Evolution-aware API analysis of developer skills
Evolutionsorientierte API-Analyse von Entwickler Fähigkeiten

Institut für Informatik
Universität Koblenz-Landau

Abschlussarbeit

zur Erlangung des akademischen Grades
Master of Science

vorgelegt von

Hakan Aksu
geboren am 03.10.1989 in Hagen

im März 2015

Erstprüfer: Prof. Dr. R. Lämmel
Zweitprüfer: A. Varanovich

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit selbstständig und nur unter Verwendung der von mir angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Datum: ______________________ Unterschrift: ______________________

Danksagungen

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Master-Arbeit unterstützt und motiviert haben.
Ganz besonders gilt dieser Dank Herrn Prof. Dr. Ralf Lämmel, der meine Arbeit und somit auch mich betreut hat. Nicht nur, dass er immer wieder durch kritisches Hinterfragen wertvolle Hinweise gab, auch seine moralische Unterstützung und Motivation waren sehr hilfreich. Vielen Dank für die Geduld und Mühen.
Mein Dank gilt auch an meinen Komillitonen und Freund Erwin Schens. Die zahlreichen Disskussionen waren sehr hilfreich und haben mich immer wieder auf neue Gedanken gebracht. Vielen Dank Erwin.
Daneben gilt mein Dank meiner Familie, die emotional immer für mich da waren. Auch an Tagen an denen ich nicht voran kam, haben sie mich motiviert und zum arbeiten angeregt. Ich kann mich glücklich schätzen, euch alle an meiner Seite zu haben.

Zusammenfassung / Abstract

Zusammenfassung

Wir analysieren versionsbasierte Software Projekte um den Entwicklern API- und Domänen-Wissen zuzuordnen. Genauer gesagt analysieren wir die einzelnen Commits in einem Repository in Hinblick auf die API-Nutzung. Auf dieser Grundlage können wir APIs (oder Teile davon) den Entwicklern zuordnen und dadurch auf die API-Erfahrung der Entwickler schließen. Im transitiven Schluss können wir auf Domänen-Erfahrung schließen, da jeder API eine programmierdomäne zugewiesen wird.

Abstract

One task of executives and project managers in IT companies or departments is to hire suitable developers and to assign them to suitable problems. In this paper, we propose a new technique that directly leverages previous work experience of developers in a systematic manner. Existing evidence for developer expertise based on the version history of existing projects is analyzed. More specifically, we analyze the commits to a repository in terms of affected API usage. On these grounds, we associate APIs with developers and thus we assess API experience of developers. In transitive closure, we also assess programming domain experience.

Contents

Eidesstattliche Erklärung . . . I
Danksagungen . . . II
Zusammenfassung / Abstract . . . III

1 Introduction 1
1.1 Problem Context . . . 1
1.2 Research Problem . . . 2
1.3 Research Contributions . . . 2
1.4 Thesis Structure . . . 2

2 Research Questions 4

3 Related Work 6

4 Basic Concepts 8
4.1 Terms . . . 8
4.2 Metrics . . . 9

5 Methodology 11
5.1 Software Project . . . 11
5.2 Data Model . . . 11
5.3 Fact Extraction . . . 15

6 Infrastructure 22

7 Discussion of Results 24

8 Threats To Validity 30

9 Conclusion 32

1 Introduction

This Chapter sets up an introduction to this thesis. In Section 1.1 the problem context is described and in Section 1.2 the general research problem is stated. Section 1.3 gives a short overview of the achieved research contributions of the thesis and in Section 1.4, the structure of the thesis is described.

1.1 Problem Context

One task of executives and project managers in IT companies or departments is to hire suitable developers and to assign them to suitable problems matching their expertise. Interviews, questionnaires, assignments, and publicly available information (e.g., on topcoder[1] or stackoverflow[2]) may be used to determine individual developer skills. The previously stated methods are known to be "problematic". For example assignments are not suitable to give an overview of programming skills of the developer or how the developer acts under certain circumstances. Those assignments are used to cover only a certain knowledge base. Interviews and questionaires give us only a limited and subjective view on the knowledge of the developer in selected topics. Those specific impressions are made during a very short time period, but they don't give an overview of the whole skill set of the developer. As previously stated, executives and project managers can use publicly available information (e.g., on stackoverflow) to extend their impression of the developer but the developer must be frequently active on those platforms to receive useful information about the developer skills.

It is desirable for executives and project managers to receive an objective instead of a subjective view of the developer. Therefore, in most cases it would be enough to know in which programming domain or API usage they are good at. To receive this objective view the previously stated "problematic" methods are not suited to determine the API-related developer expertise on an objective level. We are in need for a new technique or method to approach and solve this problem and determine the API-related developer expertise. If we could achieve this expertise based on recent software projects we could assign the developer to a better suited position in a company or the next project.

[1] http://www.topcoder.com/
[2] http://stackoverflow.com/

1.2 Research Problem

As described in Chapter 1.1 we need a new approach to determine the API-related developer expertise. The general problem is that we need an objective and statistic based value to measure the API-related developer expertise. We need to create an application to obtain semi-automatically API-related developer expertise. Then the system can be used to measure and evaluate the developer experience with a certain API. To create such a system we need a code base of the developer based on previous software projects. This code base can be created by the developer alone or in cooperation with others. In common, code bases are created in teams. To manage such a code base, a version control system (VCS, see Chapter 4) is used. With a VCS we are able to extract changes made by individual developers in a timely manner. This information extraction made by the VCS can be used to analyze the commit behavior of the individual developer. This commit behaviour can be used for further analysis. The system described in the beginning must be able to retrieve information from all the versions in the VCS and use those to extract the API usage of the developer.
Another problem which arises from this scenario is that we cannot guarantee that every version in the VCS is buildable or resolvable. Projects though need to be buildable or resolvable, if we wanted to use straightforward techniques for analysis. Once we have buildable or resolvable projects we can create easily the relationships between source code and used APIs. Therefore, we need to find a different way to create this relation. Based on this scenario, a number of research questions are raised. Those questions are described in detail in Chapter 2.

1.3 Research Contributions

We contributed a new technique to measure objectively and statistically the API-related developer expertise using a *Mining Software Repository (MSR)* method. To use this technique a semi-automatically application was developed which retrieves the important information of a VCS and analyzes the changes made by the developer. We also analyze API usages in those changes. With this analysis we can determine the API-related developer expertise. In transitive closure, we also assessed programming domain experience. This system can be used for example by executives and project managers to help them to evaluate API-related developer expertise. Additionally a new technique was developed which is based on a lexical analysis [1] and [2] to get the relation between the changed lines of code and an API.

1.4 Thesis Structure

This thesis is structured into nine chapters. The Chapters 1 and 2 give an abstract overview of the thesis domain. Then Chapter 3 shows related work on which this thesis is built on. In Chapters 4, 5 and 6 the developed system which is used for the analysis is described in detail. The last three Chapters 7, 8 and 9 show results of the application of the system

followed by a discussion and a list of threats to validity.

In Chapter 2 we set up a list of research questions which are based on the research problem described in Section 1.2. In Chapter 3 we summarize and evaluate the papers in the "Mining Software Repositories" (MSR) field. In Chapter 4 the basic concepts are described to create an essential knowledge base. In Chapter 5 we describe our methodology. In Chapter 6 we describe used tools and APIs and in Chapter 7 we show and discuss our results which we received from our research. In Chapter 8 we demonstrate the threats to validity. In the last Chapter 9 we summarize our work and discuss about future work.

2 Research Questions

RQ1: ***How can we semi-automatically obtain API-related developer expertise from the analysis of existing software projects with version history using MSR methods?***

RQ1.1: ***How can we measure API-related developer expertise?***

There is no clear way to measure API-related expertise in a specific way. We need a value to compare the API-related expertise between the developers. In addition, we have to be able to tell if the developer has experience or not. We can determine the practical experience of a developer through a specific usage of an API. We call this practical experience API-related developer expertise. Therefore, we have to look at how often a developer has used a particular API. The frequency of usage of an API may be a good indication of the API-related developer expertise.

RQ1.2: ***What is the important information from the version history to get the API-related developer expertise?***

There is a lot of information in every commit to the repository. From those information we have to take the right ones. Commits where, for example, only files have been moved and copied, are not to be considered as experience. Therefore we need to consider only the files which are written by the developer and are processable as source code. In other files we don't find API usage and accordingly these files are not necessary for the API-related expertise. The modified source code in a commit can consist of deleted and added lines in which API elements are used. In our proposed technique, deleting a line is treated as experience and inserting a new line is also treated as an experience. For these and similar cases, we need to define metrics which contains the important information.

RQ1.3: ***How can we create a system to obtain semi-automatically API-related developer expertise?***

In our proposal we have many steps to obtain automatically and manually the API-related developer expertise. For example, we have to search manually for API-files (e.g., *.jar* files) from the web and we have to automatically extract the information of the repository and the APIs. Therefore we need a semi-automatically system to obtain the API-related developer expertise.

RQ2: ***How can we create a connection between changed lines of code and the API without building the underlying project?***

If we had buildable projects we can create easily the relationships between source code and used API. A problem is that we cannot guarantee that every version is buildable. Therefore, we need a different way. In each version, we need to look at only the changed lines and determine which API was used there. This could be solved with a lexical approach. We could tokenize a changed line and check for used API elements (e.g. classifier or method names). In addition, we can restrict the search using only API elements from the imported packages. Results which we receive from such an approach can be influenced by the fact that two different imported packages can contain a method with the same name. In this case a clear assignment to an API is not possible. The expectation is that this does not often occur.

3 Related Work

This effort relates broadly to these research areas:

- Analysis of API usage; see, e.g., [3, 4, 5, 6]
- Analysis of changes along evolution; see, e.g., [7, 8].
- Analysis of developer activity; see e.g. [9, 10]

Our project combines all three areas with some emphasis on the last one, as far as the need for new techniques is concerned. That is, we aim at analyzing and interpreting developer activity in terms of changes along evolution based on indicators of API usage. Each of these research areas is described more in detail below.

Analysis of API usage

An important research topic which this thesis builds on is the analysis of API usage. Further work on this area and how it influences our research are described.
In [3] a tool named MAPO is described. MAPO is used to find the common usage of an API based on information extraction of open software repositories. It was developed due to the fact that most APIs are not well documented and therefore it is hard to extract usage information. With this approach they extract the lines of code where a specific API is used. We have a contrary approach where we search in line of codes for API usage. The fact that MAPO does not involve developers in its search it would not be possible to establish a relation between the developer and the lines of code. Additionally we would need to have a deeper knowledge of the APIs used in the analyzed software project. If the previous issues don't exist then we could iterate over all versions and apply MAPO for all API usages. In [6] the approach from [3] is extended by adding quality metrics to increase the quality of the search result. Another API usage analysis can be found in [5] where the software project JHotdraw is analyzed for API usage and is building the foundation to our research. This analysis consists of only one buildable version of JHotdraw and shows the frequency of the API usage. It does not include the developers and the evolution of the software project in its analysis.
The evolution happens mostly in VCS repositories. But how can we analyze the API usage if some revisions of the repository are not buildable? (RQ2). In [1] and [2] approaches are discussed for syntactical and lexical analysis of source code. The syntactical analysis is a grammar-based approach which is performed by a parser. The types of representations are a *parse tree* and a *abstract syntax tree*. With the syntactical analysis we are not able to associate the API usage directly. The suited approach for our proposal is the lexical analysis.

The changed files and lines are the important information. Independently of the rest we can work on only the changed lines. We can tokenize the lines and analyze the tokens to identify the API usage.

Analysis of changes along evolution

In [7] a new change-based repository is proposed where the analysis to a system is improved. Disadvantages of current VCS are described and avoided in the new change-based repository. The change based repository is using an incremental approach where every change inside the IDE is recorded. The current VCS records only snapshots of changes. In our research it is enough to have only those snapshot information because it is not important to our approach to analyze data which led to the result of the commit. In addition such a change based repository is not common in developer communities.
The evolution of an API may be analyzed to guide the implied migration work on projects that use an API [11].
In [12] a visualization tool is demonstrated. It visualizes the information of CVS repositories. The approach of the tool can be used for our research to identify the changed lines of source code and the developer in subversion repositories.
In versions of VCS only deleting and adding of lines exists. In [8] a solution is presented where it is possible to locate actual changed lines with the help of the levenshtein distance. In our research we assume that the developer is gaining equal experience by deleting, adding or changing a line.

Analysis of developer activity

In [9], interactions of distributed open-source software developers are analyzed. Data mining techniques are utilized to derive developer roles. The underlying modeling and data mining techniques may be also be applicable, to some extent, to our problem in that the concepts of developer roles and (API-related) developer skills are not completely different. The concrete open-source projects of this work (ORAC-DR and Mediawiki) may also provide a starting point for our corpus.
In [10], statistical author-topic models are applied to a subset of the Eclipse 3.0 source code. The authors state that this technique provides an intuitive and automated framework with which to mine developer contributions and competencies from a given code base. The resulting information can serve as summary of developer activities and a basis for developer similarity analysis. The technique may be also be applicable, to some extent, to our problem, if we manage to identify API “topics” on the grounds of API definitions (types and method signatures) and extra metadata about domains and API facets [5].

4 Basic Concepts

This chapter sets up the basic concepts underlying this thesis. It is divided into two sections. The first Section 4.1 introduces basic terms which are necessary for this thesis. In Section 4.2 metrics are introduced which will be used to measure API related developer expertise.

4.1 Terms

Version control system

A version control system (VCS) is a system that is used to detect changes to documents or files. All versions are saved in an archive with time stamp and user ID and can be restored later. Version control systems are typically used in software development to manage source code. Version control is also used for office applications or content management systems. A version control systems consists of multiple versions where each version is committed by a developer.

Subversion

Subversion (SVN) is a specific VCS and a free software for managing versions of files and directories. Versioning is done in a central repository in form of a simple version history. The version history consists of a simple incremental version count where each version contains changed files and lines. All files, changed in a version, are called changed files and all lines, changed in a changed file, are called changed lines. Other specific VCS are Git[1], Mercurial[2] or CVS[3]. We use Subversion to apply our research on, because it is widely used.

Version

A version describes the state of a set of files in a VCS after a commit. With a commit we mean the upload of file changes to a repository. In our research we assume that every new version is created by a specific developer.

Changed file

All files which are commited to create a new version are called changed files. Every changed file can be associated to a developer which committed this file.

[1] `http://git-scm.com/`
[2] `http://mercurial.selenic.com/`
[3] `http://www.cvshome.org/`

Changed line

Changed lines are deleted or added lines in a changed file. Every changed line can be associated to a specific developer.

API

We refer to APIs in a way which is described in [5]. Based on this definition the term API is refering to an interface and also to the underlying implementation. We also make no distinction between libraries and frameworks. An API is according to [5] a set of types (classes, interfaces, etc.) referable by name and distributed together for use in software projects. Also APIs can be described by their package names, package prefixes, and types within packages. An example would be the package prefix javax.swing which can be associated to GUI programming.

Domain

We use the term (programming) domain for extra dimensions of abstraction of APIs. We associate with domain the programming domain like the GUI or IO domain. One domain can consist of multiple APIs (e.g. java.awt and javax.swing associate to the GUI domain). This abstraction is further described in [5] where APIs are associated to a certain programming domain.

4.2 Metrics

We define several metrics to compare the experience of the developers. We consider the information provided by a software repository with version history. Our main objective is to compare the API-expertise of the developer.
The first metrics give an overview of the version history. Thus we are able to estimate how much the developers were involved in the project. After that we define metrics to get informations about the API usage of the developers.

CT *Commit Time* - The first view of the data is the time of the commit of every developer. With this metric we can see how long the developer work on the software project. We can also derive the frequency of commits in the period. Thus we are able to compare which developer has actively worked on this project. In addition it has a difference if a developer commits n-times before 10 years or n-times in the last year. The knowledge or experience of the second one is fresh in his mind.

#VER, #CF, #CL *Number of versions (=commits), number of changed files, number of changed lines* - We look at the number of general changes. Thus we can see how much a developer worked on the project. A developer with one commit, 30 changed files and 90 changed lines can have the same experience like a developer with 15 commits, 2 changed files (in every commit) and 6 changed lines (in every commit). Therefore these metrics depends on each other.

#NJCF *Number of non-java changed files* - The task of some developers can be in the management of the image files or in creating *.xml files*. Our objective in this thesis is to determine the API skills of the developer. Therefore, it don't matter on which non-java files (e.g. .gif, .jpg, .xml, etc.) a developer works on. But the #NJCF gives us an indication to determine if the developer is involved in the java or non-java part of the development.

#QVER, #QCF, #QCL *Number of qualified versions, number of qualified changed files, number of qualified changed lines* - All version, which satisfy the following criteria, are qualified.

- The developer have to be not *null*. Some commits are made by tools like cvs2svn. In this case the developer is not clear and the value of the developer name is *null*.
- The version has at least one changed *.java* file. Commits with only non-java files don't matter in the analysis of API usage.
- The version is not a simple reorganization (refactoring) of the code. If a developer creates branches or copy and paste many files we can expect that no API experience is involved. In contrast to the other criteria we have more work to identify the affected versions. With a SQL-query (see Chapter 5) we can list the versions with the commit message and the number of changed files per version. An example is demonstrated also in Chapter 5

The changed files and lines are qualified if the version of them is qualified. These metrics show how many versions, changed files and lines exist with potential API usage.

#UAE *Usage of API elements* - How many API elements (classifier, method names and enum constants) are used per developer. This metric gives an overview of the API. It is the size of a set. The set consists of API elements which are used in changed lines by developers.

#AQCL *Number of API usage in qualified changed lines* - This metric shows how much specific API are used by a developer. So we can compare the API usage of the developer with each other. It helps to identify the developer with the most experience in a specific API. We take only cleary identified API usages. Some API elements can appear in different APIs. For instance, the method name *get* can appear in the APIs java.awt and *javax.swing*. If a changed file imports this two APIs and the method name appears in a token of a changed line, then we cannot associate the token to the right API. (See also Chapter 8)

#DQCL *Number of programming domain usage in qualified changed lines* - This metric is like the last one. The difference is the abstract level. We look at the programming domains which are groups of APIs. This make it possible to determine the experience in the scope of domains.

5 Methodology

This chapter describes the approach on how we applied our research to a concrete project. In Section 5.1 we describe our used software project on which we applied our analysis. In Section 5.2 we define a data model. In Section 5.3 we describe how we extract the data and use the data model.

5.1 Software Project

Without loss of generality, our approach is implemented for Java as the source-code language and subversion as the version control system. We use the subversion repository of *JHotDraw*[1] project, because it is a favorite software project in MSR research. In [5] a buildable version (without version history) is used to analyze the API usage. For our proposal *JHotDraw* is also a suitable project. We use the version history with 800 versions in 15 years (2000-2015). There are more than 17K changed files and more than 650K changed lines. We identify 45 APIs and grouped them in 18 programming domains. We associate changes with 11 developers.

We define some criteria to select a suitable software project in a software repository. The criteria are listed here:

- More than one developer should work and commit on the repository
- Developer should use APIs.
- Some commits with changes on source code.

5.2 Data Model

We want to obtain API related developer expertise. For that we have to collect data from several sources. Therefore we define a data model. Our starting point is the subversion repository. With every commit we get a new version to the repository. The new version contains specific information like the developer, revision number and commit message. We are able to find out the differences between two versions with a so-called *svn diff* command with a subversion client. We are interested in the differences between two consecutive versions. With this command we can identify the changed files and changed lines which are committed by a developer. The second point are the used APIs. We identify the imported packages in

[1] https://svn.code.sf.net/p/jhotdraw/svn/

the *.java* changed files. With these information we can search for the APIs (e.g. *.jar* files) which contains these packages. In section 5.3 we describe it in detail. To create the data model we need to know that an API contains packages with classifier, method names and enum constants.

The objective is to analyze the changed lines for API usage. So we combine the information of an repository and APIs in one data model (Figure 5.1). The data model shows how APIs consists of API packages, how these packages declare certain API elements (e.g., classifier and methods), how APIs are associated with domains, how repositories consists of files, which files have changed, which specific lines have changed, and how these changes are associated with APIs and elements thereof on the grounds of analyzing the changed lines.

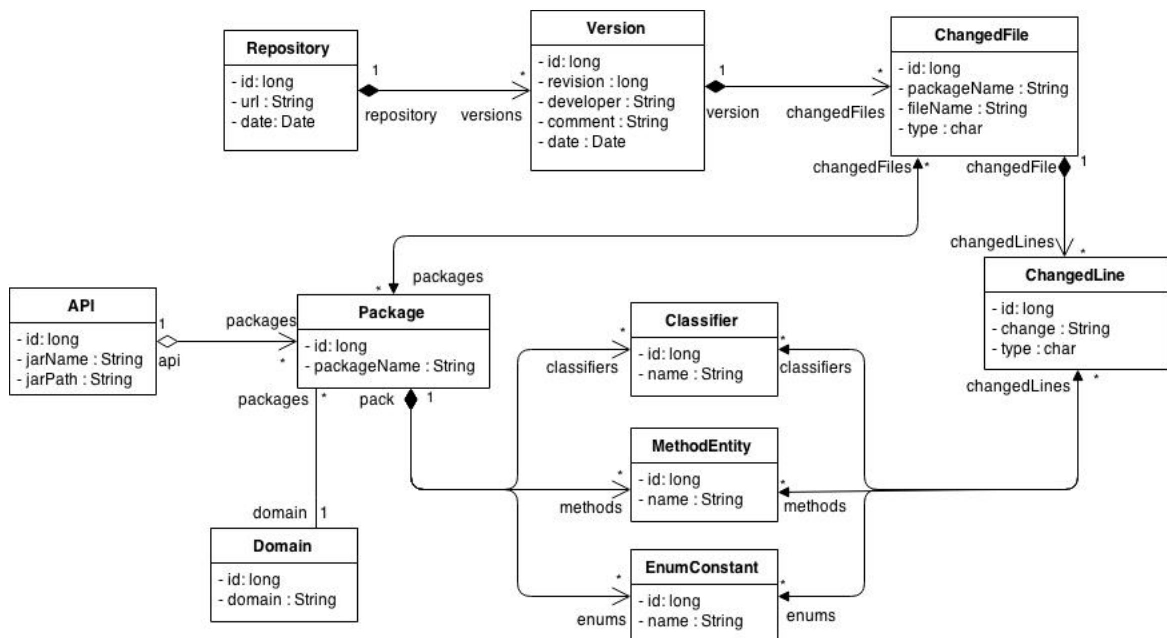

Figure 5.1: Data Model

We can divide the class model into three parts. The classes *Repository*, *Version*, *ChangedFile*, *Package* and *ChangedLine* are for the information we get from the Repository. The classes *API*, *Package*, *Classifier*, *MethodEntity* and *EnumConstant* are for the information we get from APIs. The third part are the classes *Domain* and *Package*. The class *Package* is the intersection of the three parts. It represents the imported packages of the changed file in the repository and also the packages in an API (*.jar file*). In the following subsections we explain the classes and the relations between them.

Repository

The **Repository class** contains the attributes *url* and *date*. The *url* is the address to the repository. The *date* is the start time of the first analysis. Between *Repository* and *Version* we have a One-To-Many relation. One repository can have many versions.

The **Version class** contains the attributes *revision*, *developer*, *comment* and *date*. The *revision* is the number of version, which is committed by a *developer* with a describing *comment* and the *date*. One Version can have many changed files. Here we have a One-Two-Many relation between *Version* and *ChanedFile*, too.

The **ChangedFile class** contains the attributes *packageName*, *fileName* and *type*. The *packageName* is the path to the file without the filename (e.g. src/folder1/folder2/). The *fileName* contains only the name of the file. The files can only be *.java* files. The *type* has the scope "D" for deleted, "A" for added, "R" for renamed and "M" for modified.

The **ChangedLine class** contains the attributes *type* and *change*. The ChangedLine class represents one line which is added or deleted. This is also the scope of the Attribute *type* ("D" for deleted and "A" for added). The attribute *change* is the whole changed line as a *String*. Every changed line can be clearly assessed to a changed file. Every changed file can have more than one changed line. Therefore we have a One-To-Many relation between the classes *ChangedFile* and *ChangedLine*.

The **Package class** contains the attribute *packageName* which represents an import line in the *.java file*. With the Many-To-Many relation between *ChangedFile* and *Package* we know every imported package in a *ChangedFile* and also we know every *ChangedFile* from one *package*.

API

The **API class** contains the attributes *jarName* and *jarPath*. The *jarName* is the name of the *.jar file*. The *jarPath* is the local address to the *.jar file*. Between the *API* and *Package* we have a One-To-Many relation. Every package have exactly one API, but one API can have more than one package.

The **Package class** represents the packages in the APIs (*.jar files*). These are the same package names which are used to import it to the *.java file* Almost we use the shortest prefixes of the package names which clearly defines the API. The organization of the Packages is described in detail in step 2 of Chapter 5.3.

The **Classifier class**, the **MethodEntity class** and the **EnumConstant class** represent three different API elements. All elements contain the attribute *name*. *Classifier* represents e.g. classes, enum types and interfaces. *MethodEntity* contains the names of all methods. *EnumConstant* contains the constants of the enum type.

In this data model is the important relation between the API elements and the changed lines. This relations demonstrate the API usage in the changed lines.

Domain

The **Domain class** contains the attribute *domain*. The attribute *domain* is the name of the programming domain. We have a One-To-Many relation between *Domain* and *Package*. We decide to create this relation and not a relation between *Domain* and *API* because one API can have many packages with different domains. For example the packages *java.util.jar* and *java.util.logging* are in the same *.jar file* but they are assigned to different programming domains.

5.3 Fact Extraction

We have described our data model in Section 5.2. It is implemented as a relational database (relying on *Java DB*) Now we have to extract the data from the various sources and map them to our database. Therefore we define an extractor model which is able to extract the information from our Subversion repository and APIs.

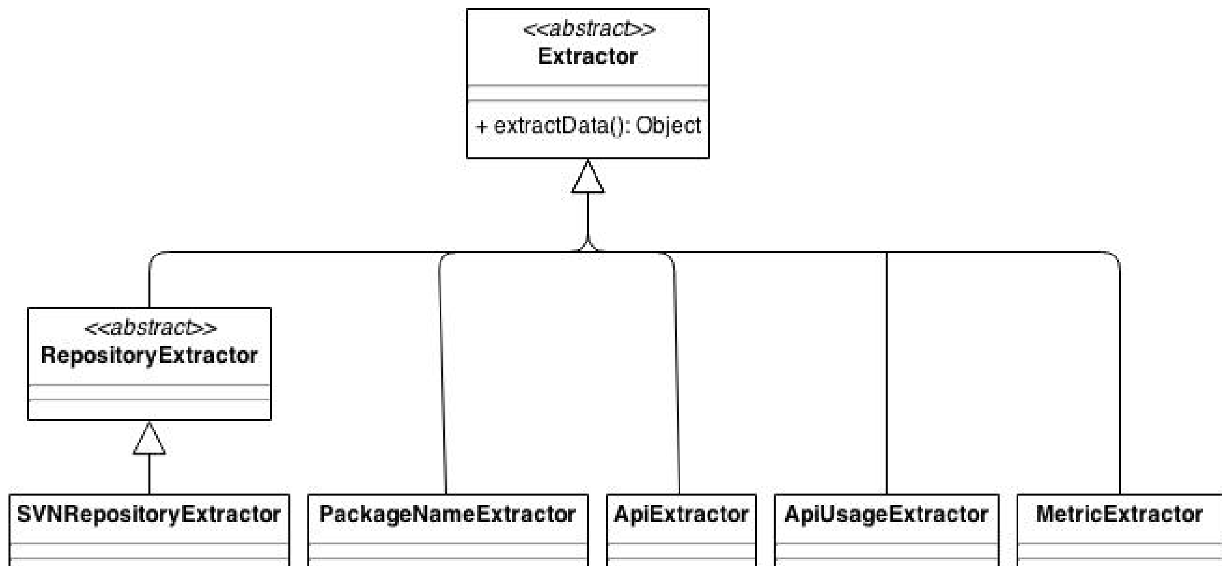

Figure 5.2: Extractor Model

In Figure 5.2 we can see a simple class diagram with only the class names. We have *RepositoryExtractor*, *PackageNameExtractor*, *ApiExtractor* and *ApiUsageExtractor* to obtain all data that is needed in the data model in Section 5.2. We define a *MetricExtractor* to get the metric results based on the data
The extractors have dependencies on each other. The output of an extractor is the input of another extractor. An exception are the domains. The *Fact Extraction Process* in Figure 5.3 demonstrates the work flow in eight steps. The following sections describe the eight steps and the *Extractors* in detail.

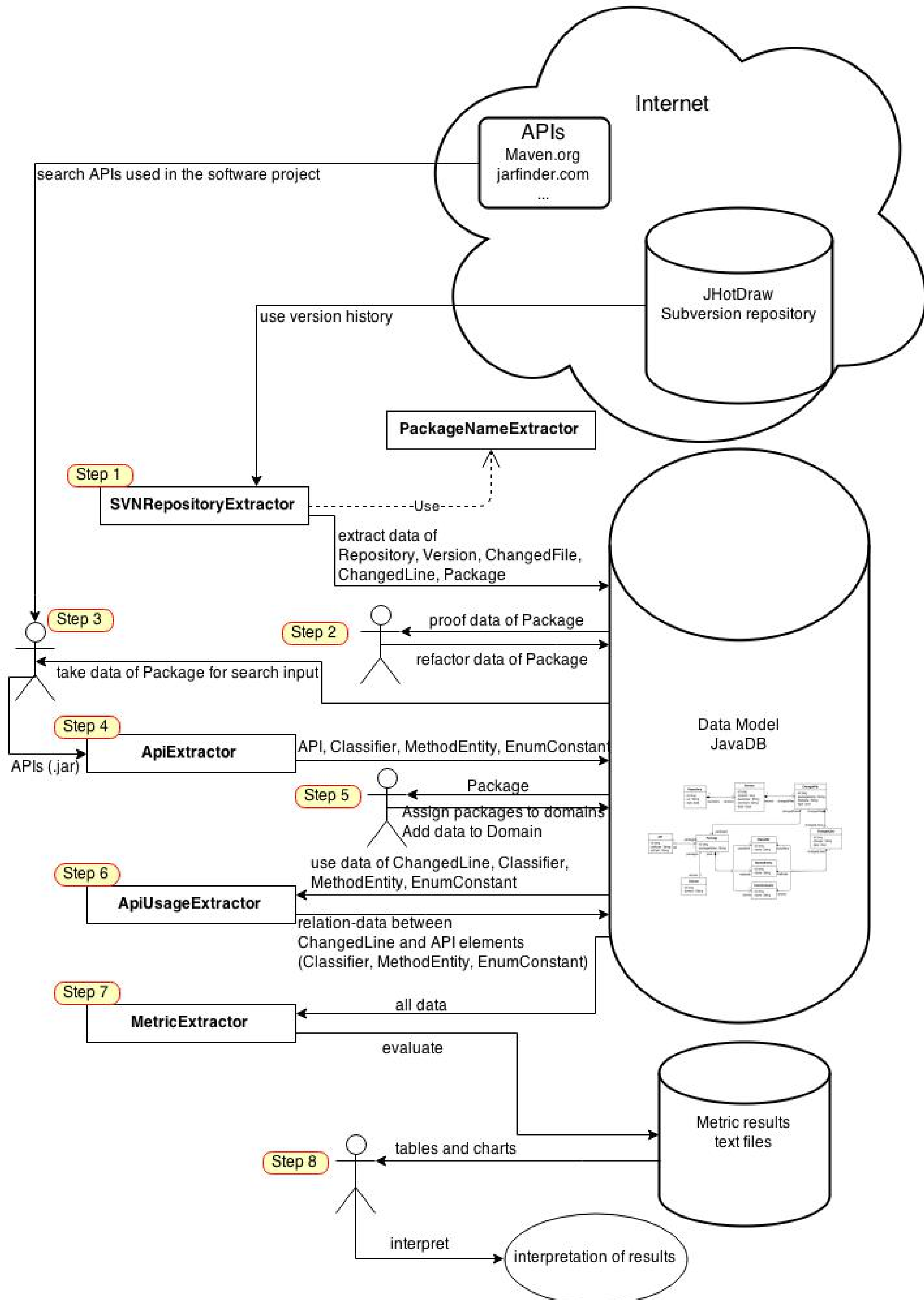

Figure 5.3: Fact Extraction Process

Step 1

The *RepositoryExtractor* extracts the data of a repository with version history like Subversion or Gitorious. In this thesis, we decide to use Subversion and implemented the *SVN-RepositoryExtractor*. This *Extractor* covers the data of *Repository*, *Version*, *ChangedFile* and *ChangedLine* from the data model in section 5.2. To realize the implementation we use the SVNKit[2]. It is a subversion client API for Java (see also Chapter 6). We iterate over all versions and get the changed .java files. With a *svndiff* of every changed file we get the changed lines.

While iterating over changed files we analyze also the imports of every file with the **PackageNameExtractor** and get the data for *Package*. The PackageNameExtractor takes only packages to external APIs. Associations to internal packages are ignored. Some of the internal packages can be in the database. This can happen if the packages have been renamed, but older imports into the changed files still exist. This and similar problems with the data of *Package* is described and solved in step 2.

Step 2

In this step, we proof the data of *Package*. The objective in this step is to have short and clear package names, which are also contained in the APIs. These are the criteria to remove or reorganize the data:

1. **Static packages** Some imports contains the keyword *static* in the beginning of the package name. This keyword can be eliminated, because the pacakge names of an API does not contain the keyword *static* in the beginning.

2. **Internal packages** Package names, which are from the package structure of the project, can be removed, e.g. package names beginning with *org.jhotdraw*, because all of them are not references to external APIs.

3. **Subfolder elimination** If there exist package names with different subfolders, e.g. *javax.swing.border* and *javax.swing.plaf*, then it can be reduced to the clear part of the package name, e.g. *javax.swing*. Mostly The clear part is the first two prefixes. The packages found in the changed files must also be adjusted, e.g. all changed files which imports the packages *javax.swing.border* or *javax.swing.plaf* must be linked to *javax.swing*. The links to *javax.swing.border* and *javax.swing.plaf* can be deleted after the relink process.

In the *JHotDraw* project we identified 234 packages. We applied the criteria and eliminated most of them.

1. **Static packages** We find 9 entries (see Figure 5.4) and remove the keyword *static*.

2. **Internal packages** With the following query we eliminate 118 internal packages:

[2] `http://svnkit.com/`

```
select * from PACKAGE where PACKAGENAME like 'static%';
```

ID	PACKAGENAME	API_ID	DOMAIN_I
174	static java.lang.Math	<NULL>	<NULL>
163	static org.jhotdraw.color.HarmonicColorModel	<NULL>	<NULL>
73	static org.jhotdraw.draw.AttributeKeys	<NULL>	<NULL>
172	static org.jhotdraw.draw.HandleAttributeKeys	<NULL>	<NULL>
186	static org.jhotdraw.draw.handle.HandleAttributeKeys	<NULL>	<NULL>
145	static org.jhotdraw.samples.odg.ODGAttributeKeys	<NULL>	<NULL>
141	static org.jhotdraw.samples.odg.ODGConstants	<NULL>	<NULL>
110	static org.jhotdraw.samples.svg.SVGAttributeKeys	<NULL>	<NULL>
109	static org.jhotdraw.samples.svg.SVGConstants	<NULL>	<NULL>

Figure 5.4: Static packages

```
delete from PACKAGE_CHANGEDFILE where PACKAGES_ID in
(select ID from PACKAGE where PACKAGENAME like 'org.jhotdraw%'
or PACKAGENAME like 'CH.ifa%');
delete from PACKAGE where PACKAGENAME like 'org.jhotdraw%'
or PACKAGENAME like 'CH.ifa%';
```

3. **Subfolder elimination** For illustration we eliminate the 11 subfolder of *java.awt*. First we relink the 1455 associations from the subfolders to the *java.awt* package (*PACKAGE_ID*=2).

```
insert into PACKAGE_CHANGEDFILE (PACKAGES_ID,CHANGEDFILES_ID)
select distinct 2 as PACKAGES_ID,CHANGEDFILES_ID
from PACKAGE_CHANGEDFILE where PACKAGES_ID in
(select ID from PACKAGE where PACKAGENAME like 'java.awt.%')
and CHANGEDFILES_ID not in (select CHANGEDFILES_ID
from PACKAGE_CHANGEDFILE where PACKAGES_ID=2);
```

Then we can remove the links from the packages with subfolder.

```
delete from PACKAGE_CHANGEDFILE where PACKAGES_ID in
(select ID from PACKAGE where PACKAGENAME like 'java.awt.%');
```

At least we can eliminate 11 packages with subfolder.

```
delete from PACKAGE where PACKAGENAME like 'java.awt.%';
```

At the end we eliminated 58 additional entries.

As result we eliminate 187 and keep 47 packages.

Step 3

In this step we search the used APIs. With the package names we have to find the APIs (.jar files). We can expect that for the most APIs exist more than one version. In this case we

take the newest version of the API. That means that we measure the API-related expertise for the newest version of the APIs. The following possibilities are available to find the right API:

- Some APIs are integrated in the *Java SE Runtime Environment (JRE)* or in the *Java SE Development KIT (JDK)*. Figure 5.5 shows an overview of the integrated APIs in Java.

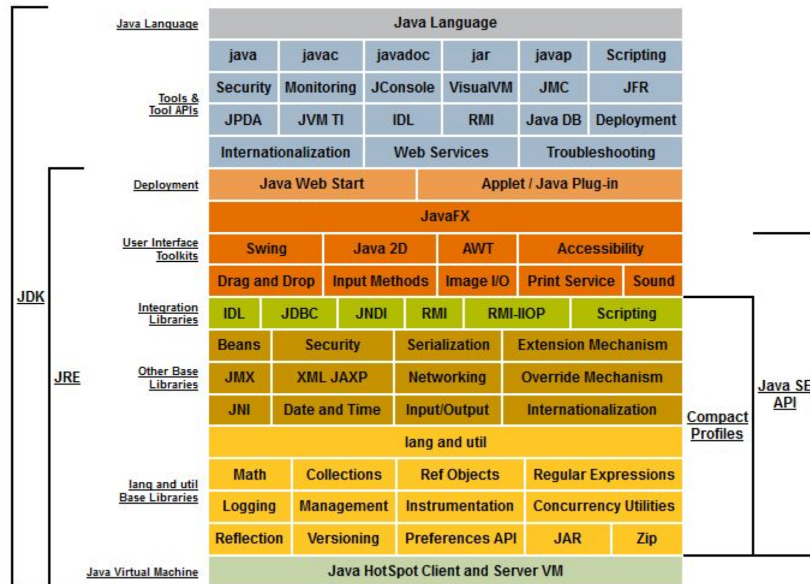

Figure 5.5: Java Conceptual Diagram [13]

- Some APIs can be found on the homepages of the APIs, e.g., *junit API*[3] or *quaqua API*[4].

- The other APIs can be found from the maven repository[5] or on sites like findjar.com[6]

Step 4

The *ApiExtractor* extracts the data of an API file like .jar file. It covers the information of *API*, *Package*, *Classifier*, *MethodEntity* and *EnumConstant*.
The extractor iterates over all packages in the *.jar file*. We pick up all classifier, e.g., class names or interfaces, and their method names or enum constants to store it in our database. For instance, we extract the API elements of some *java.* packages in this way:

[3] `http://junit.org/`
[4] `http://www.randelshofer.ch/quaqua/`
[5] `http://search.maven.org/`
[6] `http://findjar.com`

```
File f = new File("apis/rt.jar");
ApiExtractor extractor =
        new ApiExtractor(new URL("file:"+f.getAbsolutePath()));
extractor.extractData("java.util",false);
extractor.extractData("java.awt",true);
extractor.extractData("java.io",true);
extractor.extractData("java.applet",true);
extractor.extractData("java.util.zip",true);
extractor.extractData("java.util.logging",true);
```

The boolean value after the package name defines, if the subfolders should also be considered. We found in total 17 APIs (*.jar files*) with 10979 classifiers, 28556 method names, and 1730 enum constants.

Step 5

We grouped the 47 packages in 18 programming domains. The domains have to be added manually to the database.

```
insert into "DOMAIN"("DOMAIN") values ('Achieving'),('Basics'),
('Component'),('Concurrency'),('Configuration'),('Distribution'),
('Format'),('GUI'),('IO'),('Logging'),('Math'),('Meta'),('Output'),
('Parsing'),('Persistence'),('Testing'),('Web'),('XML');
```

With update-querries we can assess the packages to domains. It looks like this:

```
update PACKAGE set DOMAIN_ID="IDofDomain" where ID="IDofPackage";
```

Step 6

The *APIUsageExtractor* extracts the data of the relation between the *Classifier* and *ChangedLine*, *MethodEntity* and *ChangedLine* and *EnumConstant* and *ChangedLine*. As described in Chapter 3 we use a lexical approach [1][2] to analyze the changed lines for API usage. More specifically we tokenize changed lines in lexical units. We iterate over all lexical units and check if an API element occurs there. To restrict the search we use only the API elements of the imported packages.

Step 7

The *MetricExtractor* takes all data of our database and generates the metric values. The metrics are defined in Chapter 4. The values are stored as .csv files and can be opened with Microsoft Excel, for example.
To determine which versions are qualified we need to check the criteria. The first criterion is that the developer is not null. The second criterion is that in the version is at least one

changed file. If we join both table with the *ID* of *Version* then we get only the versions with changed lines. We can sum up these criteria with this query.

```
select distinct v.ID,v.COMMENT,v."DATE",v.DEVELOPER,v.REVISION,
v.REPOSITORY_ID from VERSION v, CHANGEDFILE cf
where DEVELOPER is not null and v.ID = cf.VERSION_ID;
```

We get 627 versions. We can eliminate some versions with the third criteria. In Table 5.1 we demonstrate some unqualified versions. The commit message describes the changes. Therefore, we can eliminate the most of them by identifying the reorganization from the commit message. If no message are available or the message is not clear, then we look at the number of changed files. The versions with many changed files (and not clear description in the commit message) are unqualified.

Revision	commit message	#CF
2	Initial revision	284
19	Merge to JHotDraw 5.2 (using JFC/Swing GUI components)	304
32	before merge for version 5.3 (dnd, undo,...)	86
33	before merge for version 5.3 (dnd, undo,...) - 2	192
34	before merge for version 5.3 (dnd, undo,...) - 2	32
35	before merge for version 5.3 (dnd, undo,...) - 3	174
36	merge dnd (before 5.3)	220
36	merge dnd (before 5.3)	16

Table 5.1: samples for unqualified versions

We identify 638 qualified versions.

Step 8

The results and the interpretation of them are illustrated in Chapter 7.

6 Infrastructure

Java

The programming language Java as defined in [14] is a concurrent, object-oriented, class-based language with a focus on having as few as possible implementation dependencies. Java was designed with a focus on interoperability where a program can run on multiple platforms. To achieve this interoperability Java Code compiles into bytecode which then can be executed by the Java virtual machine (JVM) regardless of computer architecture. It was developed by James Gosling at Sun Microsystems and was released in 1995.
Without loss of generality, the contributed application which was used to apply our research is written in the Java programming language. In our research we use a software project which is written in the Java programming language to demonstrate our intent.

Netbeans

As described in [15] NetBeans is an IDE which is used to build rich client applications. It was originally developed by a student in the Czech republic. In 1999 the company Sun Microsystems bought it and released it as an open source project. It was mainly developed for the Java programming language but it can also be used for C, C++, PHP, Ruby and Python. NetBeans relies on plugins to extend and customize its functionality.

SVNKit

SVNKit[1] is a Java library to interact with a subversion repository. It provides an API to access and manipulate a subversion repository. The library itself is written in Java and is open source. Therefore OS specific dependencies are not required to use this library. It was used to retrieve information from the *JHotDraw* project which is under version control by subversion.

Java Database Connectivity (JDBC)

JDBC as described in [16] enables us to establish a database connection from a project using the Java programming language. There are a lot of different databases, SQL databases or spreadsheets and therefore JDBC is database independent. We use JavaDB as database. JDBC is integrated in the NetBeans IDE and is used to map our data model (Figure 5.1) to a JavaDB database and extract our results.

[1] `http://svnkit.com/`

Apache Derby

Apache Derby[2] is an open source relational database written in the Java programming language and is distributed under the Apache License, Version 2.0. Derby is based on the Java, JDBC, and SQL standards and easy to install, deploy and use. Therefore it was used as our database.

[2] http://db.apache.org/derby/

7 Discussion of Results

In this chapter we interpret our results and compare the developer skills with each other. Our results are the metric values. Before we begin with our results, we give every developer an ID for better representation of our charts and tables. In text we use *"developer name" ("developer ID")*

Developer	**ID**
birdscurrybeer	1
cfm1	2
dnoyeb	3
gesworthy	4
jeckel	5
mrfloppy	6
mtnygard	7
pleumann	8
pmorch	9
rawcoder	10
ricardo_padilha	11
null	12

Table 7.1: Developer -> ID

In Chart 7.1 we demonstrate the values of the metric CT. On the x-axis we have the commit times and on the y-axis we have the developers. At first glance, we can see two periods. The first period ranges from end of the year 1999 to beginning of the year 2007. The second period ranges from end of the year 2006 to now (2015). In every period we have a main developer. The main developer is in the first one *mrfloppy (6)* and in the second one *rawcoder (10)*. It can be supposed that both of the main developer have a leading position in this software project. Therefore we can expect a wide range API usage of both developer. For detail information we need to see the other metrics. The two years 2003-2004 stand out in contrast to the other years. In this short period eight developers work together on this JHotdraw project. The rest of time it looks like a one man project with two little exceptions in year 2008 and 2010.

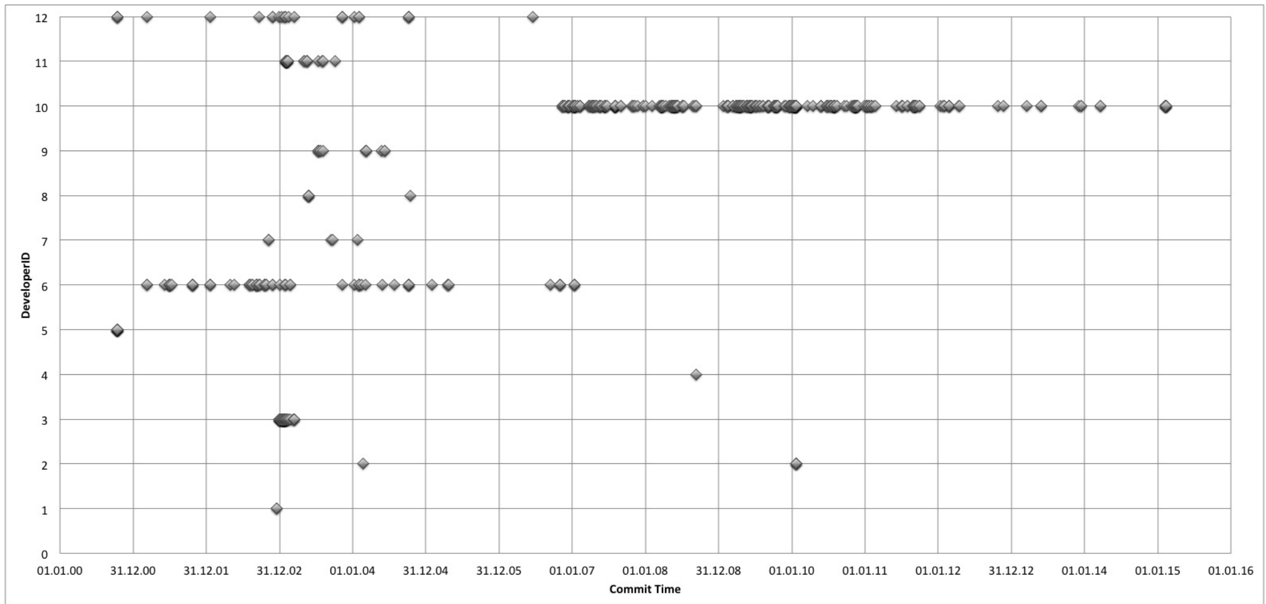

Figure 7.1: Metric results of CT *"Commit Time"* for every developer

In Table 7.2 we demonstrate the metric results for #Ver, #CF, #CL and #NJCF of every developer. The developer *birdscurrybeer (1)* has changed 36 lines of code and *gesworthy (4)* has changed 19 lines of code. They also commit one and two times. Those should be insufficient information to make a statement about their developer skills. As expected the most changes is made by the two main developer with almost 300K changed lines. The developer *dnoyeb (3)* has more commits than *mrfloppy (6)*. If we have a look back to the CT's, we recognize that *dnoyeb (3)* commits in few months 90 times. The developer *mrfloppy (6)* needed six years (two of them are without commits), but he had 13x more changed lines. The two main developers also work on non-java files. The developers *jeckel (5)* and *cfm1 (2)* works less than the previous stated ones. The other developer only work on *.java files*.

DeveloperID	#VER	#CF	#CL	#NJCF
1	2	4	36	0
2	6	88	9495	52
3	90	916	20942	0
4	1	6	19	0
5	14	308	19856	239
6	83	4507	279520	1624
7	5	452	2719	0
8	7	19	280	0
9	13	26	638	4
10	517	10070	298191	2939
11	35	194	2482	0
12	27	392	46066	0

Table 7.2: Metric results of #VER *number of versions*, #CF *number of changed files*, #CL *number of changed lines* and #NJCF *Number of non-java changed files* for every developer

After elimination of the unqualified versions and their changed files and lines, we get the Table 7.3. We recognize that for some developer the circumstances changes extremly. At the beginning, we thought that *mrfloppy (6)* can be a main developer, but we also see that he have not a big knowledge in several APIs. We identify 5907 qualified changed lines (#QCL) and 279520 changed lines (#CL) which means that he has 273613 unqualified changed files. Thus unqualified changed files come only from the reorganization process of the project. The values of the other developers with IDs *(1),(2),(3),(4),(5),(7),(8),(9), and (11)* have no relevant changes.

DeveloperID	#QVER	#QCF	#QCL
1	2	4	36
2	2	88	9495
3	90	916	20942
4	1	6	19
5	12	24	1782
6	41	336	5907
7	3	12	397
8	7	19	280
9	11	26	638
10	434	5992	198976
11	35	194	2482

Table 7.3: Metric results of #QVER *umber of qualified versions*, #QCF *number of qualified changed files*, #QCL *number of qualified changed lines* for every developer

Before we look at the specific API usage of every developer, we show the results of the metric #DQCL in Table 7.4. It demonstrates the API usage grouped in programming domains. The API elements from the *GUI*, *IO* and *Meta* domains are used very often from the most of the developers. The other API elements are mostly used by the developer *rawcoder (10)*. This result is realistic, because this application is a drawing program and therefore it contains a lot of *GUI* and *IO* elements. We also expected that the developer *rawcoder (10)* has a wide API usage. He consistently developed over eight years alone. With this clear results we can compare the developers only in the three stated domains. So we pick as example the domain *GUI* in table 7.6 and compare the developer with each other. Before we do that, we make a overview of the usage of API elements. For illustration we also take the APIs of the *GUI* domain in Table 7.5.

APIs	1	2	3	4	5	6	7	8	9	10	11
Achieving	0	0	0	0	0	0	0	0	0	79	0
Basics	0	0	0	0	0	0	0	0	0	3704	0
Component	8	0	20	0	0	6	0	0	0	8440	6
Concurrency	0	0	0	0	0	0	0	0	0	317	0
Configuration	0	0	0	0	0	0	0	0	0	1686	0
Distribution	0	0	8	0	0	22	0	0	40	3304	0
Format	0	0	0	0	0	0	0	0	0	2106	14
GUI	4	64	4580	3	568	828	82	26	169	65796	650
IO	0	20	752	0	14	189	61	2	10	10523	90
Logging	0	0	0	0	0	0	0	0	0	348	0
Math	0	0	0	0	0	0	0	0	0	7	0
Meta	0	50	1754	2	246	520	2	2	53	19450	204
Output	0	0	0	0	0	0	0	0	0	46	0
Parsing	0	0	0	0	0	0	0	0	0	88	0
Persistence	0	0	0	0	0	0	0	0	0	37	0
Testing	0	0	0	0	0	0	0	0	0	92	0
Web	0	0	0	0	0	0	0	0	0	217	0
XML	0	0	0	0	0	0	16	0	0	1415	0

Table 7.4: Metric results of #DQCL *Number of programming domain usage in qualified changed lines* for every developer

Table 7.5 shows the numbers of usage of API elements. The most differently used API elements of the *GUI* domain are the APIs *java.awt*, *javax.swing* and *javafx*.
In Table 7.6 we can see that the *java.awt* and *javax.swing* are the mostly used APIs of the *GUI* domain. With the metrics #UAE and #AQCL we can infer that the APIs *java.awt* and *javax.swing* have a central role in this project, but *java.awt* is the favored one. Here, we can see again that the developer *rawcoder (10)* has the most expertise in those APIs. The second one is the developer dnoyeb (3) with a difference of 50K, and the third one is *mrfloppy (6)* with a difference of 4K. So we are able to rank them all and to identify the developer with the most API-expertise in this comparison.

APIs	Classifier	Methods	Enum Constants
ch.randelshofer.quaqua	50 (597)	159 (1757)	50 (597)
com.sun.javafx	11 (1227)	75 (3143)	11 (1227)
java.awt	175 (501)	589 (1723)	175 (501)
javafx	93 (2391)	161 (5526)	93 (2391)
javax.swing	335 (1697)	906 (3372)	335 (1697)
org.apache.batik.ext.awt	14 (237)	79 (740)	14 (237)
org.apache.batik.svggen	0 (92)	2 (268)	0 (92)
org.jdesktop.application	22 (81)	71 (439)	22 (81)
org.jdesktop.swingworker	2 (5)	6 (18)	2 (5)
sun.awt.geom	1 (20)	5 (103)	1 (20)
sun.swing	1 (116)	52 (753)	1 (116)

Table 7.5: Metric results of #UAE *usage of API elements* for the API elements of *GUI* domain

APIs	1	2	3	4	5	6	7	8	9	10	11
ch.randelshofer.quaqua	0	0	0	0	0	0	0	0	0	2806	0
com.sun.javafx	0	0	0	0	0	0	0	0	0	199	0
java.awt	6	69	5204	1	568	867	24	96	211	58139	598
javafx	0	0	0	0	0	0	0	0	0	1443	0
javax.swing	10	5	1620	4	0	475	64	82	102	54687	328
org.apache.batik.ext.awt	0	0	0	0	0	0	0	0	0	706	0
org.apache.batik.svggen	0	0	0	0	0	0	6	0	0	0	0
org.jdesktop.application	0	0	0	0	0	0	0	0	0	388	0
org.jdesktop.swingworker	0	0	0	0	0	0	0	0	0	64	0
sun.awt.geom	0	0	0	0	0	0	0	0	0	21	0
sun.swing	0	0	0	0	0	0	0	0	0	80	0

Table 7.6: Metric results of #AQCL *number of API usage in qualified changed lines* of every developer in APIs of *GUI* domain

8 Threats To Validity

Internal Validity

Internal validity verifies that the outcome is really caused by the treatment. We identify some threats to internal validity.

- During the data analysis, we did observed some inconsistent results. The inconsistent results appeared during the lexical analysis described in the Section 5.3. We tokenized a changed line and checked for used methods of APIs. In addition, we restrict the search using only methods from the imported packages. Those results which we receive from such an analysis are influenced by the fact that two different imported packages can contain an API element with the same name. In this case we are not able to clearly determine the correct API of the token. Therefore our analysis consider only the clearly associated tokens to APIs. We will illustrate the precision of our data in our study. The APIUsageExtractor found 52982 classifiers, 61309 method names, and 1033 enum constants used in the tokens of changed lines of qualified versions. 52521 of 52982 (99,1%) classifiers, 23988 of 61309 (39,1%) method names, and 1018 of 1033 (98,5%) enum constants are clearly associated to tokens of changed lines.

- Our found APIs from maven repository or other sites are always the newest versions. There may be problems with deprecated methods if the analyzed software project used older versions of APIs. In this thesis we look at the API related expertise based on the newest Version of the chosen API.

- We search the APIs from maven repository or other search engines like findjar.com. We use package namespaces or prefixes such as javax.swing to search the APIs. Our search results shows us multiple APIs with the same package namespace or prefix. Therefore it is possible to choose the wrong API for our analysis.

- We have defined the programming domains on the base of [5]. It is possible to associate a package to another domain. Therefore it is a threat to evaluate domain expertise of developer. We tried to define the domains clearly, but we can always refer back to APIs rather than domains.

- We chose a repository where are more than one developer and all developer have their own user accounts to commit to the repository. We don't know if one account can be used by more than one developer. In software projects with one account and multiple developers we cannot evaluate the API skills.

We believe the threats to internal validity are well controlled in our study. We also described the methodology and algorithms of our study in details, in order to facilitate future replications by other researchers.

External Validity

External validity verifies that the results of a study are generalizable.

- we only analyzed one repository. Although popular and large, this repository may not be representative of the general population of repositories. We assume the normal usage of the repository. Other software repositories can be used in a different way. But it should be acceptable, because we ignore the files which we are not needed in our analysis.
- We consider only the experience of the developer in one project. To make a total profile, we need all software projects of a developer.
- We cannot say if a developer has API skills or not. The absence of a measure scale make it impossible. How often should we use an API to get API expertise? 10,100,1000 times? In this thesis we only compare the developer with each other.
- For executives and project managers the analysis of the API skills could not be useful. It could be, that the new project needs API skills which are not contained in the analyzed software project.

9 Conclusion

We developed a new application which evaluates the version history of a software project and the used APIs to determine API-related developer expertise. In order to accomplish our objectives we answer our research questions.

RQ1.1: How can we measure API-related developer expertise?
We defined several metrics to determine API-related developer expertise. They consider several parts. First part are the metrics CT, #VER, #CF #CL, #NJCF,#QVER, #QCF and #QCL. Then we consider the general project information. Questions like, who is the mostly active developer? Who worked on the source code and who worked on other files? are answered by these metrics. The rest of the metrics #UAE, #AQCL and #DQCL consider the API usage. The last ones allows us to compare the API skills of the developer with each other. Taken as a whole, we can measure the API-related developer expertise.
In this thesis we apply our analysis on *JHotDraw*. With these metrics we identify some types of developer. For illustration we list two samples:

- The developer *rawcoder* has the most API-related developer expertise in all used APIs. He works for over eight years alone on this project.
- The developer *mrfloppy* has developed a long time, but in contrast to *rawcoder* he developed in the same time as other developers. He has skills in *Component*, *Distribution*, *GUI*, *IO* and *Meta* domains. His best API skills are in the *GUI* domain. In the *Component* and *Distribution* domains he has few skills.

As described we are able to make profiles of the developers with those comparisons.

RQ1.2: What is the important information from the version history to get the API-related developer expertise?
We look at the structure of a version history. In this thesis we decide to use *Subversion*. With these information we define a data model (Figure 5.1). We identify *Version* information like developer name, revision number, and commit message, the *ChangedFile* information, and the *ChangedLine* information. In this manner, every changed line can be associated with a specific developer. In addition we identify the information of APIs and build with all information the data model to determine the API usage in the changed lines.

RQ1.3: How can we create a System to obtain semi-automatically API-related developer expertise?
We create a system where we can obtain semi-automatically API-related developer expertise. Our extractor process Figure 5.3 illustrates our system. We can see all steps of our analysis

to determine the API-related developer expertise.

RQ2: How can we create a connection between changed lines of code and the API without building the underlying project?
We decide to use a lexical approach. We tokenize every changed line and search for API elements. In related work Chapter 3 we discuss this approach and alternatives. We use this lexical analysis in our application.

Executives and project managers have the possibility to evaluate a developer in an objective way. They need only previous software projects of a developer. The projects must be in a VCS.

In the future we can extend this approach in different ways. One opportunity is to combine a lexical analysis with a syntactical analysis to identify the API usage in the changed lines. Another opportunity is to apply the analysis on other software projects with other languages, like C#, Python or Visual Basic. Not only the language can be changed. Also the VCS can be changed to Gitorious or Mercurial. One aspect are the several versions of APIs. This thesis uses the newest versions, but it is possible to create methods to determine the used versions of the APIs. An interesting approach would also be to define an measuring scale for the metrics. Then it would be possible to evaluate independently every developer.

List of Figures

5.1 Data Model 12
5.2 Extractor Model 15
5.3 Fact Extraction Process 16
5.4 Static packages 18
5.5 Java Conceptual Diagram [13] 19

7.1 Metric results of CT *"Commit Time"* for every developer 25

Bibliography

[1] Tripathy, P. ; Naik, K.: *Software Evolution and Maintenance*. Wiley, 2014 https://books.google.de/books?id=7XrDBAAAQBAJ. – ISBN 9781118960295

[2] Klusener, A. S. ; Lämmel, Ralf ; Verhoef, Chris: Architectural modifications to deployed software. In: *Sci. Comput. Program.* 54 (2005), Nr. 2-3, 143–211. http://dx.doi.org/10.1016/j.scico.2004.03.012. – DOI 10.1016/j.scico.2004.03.012

[3] Xie, Tao ; Pei, Jian: MAPO: mining API usages from open source repositories. In: *MSR*, IEEE, 2006, S. 54–57

[4] Lammel, Ralf ; Linke, Rufus ; Pek, Ekaterina ; Varanovich, Andrei: A Framework Profile of .NET. In: *WCRE*, IEEE, 2011, S. 141–150

[5] Roover, Coen D. ; Lämmel, Ralf ; Pek, Ekaterina: Multi-dimensional exploration of API usage. In: *ICPC*, IEEE, 2013, S. 152–161

[6] Wang, Jue ; Dang, Yingnong ; Zhang, Hongyu ; Chen, Kai ; Xie, Tao ; Zhang, Dongmei: Mining succinct and high-coverage API usage patterns from source code. In: *MSR*, IEEE, 2013, S. 319–328

[7] Robbes, Romain: Mining a Change-Based Software Repository. In: *MSR*, IEEE, 2007, S. 15

[8] Canfora, Gerardo ; Cerulo, Luigi ; Penta, Massimiliano D.: Identifying Changed Source Code Lines from Version Repositories. In: *MSR*, IEEE, 2007, S. 14

[9] Yu, Liguo ; Ramaswamy, Srini: Mining CVS Repositories to Understand Open-Source Project Developer Roles. In: *MSR*, IEEE, 2007, S. 8

[10] Linstead, Erik ; Rigor, Paul ; Bajracharya, Sushil K. ; Lopes, Cristina V. ; Baldi, Pierre: Mining Eclipse Developer Contributions via Author-Topic Models. In: *MSR*, IEEE, 2007, S. 30

[11] Wu, Wei ; Adams, Bram ; Guéhéneuc, Yann-Gaël ; Antoniol, Giuliano: ACUA: API Change and Usage Auditor. In: *Proc. of SCAM 2014*, IEEE, 2014, S. 89–94

[12] Wu, Xiaomin ; Murray, Adam ; Storey, Margaret-Anne D. ; Lintern, Robert: A Reverse Engineering Approach to Support Software Maintenance: Version Control Knowledge Extraction. In: *Proc. of WCRE 2004*, 2004, S. 90–99

[13] *Java Platform Standard Edition 8 Documentation - last visit: 03/2015.* `http://docs.oracle.com/javase/8/docs/index.html`

[14] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. – ISBN 0321246780

[15] Boudreau, Tim ; Tulach, Jaroslav ; Wielenga, Geertjan: *Rich Client Programming: Plugging into the Netbeans™ Platform*. First. Upper Saddle River, NJ, USA : Prentice Hall Press, 2007. – ISBN 9780132354806

[16] Reese, George: *Database programming with JDBC and Java*. " O'Reilly Media, Inc.", 2000