String> reviews) {
   final Collection<String> codeElementBag =
      corpusProcessor.process(candidate.getBag());
   if (!clusterBag.isEmpty() && !codeElementBag.isEmpty()) {
      double similarity =
         similarityMetric.similarity(clusterBag, codeElementBag);
      if (similarity >= THRESHOLD) {
         LinkingResult result = new LinkingResult(
               topicId, reviews, clusterBag, codeElementBag,
               candidate.getFullyQualifiedClassName(), similarity,
               null);
         return Optional.of(result);
      }
   }
   return Optional.empty();
}
```

Listing 4.26: `ChangeAdvisorLinker` *link* method. The linker is mostly a cosmetic refactoring of the PoC version, and for the greater part, functionally equivalent.

```
/** @class: ClusterWriter.*/
@Override
public void write(List<? extends List<LinkingResult>> items) {
   setClusterTypeAndAppNameOnResults(items);
   items.forEach(repository::saveAll);
}

private void setClusterTypeAndAppNameOnResults(
                           List<? extends List<LinkingResult>> items) {
   items.forEach(results -> results.forEach(result -> {
         result.setClusterType(this.clusterType);
         result.setAppName(this.appName);
      }));
}
```

Listing 4.27: The `ClusterWriter` sets metadata on each result, before saving.

Linking Results. Finally, we are at the end of the entire `ChangeAdvisor` process, at this point, the database has been populated with `LinkingResult` items, each item representing a link between a cluster of reviews, and a code element. The set of original reviews belonging to the cluster, is also included in the results, so that we can better present the results to the end-user of `ChangeAdvisor`, together with the cluster's terms, and the achieved similarity score.

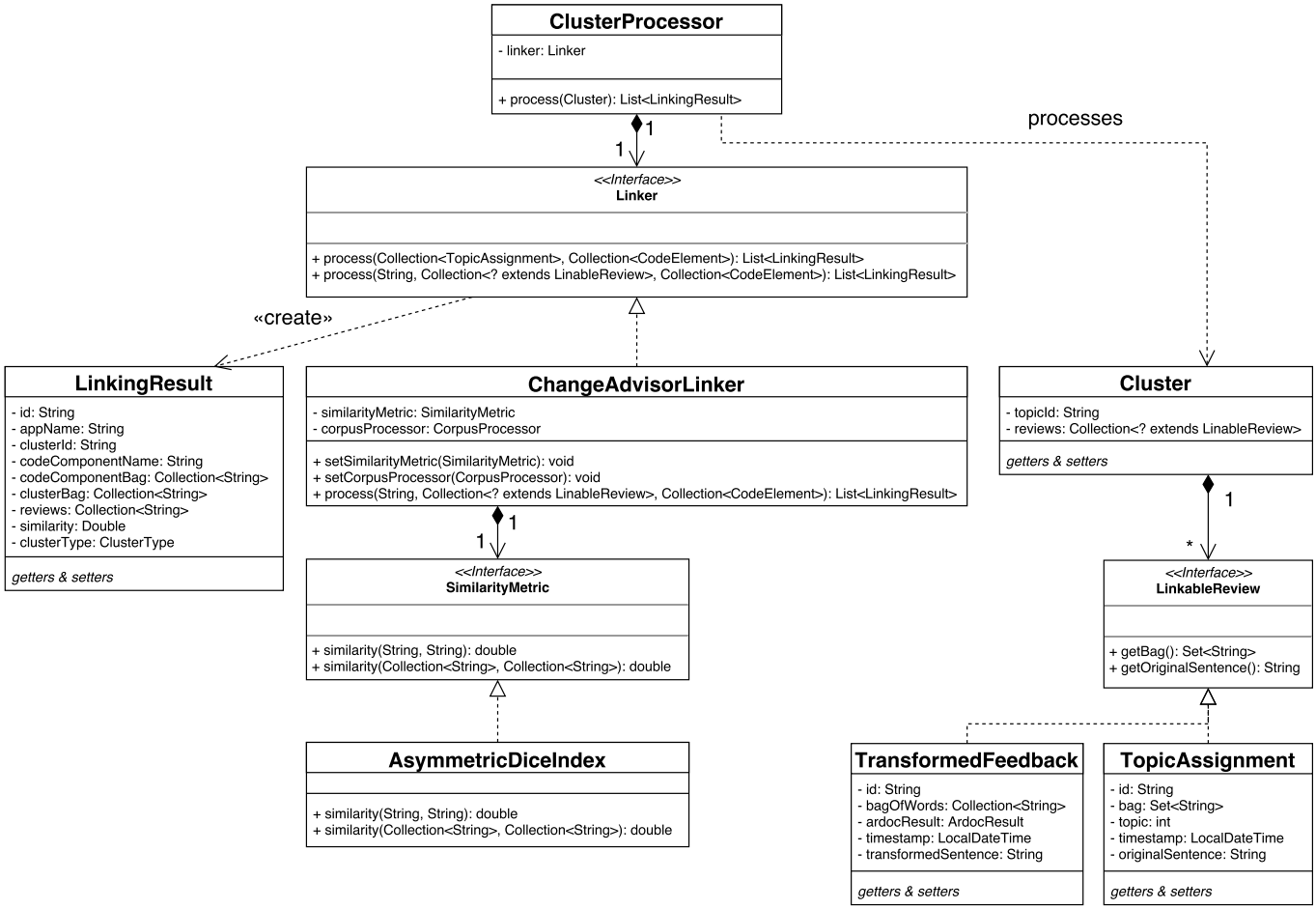

Figure 4.11: Class diagram for the linking part of the tool.

4.2.6 Persistence

During this long chapter, we have often used the term *database*, without ever explicitly saying what kind of database is in use for `ChangeAdvisor`. Thus in this chapter, we shall briefly discuss the persistence layer.

The original PoC used flat files to save user reviews. The reviews' format did not contain any metadata, rather it contained only the feedbacks' text. In the newest iteration, `ChangeAdvisor`, persists user reviews together with ratings and date of review, thus enabling the possibility, in future, for historical analysis. It would be interesting, for example, to be able to run the tools linking algorithm over a specific period in time, e.g. between the previous and latest big releases of an app, correlating not only the reviews to the code, but taking into account also rating statistics. Adding *VCS* into the mix, would then further expand the possibilities, by examining, a snapshot of the code at a specific point in time, and its commit messages.

MongoDB. This work uses **MongoDB** [Mon17] as the database for `ChangeAdvisor`, due to the following reasons:

- Ease and speed of development, due to the schema-less nature of NoSQL, allowing for faster iterations, without the need for database schema migrations.
- Compatibility with the *Review Crawler Tool* [Gra] (4.2.3), since the review crawler offers to import reviews in either **MongoDB** or in a flat file in CSV format.
- And in part, also due to personal preference, and desire to experiment with tools and technologies new to me.

The choice of the database, however, is not of particular importance to this work, which is why, the discussion regarding data storage appears so late in the thesis. Indeed, any kind of persistence layer which allows for key-value retrieval, alongside basic field search functionality, would have sufficed. This work does not attempt any fancy, real-time processing or viewing of data streams, and neither does it implement complex relationships at the database level. Indeed the data model, is rather simplistic, with much of the complexity being in the processes, rather than the relationships between data. Thus, the only functionality, needed out of a persistence layer for this work are:, (i) the capability to search by field, and (ii) basic aggregation functions, such as grouping by field.

Because of this, the choice of the database, falls into the background, when considering the overall scope of this project.

Spring Data. In order to interface `ChangeAdvisor` with the database, **Spring Data** [Piv17b] was used. **Spring Data** is a project that aims to enable developers to access various persistence stores over very different technologies (SQL, NoSQL, Graph, etc.), by defining a common abstraction layer based on the repository pattern [Fow02], and in doing so, allowing developers to significantly reduce the amount of boiler plate code needed.

The center point of the **Spring Data** abstraction is the `Repository` interface, in particular the `CrudRepository` interface, provides all of the basic *Create-Read-Update-Delete* functionality, which can be seen in Listing 4.28. From the listing: the parameter `T` is the class we want to persist, and `ID` is the type of the identifier of `T`. Through this simple interface, **Spring Data** completely hides, all technology, and vendor -specific details, greatly simplifying the interaction with the database. Indeed, to anyone, who has ever implemented a *Data Access Object* (DAO), in plain **JDBC** (Java Database Connector) or **JPA** (Java Persistence API), it should become evident the difference, in the amount of plumbing code needed.

An additional advantage of **Spring Data**, comes in the form of query derivation. **Spring Data** uses, what it calls, *query derivation* to create implementations of DAOs from interface method names. In Listing 4.29, we show an example of *query derivation*, extracted from the `ChangeAdvisor` source code. When we shall start the application, **Spring Data** parses the method declarations in the interfaces marked with `@Repository`. If it finds a method, that begins with `findBy`, it parses the rest of the method, in search for properties of the Java Bean defined in the interface declaration.

```
1  public interface CrudRepository<T, ID extends Serializable>
2    extends Repository<T, ID> {
3
4    <S extends T> S save(S entity);
5
6    Optional<T> findById(ID primaryKey);
7
8    Iterable<T> findAll();
9
10   void delete(T entity);
11
12   boolean existsById(ID primaryKey);
13 }
```

Listing 4.28: The `CrudRepository` interface, offers all the basic CRUD operations.

```
1 @Repository
2 public interface ReviewRepository extends CrudRepository<Review, String> {
3     ...
4     List<Review> findByAppNameOrderByReviewDateDesc(String appName);
5 }
```

Listing 4.29: The `ReviewRepository` interface, review query derivation.

In this concrete example, **Spring Data** shall search for us in the database, all reviews which have a given *appName*. Before returning, it shall order the results by the *reviewDate* field, in descending order. This way we can interface with the database, without ever writing a concrete implementation: the process is, in fact, entirely declarative.

More complex queries can also be implemented in other ways. Indeed, **Spring Data** does not restrict access to the persistence store in any way. One could: define native queries through the `@Query` annotation, manually implement the DAOs logic, or utilize one of the numerous abstractions implemented by **Spring Data**.

Finally, one of the last, but certainly not least advantages, is the possibility to easily switch persistence store, with little to no attrition.

4.3 ChangeAdvisor Client

As we have mentioned at the beginning of this long chapter, the client part of `ChangeAdvisor` is a thin client, meaning it contains almost no logic. The responsibilities of the client are, thus, limited to:

- presentation of results
- editing of app data, e.g. git repository url, review import schedule, etc.

- manual triggering of certain operations

Thus, we start our discussion with what the client has to offer.

4.3.1 Functionality

The UI offers, at the moment, most of the CRUD operations that the **REST API** 4.2.2 offers, albeit, not always in the most ergonomic way. For example, in order to define a schedule a user has to enter a cron expression, representing the schedule, instead of having, easy-to-use options, such as daily, weekly, etc. Additionally, not all options and functionalities, which are present in the back-end, are represented in the client-side. As an example, the clustering algorithm, defaults to **TFIDF**, without the possibility to change it from the UI, due to time constraints. It will have to be added at a later time in future. More details, regarding future works in Section 6.

Managing Applications. After firing up both the back-end server and front-end server, the user is presented a list of applications, that `ChangeAdvisor` is currently managing. Figure 4.12 shows this list of applications. From here he has the possibility to create a new project. A *Project* is an entity, that we have not really discussed up until now. It represents, *the* entity, containing all metadata for an application. In **DDD** parlance (Domain Driven Design), it is the aggregate root, for the metadata of an application, containing data such as the *Google Play Id*, *app name*, information regarding the last *source code and review imports*, schedules, etc. Through this entity and its value objects, we configure most parameters for `ChangeAdvisor`. Directly after creating a project, he can manually trigger the code and review import. By selecting one of the apps from the list, the user is brought to the *dashboard*, which is the main view of the front-end.

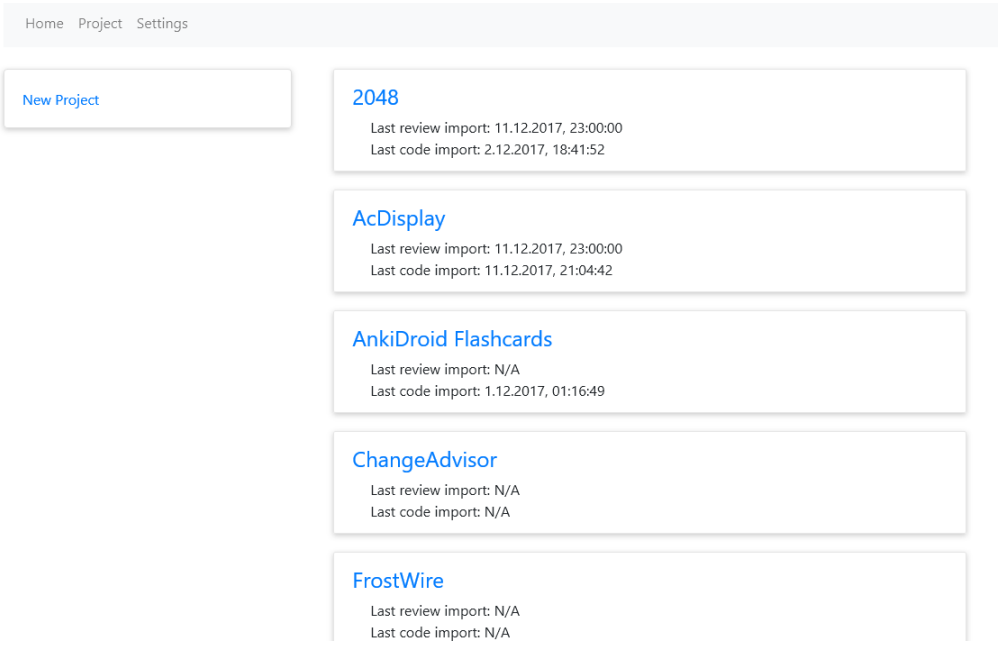

Figure 4.12: List of applications, that `ChangeAdvisor` is analyzing.

Dashboard. The *dashboard* aggregates the data that has been analyzed by `ChangeAdvisor` and presents it to the user. From here we shall also start the linking of reviews and code, which at the moment must be triggered manually. This is done this way, because the user will have the possibility to choose which clusters he wishes to see by selecting one of the labels and a category, as we can see in Figure 4.14.

First of all, we have an overview of the feedback the app has received, in the form of a time series of reviews versus average ratings. We can interact with the diagram, by changing the interval shown.

Directly below it, we can see the distribution of reviews by category with a pie-chart, and the labels that have been computed via **TFIDF** clustering 3.1.4. We can select to view up to 100 labels, and change the size of the N-grams. By selecting any label, we see how the category distribution of the reviews linked to this label, and a table with all reviews belonging to the cluster.

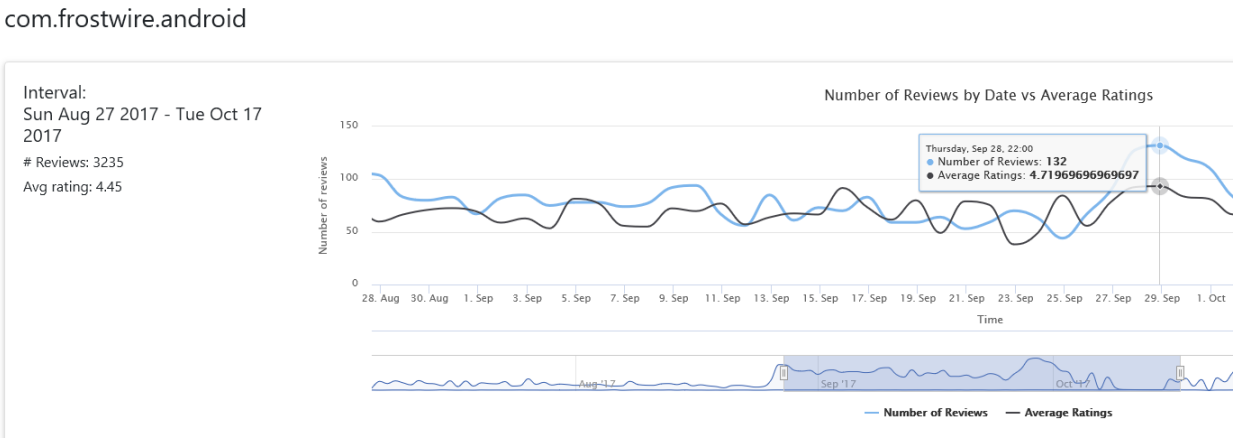

Figure 4.13: Time series of number of reviews vs. average ratings.

Results. Once a user has selected a label from the menu on the left, and selected a category from the table in the middle, the computation of the links begins, and the user is brought to the results page, which can be seen in Figure 4.15. On the left side of the results screen, is the list of all reviews, belonging to the cluster. Ideally, these reviews, represent the same change requests. On the right side, we have the code that was linked to said cluster, representing the code, that should be changed, in order to fulfill the change requests.

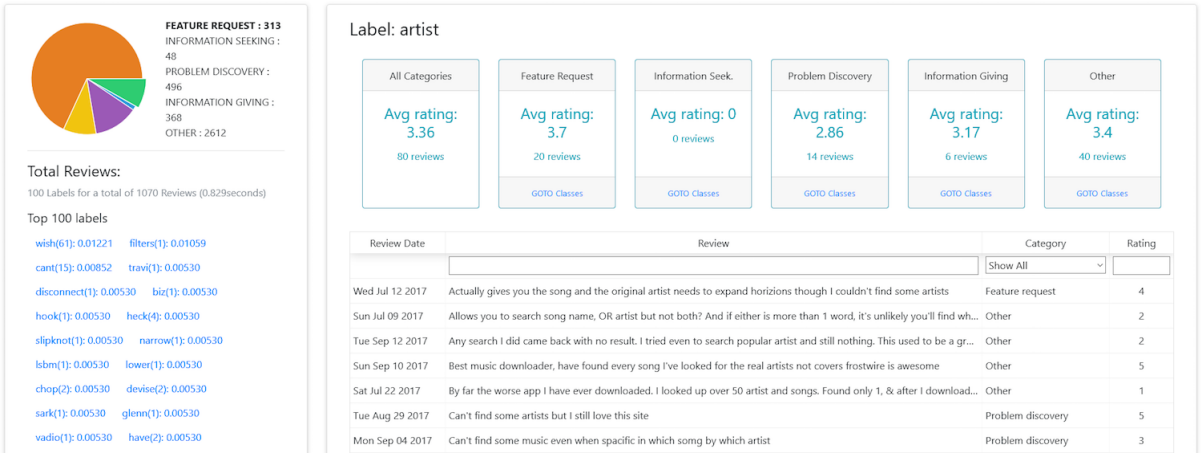

Figure 4.14: Main panel of the client-side. By clicking on *go to classes* we can view the classes linked to the cluster of reviews.

4.3.2 Tools.

The client was implemented using **React** [Fac17] a client-side web framework from Facebook, although, here any web framework would have sufficed, as well as **Vanilla JS**. Indeed, the only real requirement for a client framework was the possibility to consume a **RESTful API**, which is nowadays, possible with any client-side framework, either natively, or by importing a library. So in order to consume the API, we used the **Axios** library [Axi17], a simple promise-based, HTTP client for node.js.

Charts were implemented using two different libraries, the popular **HighCharts** [Hig17] library and **react-svg-piechart** [Rea17], a simple to use library to draw pie-charts, for node.js.

4.4 Installation and usage of ChangeAdvisor

In this section, I shall, briefly, show how the set up `ChangeAdvisor` and how to use the GUI. `ChangeAdvisor` is still in active development. Thus far, it has been tested on *Windows 10 (1709)*, and on *macOS High Sierra (10.13.1)*, both running *Java 1.8.0_73*

4.4.1 Getting Started

Installation. First of all, in order to get started we need to have the necessary software installed. The server side needs the following, in order to function:

- MongoDB 3.4

  A mongodb instance is required. For compatibility with **Spring Data**, version 3.4 of mongodb is necessary. In order to install, simply follow the instructions for the host operating system from: `https://www.mongodb.com`.

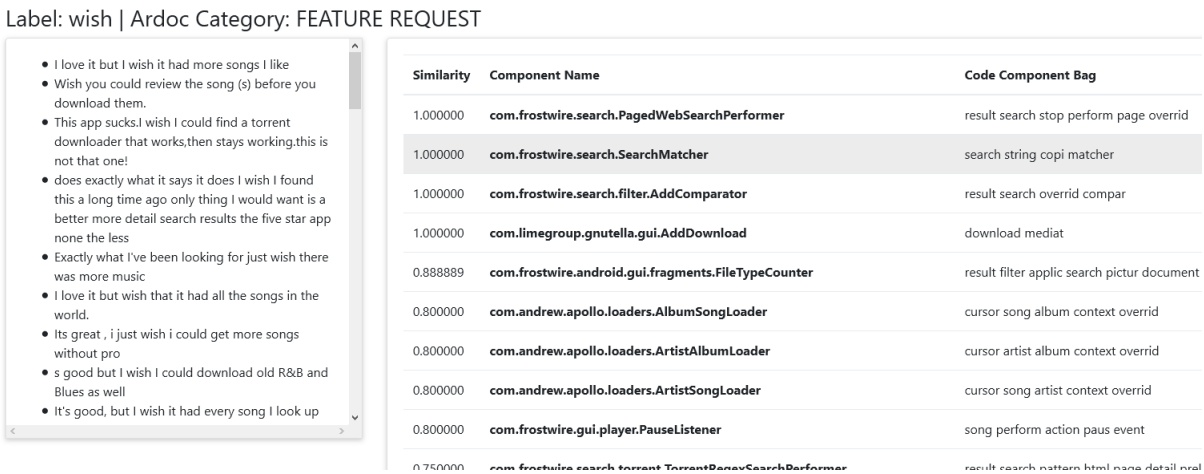

Figure 4.15: Results screen. Here we can see the reviews belonging to a cluster on the left side, and the list of linked classes on the right, sorted by similarity score.

- Java

  A Java Runtime Environment (JRE) is required in order to run the `ChangeAdvisor`. It needs to support at least Java 8.

- Python

  In the following, we show how to start the client static web server using Python, as it is the simplest and fastest way. But, any other web server can be used, e.g. Apache, Nginx, etc.

Once the dependencies are installed, we can download the most recent version of the build from VCS[1].

Running. Under the `release` folder we find the `ChangeAdvisor` jar file, a `dump` folder containing data for demonstrative purpose and the client-side files, which can be deployed to any static web server. For the next following steps, we shall need three terminal windows [2], *cmd/Powershell*, in windows, or *Terminal* in macOS.

First, start mongodb, by running the following command from the CLI:

```
mongod
```

[1] `https://bitbucket.org/alexander_hofmann/changeadvisor`
The link can also be found in one of the cover pages.

[2] An alternative to multiple CLI windows would be to use a multiplexer, or running the applications in background.

In order to import the database, run the following command from the `release` directory. This is needed only the first time, and the mongodb instance needs to be running in order for this to work.

```
mongorestore dump
```

Afterwards, we can run the server code by executing `changeadvisor.jar` as an executable:

```
./changeadvisor.jar
```

If Python is installed on the host machine, we can start a static web server by running the following command from the root of the `release/client` folder, in a new CLI session:

```
python -m SimpleHTTPServer 8000
```

This is all that is needed, in order to start the server. By browsing to `localhost:8000` with a browser, we are greeted by the `ChangeAdvisor` project selection screen (Figure 4.12). By selecting *com.frostwire.android*, we are brought to the main dashboard, where we can see the time series of reviews (Figure 4.13), and directly below it, the table with the main labels, computed using **TFIDF** (Figure 4.14). Let us select the first label: *wish*. The table should have populated itself with the reviews associated with the selected token. By clicking on the *GOTO Classes* link directly inside the *Feature Request* panel, we are brought to the results screen. After a few seconds, the table should look as it does in Figure 4.15, containing all results found by linking the reviews related to the label *wish*, with the source code of the application.

Chapter 5

Evaluation

This work set out with the goal of rewriting the original Proof-of-Concept, going beyond the PoC stage, as a full-fledged application, with the specific goal to improve aspects such as performance, maintainability, extensibility, and usability.

This work, however, is not a research thesis. As such a *quantitative* review of the results, is not possible. There are no experimental values, and no hypothesis to be confirmed or confuted. Rather, we do a *qualitative* evaluation, by reviewing the aspect mentioned above, the goals set out, and by comparison with the PoC.

It is important to remember however, that the difference between `ChangeAdvisor` and the PoC is considerable. The previous iteration of `ChangeAdvisor` was developed as a tech demo and as a throwaway PoC and because of this, aspects such as maintainability and extensibility were, at the time, a non-issue.

We review `ChangeAdvisor`, under the following aspects:

- Maintainability: ease, for a developer, to apply fixes in a timely manner; brittleness of the system.
- Extensibility: openness to extensions, with or without code changes.
- Performance: speed of execution and memory consumption.
- Usability: ease of use for the end-user, flexibility for future research.

5.1 Maintainability

Refactoring and bug fixing is a natural process of each software project. Even with good requirements and top developers, code has to be flexible enough to be modified: bugs will always find a way to slither in, code has to be changed to reflect the product owner's vision and preference, new tools, libraries, or techniques might be introduced for many different reasons. Over the course of this project, bugs were, naturally, introduced, and requirements slightly changed and thus a certain deal of refactoring and bug fixing was needed.

As we have seen in Section 4, the core of the business logic is implemented through the use of **Spring Batch**. **Spring Batch** makes it easier to implement processes, defined as a list of steps, promoting *single responsibility principle* and *loose coupling*. As such, many of the main components in the *pipelines*, work completely independently from one another, and can also be tested in complete isolation. This, simple, fact greatly helps, when modifying code.

A concrete example of this, was when **TFIDF** was added to `ChangeAdvisor`, the entities used as input for the *linker* were different, depending on the clustering algorithm used. Through a

simple generalization in the linker interface (`LinkableReview`, see 4.2.5), the processor only had to modify its input parameter to that of the generalization. Then, we added a new `ItemReader` for the new input type, which could, thanks to **Spring Batch**, simply be plugged in the definition of the job.

A small test to the maintainability, will be the fine-tuning process that `ChangeAdvisor` will undergo in future. While all processes were implemented, mostly as per the PoC, it became apparent during development, that some processes could be streamlined. Some steps in the preprocessing stage for example might be omitted. The linker step also does some string manipulation, which might not be needed, considering prior preprocessing.

Over the course of this project, we strived to always apply software engineering principles and best practices, in order to improve the maintainability of the system. This is reflected in the following examples:

- Interfaces and composition, when possible, following the *strategy pattern*.
- Inversion of Control (IOC), through Spring's dependency injection
- Separation of concerns, e.g. `Item{Reader, Processor, Writer}`, decoupled client and server.
- Code re-use, e.g. review and source code preprocessing, use of well-established libraries and frameworks.

Thus, under the maintainability aspect, the current version of `ChangeAdvisor`, is more advanced than the original PoC.

5.2 Extensibility

Extensibility refers to the openness of a system to extension, both with code changes and without. It is the degree to which a system can be changed, while also considering the amount of effort required for such change to happen. Examples of this are: the addition of a new feature, or the introduction of a new UI in the form of a mobile app.

The PoC was non-extensible, as it did not need to be. This work now exists to apply the original approach in a new framework, allowing it to become extensible, as we have seen above. Thus, we identify the two main assets providing extensibility to `ChangeAdvisor`, at the macroscopic level:

- Batch approach
- Decoupled front- and back-end

Extensibility through Batch Approach Under this aspect, `ChangeAdvisor` should be primed for success, indeed the designed batch approach, makes the existing jobs (*pipelines)* flexible enough to swap out components. It allows for the design of entirely new jobs, while leaving the existing pipelines unaltered, since they are independent of one another. Considering, also, the classic layered architecture of the back-end server, implemented in `ChangeAdvisor`, it makes it particularly easy to define new processes, and connect these processes to the API for triggering and then consumption of the results.

Decoupled front- and back-end The provided REST API layer, allows for unlimited extensibility in terms of clients. Indeed, the ubiquity of HTTP clients, means that it is possible to extend `ChangeAdvisor` with new UIs and clients, for virtually any kind of system. Possibilities for the future include, new web clients, mobile apps, and plug-ins for IDEs. It would be interesting, to integrate `ChangeAdvisor` into existing CI/CD pipelines. For example, with every new commit to the master branch that triggers a redeploy of the application, the source code is also analyzed, and the new results are integrated with the reviews imported between this analysis and the previous one. This way, we could run the process against the version of the code that is actually in use by end-users.

5.3 Performance

Performance comparisons between the PoC and the current iteration of `ChangeAdvisor` is an interesting aspect, as it is not as clear-cut as with the other facets. It rather is much more of a trade-off between memory consumption and speed of execution. By having a back-end and front-end server running, we consume overall considerably more memory than the PoC, since once the PoC is done, it can exit, freeing all of its memory. In the case of servers, we constantly have a set of threads hogging the memory, since it has certain availability requirements, that a command line interface (CLI) process does not have. On the other hand, overall speed of execution is dramatically increased. Often during development of the core `ChangeAdvisor` process, we compared the results and execution times of the PoC versus those of the newer implementation and found that each step of the process, is quicker. An exception is made for the source code and reviews import. With the PoC these are responsibility of the user. So it is not possible to compare the performance of these steps with the new version.

String manipulation. The biggest difference, in terms of execution speed, is in those steps, which are particularly heavy in string manipulation. This includes the preprocessing steps, and the `ChangeAdvisorLinker`. Especially, the performance of the preprocessing stage, is orders of magnitude better. One of the main reasons we have identified is that the PoC heavily relied on string concatenation using the "+" operator. Duplicate removal, as an example, works as follows: (i) input to the method is passed as a single string, (ii) which is then split using the *split()* method, (iii) to create the result, we join the tokens back together using the "+" operator, but first we check, using the *contains()* method, whether the token is already contained in the result string. The performance overhead caused by this approach is significant. The are two problems with this approach:

- string concatenation using the "+" operator, means that, for each concatenation, we must first create a character buffer large enough to contain both tokens, in to which, we then copy each character. Doing this operation for every token of every document in a corpus, becomes costly, when the size of the corpus is in the order of thousands. In fact, a simple computation shows, that its complexity is $O(n^2)$, in other words, it has a quadratic running time [1]. In some cases the compiler can optimize this process by using a `StringBuilder` instance. However, in the case of dynamic data, i.e. data not know at compile-time, it cannot do this optimization. Explicitly using a `StringBuilder` brings the running time down to $O(1)$ amortized.
- use of the *contains()* method on a string, means we need to iterate over each character of the increasing result string, to match the token we are searching for. This results in $O(n)$

[1] `http://www.pellegrino.link/2015/08/22/string-concatenation-with-java-8.html`

complexity. By using a `Set` for the removal of duplicates, we get $O(1)$ running time, since the *contains()* is optimized for this operation. In fact, it computes a hash of the value we are searching for, and uses that as the index of the backing collection, in order to find the searched token.

Some manual testing showed, that for the linking step, which also does extensive string manipulation, the run-time of the PoC was around 60 minutes, while in the new version, it was about 1 minute, while achieving similar results (small differences in how the tokens are manipulated lead to a variation in the results).

Clustering. The PoC implemented the **HDP** [TJBB05] in Python. For this work, we ported the Python code in Java. Thus, the only difference in terms of speed is given by the speed of the Python interpreter present on the system, versus that of the Java compiler and JVM running on the same system. On the machines tested, the execution time of the Java version was about 10 times quicker. However, no actual benchmarking was done, and performance, especially for Python, greatly depends on the interpreter used, thus these values are to be taken with a grain of salt. In any case, compiled languages, do tend to have an advantage, in terms of speed, over interpreted languages. It is important to note, that these values come from observations made while developing the newer iteration, the goal then was never to measure speed of execution, but rather to check whether the results were comparable.

5.4 Usability

Usability refers to the ease of use of a tool, i.e. how easy it is to use and to learn. Usability is a make-it or break-it criterion: no one wants to be stuck with a tool that is annoying to use, and is one of the main reasons for the complete rewrite of `ChangeAdvisor`. The previous iteration of the tool, was a *CLI* software. Considering the target audience of `ChangeAdvisor`, it is not a matter of learning, but rather continuous use: indeed, even though, most developers should be comfortable enough with a shell, it does not mean that they might want to spend any more time than necessary in it. Additionally, a CLI precludes the possibility for the visual exploration of results. The field of data visualization has already shown the importance of visually analyzing data, in order to make even extremely large data sets digestible, which in turn leads to better decision making. Another disadvantage of the PoC, in terms of usability, is the way the reviews are fed to it. In order to start the process, we must create a flat file containing all user feedback. Which means that the developer needs to, by some other means, export their reviews and then create this file. This process has to be done periodically, as new reviews come in. This, greatly, hurts the usability of the PoC, since this process, ends up involving multiple tools, just to start `ChangeAdvisor`.

The new version of `ChangeAdvisor` should represent a major step forward. Indeed, the possibility to automatically import the reviews from the *Google Play Store* on a schedule and the source code from *VCS*, significantly lowers the effort needed to use the tool.

Additionally, this newer version, thanks to its graphical interface, allows for the possibility to better analyze the results of `ChangeAdvisor`. This is not only an advancements, in terms of ease of use, but also of overall pleasure of using the tool.

Future works, might implement historical analysis of feedback, thanks to the database backing `ChangeAdvisor`, which would enable for even more interactivity and functionality with the system, increasing the usability of the tool.

Chapter 6

Conclusions and Future Work

This thesis had the following main goal: *to implement the* `ChangeAdvisor` *approach [PSC+18], as a newly redesigned application, which could then be the basis for future work.* Thus, two of the main non-functional requirements for this tool are *maintainability* and *extensibility*. For this purpose, the application was designed to work with two parallel data pipelines. One for user feedback, and one for source code. These two pipelines, which are described in Chapter 3, would join at the end to provide the user with insight into the change requests, contained in the review data set, and which code components would need to be changed, in order to fulfill the requests. On the maintainability side of things, each pipeline is designed as a list of step (Chapter 4), where each step is largely independent of each other, only needing to know the output of the previous step in order to process it. This makes fixes and refactoring easier, thanks to the separation of concerns, and the possibility to test these components in isolation. Regarding extensibility, the approach only defines the interface, with which the two pipelines merge, making it simple to replace, add or remove steps, or even add new pipelines. This, together with the REST API, leaves the system open for future extension.

During development, we often drew comparison between the version under development and the PoC. Both in terms of performance (memory consumption and execution times) and of similarity of results. As well as, in terms of usability, maintainability, and extensibility (Chapter 5).

We strongly believe, that this work achieves its goal of setting the ground work for future research scenarios, as well as,increasing the usability of the system for the end-user through the addition of the web UI.

6.1 Future Work

A significant amount of work still remains, and new possibilities have opened up. Parts of the pipeline need fine-tuning, and some bugs unfortunately still persist in the latest version of `ChangeAdvisor`, as is normal for a project of this size. The most interesting part starts afterwards. Indeed, considering that the goal was to set the basis for future development, from here on out, we have many new possibilities, both on the research side of things, as well as features for end-users. The `ChangeAdvisor` approach is implemented considering Java code, since it is the de facto standard language for Android. However, in the last few years, new languages have come out for the platform, e.g. Kotlin, React Native, C# (Xamarin), etc. These languages are not supported in `ChangeAdvisor` at the moment. Adding support for them would, however, be trivial. Indeed, the approach described in Chapter 3 is language agnostic. The only necessary addition would be to add parsers for the targeted languages, while the rest of the pipeline could remain unaltered. It might be interesting to add support for iOS and Apple's *App Store*. Indeed a huge slice of the market is mostly neglected from researchers in favor of Android. This is due to

the fact that there is a much larger concentration of open source app for Android than iOS. However, if `ChangeAdvisor` starts seeing use from developers, a collaboration between academia and industry might be possible, allowing research also inside of Apple's *walled garden*.

Regarding the source code import functionality, other *version control systems* (VCS) could be added, and VCS hooks functionality would be beneficial for usability purposes. Regarding the existing VCS functions, new research paths are open: it would be interesting to study the relation between commits, commit messages, and changes in rating. Trivially, we would assume that commits referring to code fixes would cause an increase of the average rating. However, changes in code might also have side-effects, such as breaking other functions, which might push the ratings in the other direction. Thus, `ChangeAdvisor` might benefit from the additional information contained within commit messages, to increase the precision of its approach.

Furthermore, the introduction of the database opens up an entire new dimension to exploration: time. Indeed, in contrast with the PoC, we now also have access to the reviews dates and the commit history through VCS. This combination allows for new approaches, not only based on code metrics and similarity with reviews, but also considering review metadata, release schedules, and as mentioned before, commit messages.

Finally, testing new clustering algorithms for topic modeling and new metrics for computing similarity, might bring increases in precision.

Considering all these aspects, we feel excited at the prospects for `ChangeAdvisor` and in general for the field of *App Store Mining*. I, personally, hope to be able to extend this tool in the near future, maybe in the form of a follow-up project.

Bibliography

[AAT10] Hazeline U Asuncion, Arthur U Asuncion, and Richard N Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 95–104. ACM, 2010.

[Axi17] Axios, promise based http client for the browser and node.js. `https://github.com/axios/axios`, December 2017.

[Aza15] Shams Abubakar Azad. Empirical studies of android api usage: Suggesting related api calls and detecting license violations. April 2015.

[BLVBC+15] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, 2015.

[CG12] Rishi Chandy and Haijie Gu. Identifying spam in the ios app store. In *Proceedings of the 2nd Joint WICOW/AIRWeb Workshop on Web Quality*, pages 56–59. ACM, 2012.

[Cha17] ChangeAdvisor. Recommending and localizing code changes for mobile apps based on user reviews. `https://sites.google.com/site/changeadvisormobile/`, November 2017.

[CLH+14] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. Arminer: Mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 767–778, New York, NY, USA, 2014. ACM.

[CLO+13] Giovanni Capobianco, Andrea De Lucia, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Improving ir-based traceability recovery via noun-based indexing of software artifacts. *Journal of Software: Evolution and Process*, 25(7):743–762, 2013.

[DSPA+17] Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Corrado A Visaggio, and Gerardo Canfora. Surf: Summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 55–58. IEEE Press, 2017.

[Ecl17] Foundation Eclipse. Jgit. `https://www.eclipse.org/jgit`, December 2017.

[Fac17] Facebook. React, a javascript library for building user interfaces. `https://reactjs.org/`, December 2017.

[Fou17] Apache Software Foundation. Apache commons. https://commons.apache.org, December 2017.

[Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gra] Giovanni Grano. Reviews crawling tool. https://github.com/giograno/reviews_crawler.

[Hid17] Ariya Hidayat. Phantomjs, full web stack no browser required. http://phantomjs.org, December 2017.

[Hig17] Highcharts. https://www.highcharts.com/, December 2017.

[HJZ12] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 108–111, June 2012.

[HW13] E. Ha and D. Wagner. Do android users write about electric sheep? examining consumer reviews in google play. In *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*, pages 149–157, Jan 2013.

[Jav17] JavaParser. Javaparsers, process java code programmatically. http://javaparser.org/about.html, December 2017.

[JON72] KAREN SPARCK JONES. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[KNH16] Hammad Khalid, Meiyappan Nagappan, and Ahmed E Hassan. Examining the relationship between findbugs warnings and app ratings. *IEEE Software*, 33(4):34–39, 2016.

[KSNH15] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, May 2015.

[lan] Languagetool.

[MN15] Anas Mahmoud and Nan Niu. On the role of semantics in automated requirements tracing. *Requirements Engineering*, 20(3):281–300, 2015.

[Mon17] MongoDB. Mongodb. https://www.mongodb.com/, December 2017.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[MS10] Susan M. Mudambi and David Schuff. What makes a helpful online review? a study of customer reviews on amazon.com. *MIS Q.*, 34(1):185–200, March 2010.

[MSB+14] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[MSJ+16] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.

[MSM93] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *COMPUTATIONAL LINGUISTICS*, 19(2):313–330, 1993.

[NGKA11] Nasir Naveed, Thomas Gottron, Jérôme Kunegis, and Arifah Che Alhadi. Searching microblogs: coping with sparsity and document quality. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 183–188. ACM, 2011.

[PB13] Dennis Pagano and Bernd Brügge. User involvement in software evolution practice: A case study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 953–962, Piscataway, NJ, USA, 2013. IEEE Press.

[PDO+13] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 522–531. IEEE Press, 2013.

[PDSG+15] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 281–290, Washington, DC, USA, 2015. IEEE Computer Society.

[PDSG+16] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. Ardoc: App reviews development oriented classifier. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1023–1027. ACM, 2016.

[Piv17a] Pivotal. Spring batch - reference documentation. `https://docs.spring.io/spring-batch/trunk/reference/html/index.html`, November 2017.

[Piv17b] Pivotal. Spring data mongodb. `https://projects.spring.io/spring-data-mongodb/`, December 2017.

[Piv17c] Pivotal. Spring mvc - reference documentation. `https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html`, November 2017.

[Por80] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[PSC+18] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald Gall, Filomena Ferrucci, and Andrea De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE 2017, page to appear. ACM, 2018.

[Rea17] react-svg-piechart. `https://www.npmjs.com/package/react-svg-piechart`, December 2017.

[Sel17] Seleniumhq, browser automation. `http://www.seleniumhq.org`, December 2017.

[SNAH15] Mark D Syer, Meiyappan Nagappan, Bram Adams, and Ahmed E Hassan. Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal*, 23(3):485–508, 2015.

[Sof17] SmartBear Software. Swagger. `https://swagger.io`, November 2017.

[Son17] SonarSource. Continuous code quality. `https://www.sonarqube.org/`, November 2017.

[TJBB05] Yee W Teh, Michael I Jordan, Matthew J Beal, and David M Blei. Sharing clusters among related groups: Hierarchical dirichlet processes. In *Advances in neural information processing systems*, pages 1385–1392, 2005.

[TNLH15] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E. Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, pages 301–310, Washington, DC, USA, 2015. IEEE Computer Society.