Developer Profiling - extract developer expertise in a Django app

Frederik Rüther
Matriculation Number: 212200063

December 18, 2015

Abstract

The identification of experts for a specific technology or framework produces a large benefit for collaborative software projects. Hence it reduces the communication overhead that is required to identify an expert on the fly. Therefore this thesis describes a tool and approach that can be used to identify an expert that has a specific skill-set. It will mainly focus on the skills and expertise of developers that use the Django framework. By adding more rules to our framework that approach could easily be extended for different technologies or frameworks. The paper will close with a case study on an open source project.

Zusammenfassung

Die automatische Identifikation von Experten in einer speziellen technologischen Domäne, wie einer Bibliothek, Framework oder generellen Technologie, schafft einen großen Mehrwert in der gemeinsamen Entwicklung von Softwareprojekten. Daher soll in dieser Arbeit ein Vorgehen sowie ein Programm zur automatischen Identifikation von Experten entwickelt werden, die gewissen Skills besitzen. Hierbei wird speziell das Django-Framework betrachtet. Jedoch kann durch hinzufügen von weiteren Regeln unser Tool leicht auf andere Technologien angepasst werden. Abschließend wird eine case study auf ein Open Source Projekt durchgeführt.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäss übernommen wurden, sind als solche gekennzeichnet.

Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. ja ☐ nein ☐

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. ja ☐ nein ☐

Koblenz, den Dezember 17, 2015

Contents

Acknowledgement

I would like to thank my supervisor Prof. Dr. Ralf Lämmel for his support during the progress of creating this thesis. Further I want to thank the whole "devProv" team for the support and advice during the weekly meetings.

1 Introduction

Communication between developers is a key success factor for software projects. In fact developers spend about 16 percent of their working time with talking and consulting with other experts or developers [1]. The first step, that has to take place before discussing the issue, is to find a person you can consult with and has the required skill-set. That task can be very easy in a small team, which is located close by or even in the same building. If teams from several countries support the project, the team is large or there is a lot of fluctuation the challenge becomes much harder. Especially in Open Source projects it can be a problem to find the right expert while facing a development problem.
This work provides a tool that can help to overcome that problem by analyzing the provided data of the Version Control System Git to assign technical skills to a developer. A skill will be assigned, if a certain rule-based condition is fulfilled. For the creation of that rules the focus lays on the Django framework[1]. That framework is coded in Python and its aim is to "make it easier to build better Web apps more quickly and with less code". Our tool provides a Domain-specific language, which is a subset of Python, to define rules in an easy but yet safe way. These rules will dynamically be executed and the result will be saved. The rule creation process can use the *Content-, Filename-, ParseTree- and Directory-Filter* that can be composed over logical operations like and, or and not.
Therefore, this work tries to answer the question, if it is possible to automatically extract the skills of the developers in a large Django project, which is hosted on GitHub. Often developers help themselves with the use of the blame feature of GitHub to find out who to ask. This, however, is just a limited help because it does not provide a history and it is just limited to one file. Furthermore the starting point is another. If you are using the blame feature, it is already known what file has to be changed or what file is a good example. But to get there is not in particular easy if you are working on a project with a few hundred or thousand files. It would be by far easier to search for specific skills like migration and you would directly see who has done the most in that context and what files could serve as an example. That is a non trivial task because you have to answer two questions first. 1) When does a developer has knowledge about migration or any other specific skill? 2) How do I identify the developers that have such a skill? The first question was answered by defining rules that identify key features or operations that tell that a developer has certain skills. By finding developers that have performed an operation that has one of those key features the second question was answered. Both questions have to be solved to provide a solution for the main research question.
The result will finally be a tool, that can take as input any Django project on GitHub and provides a list of skills that were used in this project and the contributors that have experience with a certain skill. That tool will be coded in Python and it will provide a SQLite database file, that holds all informations of the analysis. Further mining techniques can be applied to that dataset like searching for skills that two developers usually have in common. An additional output will be a set of rules that map different operations to a skill.
First of all, the related work to that topic will be discussed. After the ground-

[1] https://www.djangoproject.com/

work has been discussed a chapter will describe the methodology and finally the approach followed to find and at the end set up the rules that identify skills in the Django framework. Furthermore an explanation will be given on how we discovered interesting projects on GitHub and why the Django-Oscar project was chosen. The next chapter gives a description of the domain-specific language for the rules. After that a short overview of the high level design of the tool will be given. If the concepts and a rough understanding about the acting together of the different components were set, the next chapter will go into details of the implementation. Finally the case study will be explained in detail, and the result will be presented. A short discussion about limitations and future work will close this thesis.

2 Related work

Software repositories have been used to study various aspects of the software development process. A few of those studies have a focus on the developers and the skills or experiences they have gained through working on the code base saved in the repository. To access those data Version Control Systems (VCS) are commonly used.

Furthermore a few papers worked on classifying software artifacts by the used technology. For example De Roover [2] created a tool that mined object oriented design patterns in Source-Code.
Software vulnerabilities are a great thread in the development of software systems. Therefore, Shin et al. [3] provided a tool that used different metrics to measure code chain and complexity, to classify potential vulnerable files.
The social aspect of software development was studied as well. MacLean and Knutson [4] used the commit history of the Apache Project for 2010 and 2011 to create a large graph with the different developers as nodes. The edges connected those who have committed to the same file. Moreover further metrics were collected.
Not just the communication aspect of the developers were analyzed but the overall activity of developers on GitHub as well. This had the goal to study the characteristics of the contributors. Therefore, Onoue et al. [5] used the target project and the other projects of the contributors to get an overall activity picture of a developer.
Mockus and Herbsleb [1] calculated the amount of changed lines in a program text, or delta as they call it, to calculate the amount of experience atoms (EA's) a developer has collected. An experience atom is an "elementary unit of" experience. In the end an Expertise Browser (ExB) was coded to visualize the relationships between code, documentation and other artifacts and the people who gained experience in those.
Rysselberghe and Demeyer [6] used the data of the VCS to visualize which files were changed on a specific date. They finally used that information to visualize the commit history of the Tomcat repository over a specific period of time to interpret the plotted graphs and gain insight into the software evolution.
Alonso et al. [7] approach differs from the previous since they visualized the commits from contributors into specific folders in the Apache HTTP WebServer project. The amount of commits were used to calculate the experience of specific users which was later visualized, illustrating through the size of the font how often a skill was used.
Not all of the provided works just focus on the VCS. Bhattacharya et al. [8] added in their approach the bug tracking system to get a deeper understanding of the different roles a person has in an open source project. Finally a graph was build with the collected metrices and mining techniques were applied on it.
Teyton et al. [9] provided a tool and a Domain Specific Language (DSL) to define skills by specific modifications to a file. To elaborate, it provided a content, path, modification and a tree filter. Those filters could be composed in the DSL, based on XML, to create rules. The program, however, is limited to Java-Applications.

3 Methodology

Rule creation: One of the first steps was to find valuable rules that identify skills of a developer. A research of literature to that topic did not provide a useful result for our task because the roles of the developers were either at a level that was to high like Tester, Software Architect, Developer (and so on) or were too specific on a particular project. The roles or skills should be applicable for every project in the Django domain and further should focus on the skills of the developers. Therefore another way to get rules had to be found.
Two approaches were chosen. First the Django tutorial[2] and documentation[3] were analyzed. For example the documentation was used for the migration process to identify operations that implicit that skill. This was later used to combine different filters (see Chapter 5.2). Furthermore we had a closer look at different Django projects on the Github repository to check if the created rules would be satisfying and of any worth in a real world project. By looking into the source code of those projects a more detailed insight into Django and its features was gained. That was used as input for further examinations of the Django Documentation. The complete process is shown in Figure 1. Finally, the

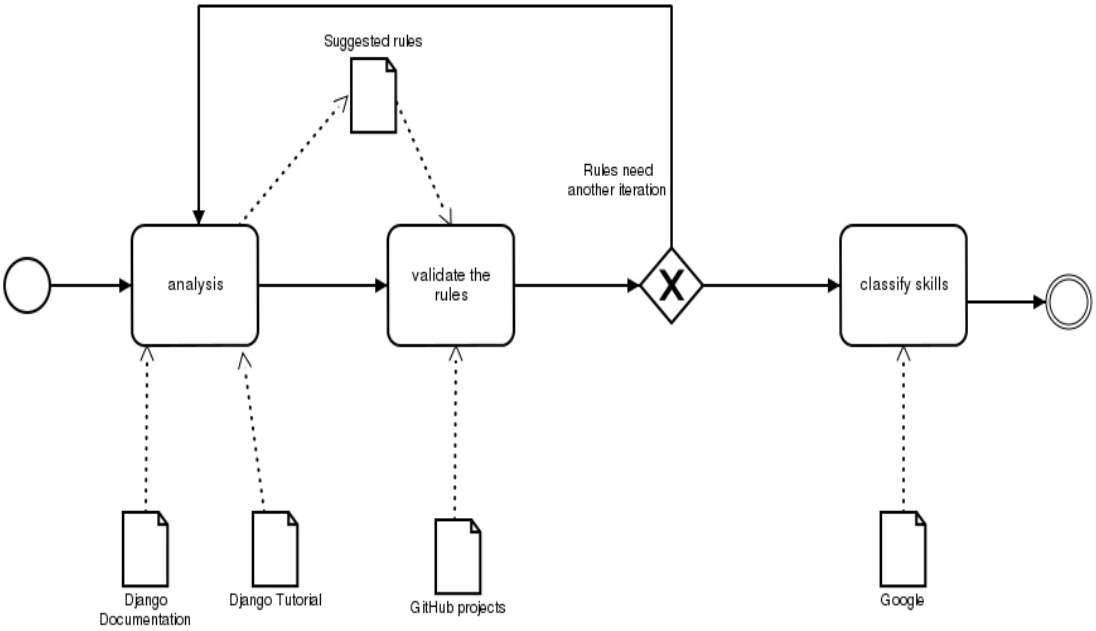

Figure 1: Process of the rule creation

found skills were hierarchical organized to give a better overview and connect and mark logical similar skills. To do so the Django documentation as well as other sources, that were available through the web, were used. Chapter 7.2 describes more in detail how the skills were defined for the case study.

[2]https://docs.djangoproject.com/en/1.8/intro/
[3]https://docs.djangoproject.com/en/1.8/

Project selection: A project for the case study had to be selected too. The list of "Awesome Django[4]" on Github was used to identify interesting projects. This list is based on the "Awesome Python" [5] document which provides a list of useful or popular libraries, frameworks and general software coded in Python. By using that list it was pretty safe to say that the projects would be successful and had a development time of a few years. This document was found by searching on GitHub for the keyword Django. From that (long-)list a shortlist containing five Django projects was selected. The criteria to be on the shortlist were:

1. more than five contributors,
2. more than hundred commits,
3. typical for a Django based project,
4. time of development longer than 3 years and
5. a large impact of Django in the project.

Afterwards the projects were prioritized by the best distribution of the commits from the developers and the general usability for an analysis based on the project structure. Finally, the Django Oscar project was selected. It has enough commits and contributors to provide a good dataset and about five contributors that could potentially be compared. The Figure 2 visualizes the state of the repository. Although the two main developers drove more than 80 percent of the project, the distribution of the touched files was still acceptable. In fact, that behavior seems to be common in small open source projects. In the Case-Study section more details about the project will be provided.

Figure 2: Diagram for the distribution of touched files per user - with and without the two dominant contributors

[4]https://github.com/rosarior/awesome-django
[5]http://awesome-python.com/

4 Domain-specific language

Layout: The analysis of the repositories is based on rules. Those rules can be described as an injective function that takes a *LogicalOperation* with arguments and returns a skill (Figure 3). The logical operations themselves are

$$
\begin{aligned}
rule\colon (LogicalOperation, arguments) &\rightarrow \mathcal{S}kill \\
Skill &\in (ID \times EA) \\
arguments &\in (mode \times path \times content \times dif) \\
ID, path, content, dif &\in String \\
EA &\in \mathbb{N} \\
mode &\in \{A, M, D\}
\end{aligned}
$$

Figure 3: Definition of the rules as mathematical function

composed through the combination of filters and other logical operations. A possible EBNF notation for the definitions of those rules is defined in Figure 4. There are also other possibilities to implement the rules, but for that demonstration a definition similar to chained function calls was used. An example of a valid rule following that grammar could be:
And(LeftOr(FilenameFilter('x', 'y'), ParseTree('a', 'a')), ContentFilter('f', 'g'))

⟨*LogicalOperation*⟩	::=	⟨*operator*⟩'('⟨*expr*⟩', '⟨*expr*⟩')'
⟨*expr*⟩	::=	⟨*filter*⟩
	|	⟨*LogicalOperation*⟩
⟨*filter*⟩	::=	'FilenameFilter('⟨*args*⟩')' | 'ContentFilter('⟨*args*⟩')' | 'ParseTreeFilter('⟨*args*⟩')'| 'DirectoryFilter('⟨*args*⟩')' | 'None'
⟨*operator*⟩	::=	'And' | 'OR' | 'LeftAnd' | 'RightAnd' | 'LeftOr' | 'RightOr' | 'Not'
⟨*args*⟩	::=	**String** (',' ⟨*args*⟩)*

Figure 4: Definition of an example grammar for the rules

As the grammar shows, there exist four types of standard filters, that can be further customized over the arguments (args): ContentFilter, ParseTreeFilter, FilenameFilter and DirectoryFilter. Those filters were selected as default filters because of two reasons. On the one hand, the related work of Teyton et al. [9] and Alonso et al. [7] gave reasons and arguments to use those and on the other hand while creating the rules for Django those filters were necessary.
The FilenameFilter analyzes the filename, as the name says, to check if a con-

dition is fulfilled. Besides that, we can also check if the commit took place into a specific directory. That one is handled by the DirectoryFilter. Furthermore, we are able to analyze the content or the diff of a specific file. That is where the ParseTree and ContentFilter come into play. They differ in the way they analyze the content. The ParseTree filter takes the semantic structure of the code into account whereas the *Content-Filter* just does a plain search for keyword or a regular expression.
How much experience or knowledge a developer has in a certain skill is measured by the amount of experience atoms he has collected. This paper will use the definition by Mockus and Herbsleb [1]: "Experience atoms are elementary units of experience. Experience, we assume, is the direct result of a persons activity with respect to a work product, enhancing it or fixing a problem. The smallest meaningful unit of such changes is an EA."
Logical operations will finally merge the experience atoms collected by the filters using different strategies.
Semantic of filter merging: The result of the filters will be returned to the logical operations such as And, Or, Not, LeftAnd, LeftOr, RightAnd or RightOr which take those and decide what to return. A preferred return value can be set by using the left or right version of an operation. Those are designed to either return the result of the filter on the right or left side if both options are given. That option is especially useful if one of the filters gives a deeper inside into a skill or is more concrete. An example for the use of an operation like the *RightAnd* or *LeftAnd* could be, if one filter is just used as a condition. Consider you are searching for added lines to a class that inherits from *Model*. To be able to determine if it is the the model class you are searching for, or if it is the part of another package you want to check the imports. To do so you can use the *ContentFilter* to check if the phrase "import XYZ" is part of the file. The *ContentFilter*, however, is just there to make sure that it is really the searched class. Therefore, the return of that filter does not really matter as long as it returns something else than None. In that scenario you can specify that the ParseTree filters should return its result if both apply by using a Left or RightAnd.
A similar case can be constructed for the use of a *LeftOr*. Consider you are combining two *ContentFilter*. One that searches for an old version and one that searches for a new version of a string. That could apply, for example, due to a refactoring that led to the rename of a variable. If there happens to be the new version, you perhaps want to return the new version, otherwise you want to return the old one. That scenario can be implemented due to the use of the left or right version of the *Or* where the new string will be setup as the dominant side.
However, there are scenarios, especially if they are combined over an *Or* operation, that would cause a bias if they define a dominant side. Therefore, the normal rules will return the average value of both filters. That is the best available solution because it is a compromise between the maximum and minimum. The minimum and maximum would both be imprecise in an *Or* operation, if just one filter would have a result. Figure 5 tries to demonstrate that. F1 is a filter that tends to return a larger value than the filter F2.
Thus it is evident that, the minimum performs more worse than the average, if the dominant side suddenly becomes zero. The maximum does perform almost as good as the average. The disadvantage of this is, that small changes to the

weaker filter have no impact to the output. The average would make those observable because it weights both sides in the same way and thus both changes will have an affect. Therefore, the average is better and fairer and it was used in the calculation.

Operation	Example
Minimum	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 0 = 200
	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 20 = 20
Maximum	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 20 = 200
	F1 $\rightarrow$ 0 Or F2 $\rightarrow$ 20 = 20
	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 100 = 200
	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 198 = 200
Average	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 0 = 100
	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 20 = 110
	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 20 = 110
	F1 $\rightarrow$ 0 Or F2 $\rightarrow$ 20 = 10
	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 100 = 150
	F1 $\rightarrow$ 200 Or F2 $\rightarrow$ 198 = 298

Figure 5: Comparison of different operations for the combination of results

5 Design

The system is designed by composing three different parts which are rules, modules and libraries. It further provides a main program that uses the three parts and implements the main algorithm or business logic. In the following three sections the layout of those will be explained in detail. During the design phase following philosophies were used. First of all, the focus was to develop simple and small code (KISS principle[6]). The functions, modules and libraries should be small (maximal twenty lines per function) and hence be easy to comprehend. If there happened to be a requirement that did not fit logically into an existing module or library a new one was created. We adhered to the principle of Separation of Concerns [10]. In the structure of the project all of those parts and even the elements in the parts are located in different folders and subfolders. The aim is to have a folder for all the related files and further have subfolders to distinguish between those. As a result, all related files can be found in the same directory and hence the locality principle was followed.

5.1 Modules

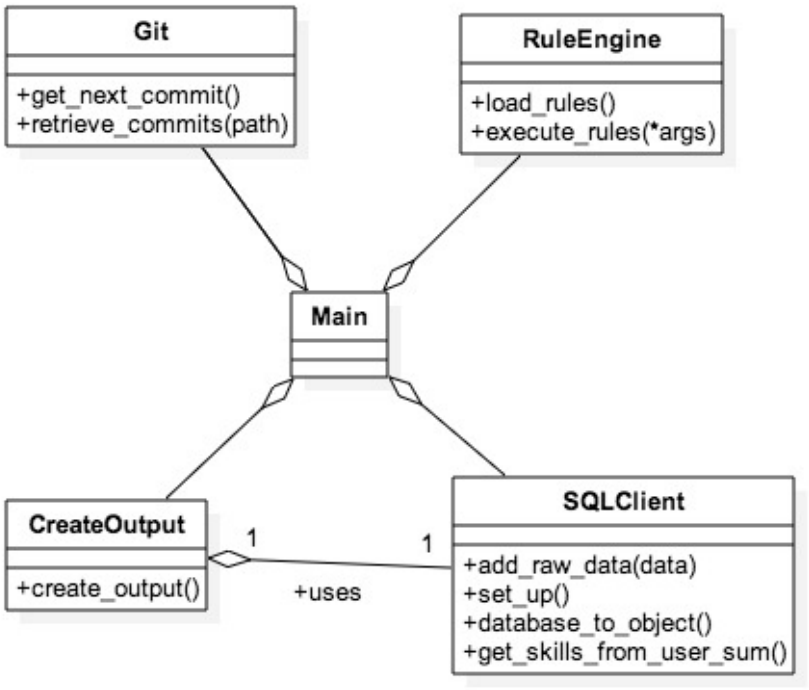

Figure 6: Diagram for the layout and dependencies of the modules

A module is a Python source code file, that encapsulates a specific function and provides an interface to call it. The modules will mainly be called from the main program so that there are just a few dependencies between those. In fact

[6] http://people.apache.org/ fhanik/kiss.html

one module has a dependency to another module. The relation between those are shown in Figure 6. However, the aim of the design is to make the modules as loosely coupled as possible. That way "the probability that [...] a system can be learned, ported, modified and extended more easily" increases [11].
The project contains four modules that extend the functionality of the program. The Git module has an interface to access the commits sorted by time. It further provides a lot of details about the commit like the contributor and the diff. Another module is the SQLClient module that provides functionality to access a SQLite3[7] database. Especially it is able to persist specific objects into the database and map them back into object form. The tool uses a SQLite database since the output will just be a single file that can be easily shared to other people and further no separate server has to run on the client or at a remote host. To be able to execute rules a RuleEngine module dynamically loads all the existing rules, makes sure to execute them and returns the collected result. The rules are Python files that follow a specific schema. Last but not least, a specific module will create the output in the form of HTML files. To be able to retrieve all the different data, it imports the SQLClient module. That module has specific functions to support that module. To be able to retrieve all the different combinations of data it imports the SQLClient module that has an interface with functions that that module has to call. Removing that dependency would force the main module to run a lot of specific SQL queries to hand them over to the CreateOutput module. A cleaner design is to import the SQLClient module directly.

5.2 Rules

Rules are an important part of the system. Hence they define the skills a contributor has. Therefore, there is a specific location, specified in the configuration file, that contains all the rules. The rules themselves are composed through the combination of filters and logical operations. Both expressions are realised over the definition of a class in Python. This has the advantage compared to XML that they are easier to read, maintain and further functionality could be added like the transformation of input.
The implementation is similar to the composite pattern [11], with the difference that it allows compositions of different types. In fact, for both types (Filter, LogicalOperation) an abstract superclass exists from which all the concrete definitions have to inherit. The class *LogicalOperation* can contain instances of itself or zero till two instances of the Filter class. Or in other words: It can contain two elements which are either of type Filter or LogicalOperation or both. The apply() method will finally walk down the complete constructed chain and merge the results of the filters.
The rule itself just contains one object of the type *LogicalOperation*. All of those logical operations like And, Or , Left/Right-And/Or and Not follow different merging strategies of the result of two filters (Chapter 4 explained those). The design aims to give the creator of the rules as less restrictions as possible.
Rules are represented through a specific class that inherits from the BaseRule class. Therefore, the tool knows that it is safe to call the match method of the super class. The match method will simply call the apply function of the

[7] https://www.sqlite.org/

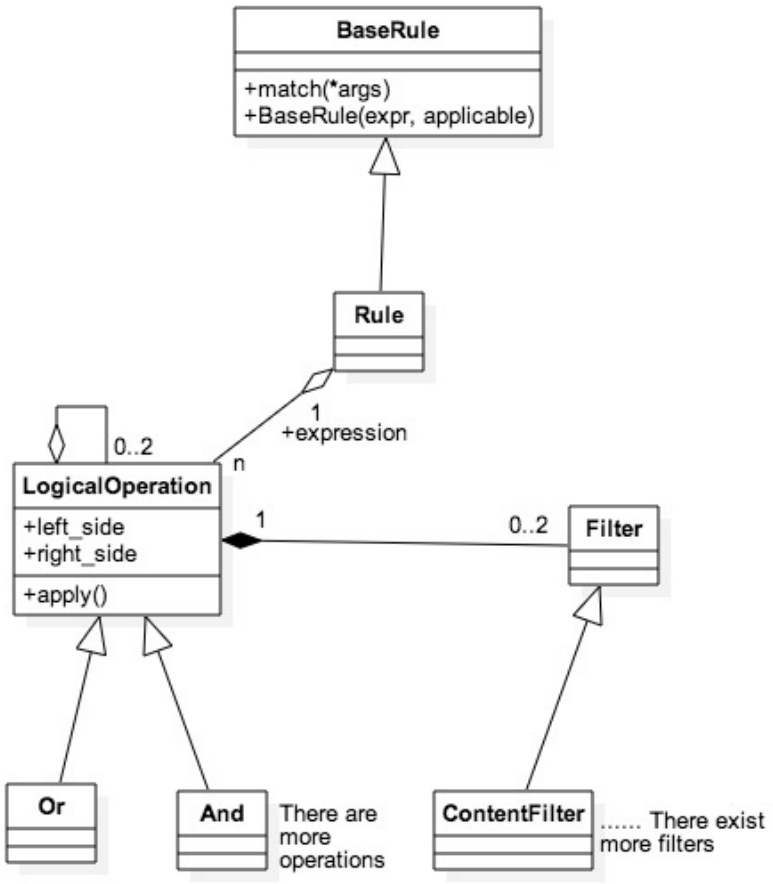

Figure 7: Class diagram for the layout of the rules

defined logical operation, which was part of the construction of the rule. The child class can overwrite the match and perform some transformation of the input if necessary. The filters that are default implemented so far are: *Directory-, Filename-, Content- and ParseTree-Filter*. The design of the complete chapter is visualized through the UML class diagram in Figure 8. In that diagram the *Rule* class is a placeholder for an actual implementation of a rule.

5.3 Libraries

Libraries collect functionality that will later be used by different modules or rules. On a high level we can distinguish between two kind of libraries.
1) The libraries that simply define objects that aggregate facts that will be used and saved into the database. There are four of those types, as the Figure 8 shows, the User, GitCommit, File and Skill class. An interesting decision was to create a new File object for every version of a particular file in the version control system. As a result, the amount of created objects increased a lot but a more in depth analysis can be performed and insight gained. For example the

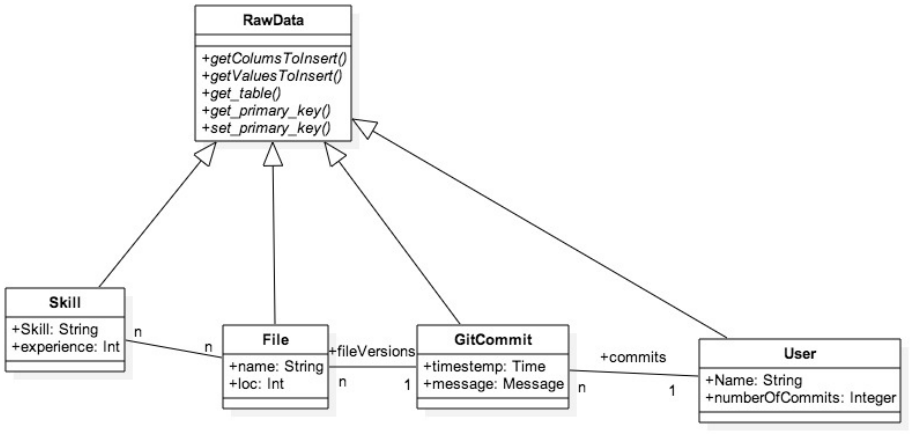

Figure 8: Class diagram for the layout of the data

data could be used to show the evaluation of skills since every version points to a set of skills it contains. Figure 9 gives an example of a dataset where that observation can be made. At first the file model.py just contained one skill. After another commit not just another skill was added to the file but the previous skill was used more heavily as well. Thus we can see how the file and the used skill develop over the time.
With the awareness of that feature the creation of a new file object per commit seemed to be reasonable. After all data that are aware of their historical development over the time were retrieved.

FileVersionId	Name	LOC	CommitId
12	model.py	12	2
13	setup.py	14	2
14	mode.py	20	3

SkillId	Name	Experience	FileVersionID
1	Model/Data	5	12
3	Model/Data	10	14
4	Model/Admin	3	14

Figure 9: Example of FileVersion and Skill table

2) The Libraries mainly providing functionality are the filters, logical operations and finally the libraries used by those two types and other parts of the project. The filters itself implement an object oriented interface to call specific functions inside another library. Those libraries are just a collection of different useful functions. They are separate from the filters to allow other parts of the code to call their functions as well without creating a filter. Therefore, the design allows to easy create and use plain libraries from everywhere in the program but still provides a clean and easy interface for the rule creation that abstracts from the implementation details.

6 Implementation

After having given an overview of the design this chapter aims to provide some interesting details about the implementation. The complete implementation can be found on Bitbucket[8].

6.1 Main

The main program is set up of a few very simple steps. First of all, it reads the configuration file to determine where the rule folder, database file, Git repository and the output directory are located. Those values will be handed over to the right places. After that, it calls the initial process of the different modules and libraries if necessary. Finally the setup phase is finished and the main loop starts. The first step is to ask the Git module for all values of the next commit. Those values will be used to create the data-storing objects which will be saved into a SQL database through the interface of the SQL-Module after the creation. Afterwards it will walk two times over all the files in the commit. The first walk updates the inheritance tree which contains the current state of the class layout and their inheritance structure. The second time the RuleEngine module is called with the mode, filename, content and diff as argument which will execute all the rules with those arguments and compose the result. This process avoids that the order of execution of the rules and files has an affect on the result. Then the main program saves the results into the database through the SQLClient module. Once the loops are finished the results are available. However, they are just available in the database at this point. Therefore, the output module will be called which plots the database into easier to read HTML files.

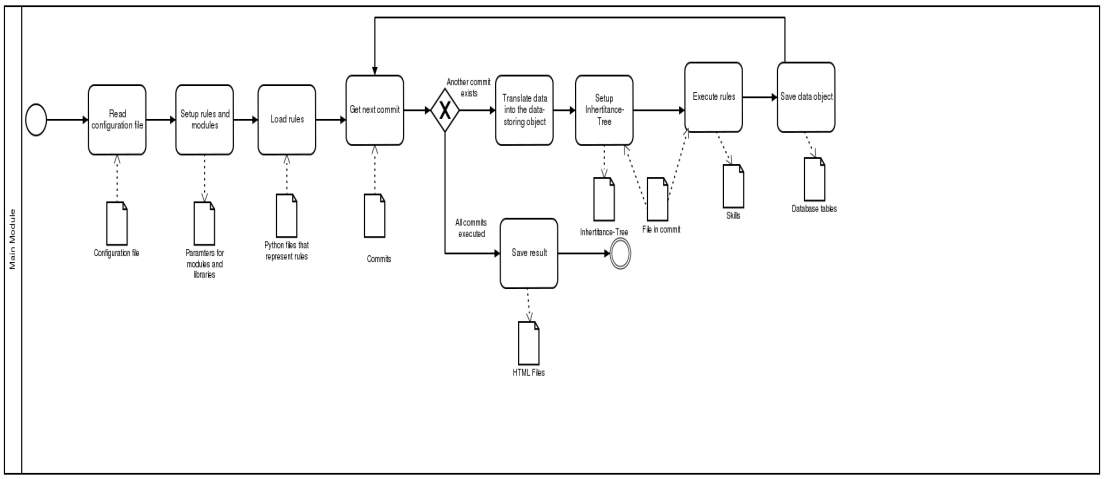

Figure 10: BPMN diagram for the main algorithm

[8] https://bitbucket.org/Freddy92/bsc.-developer-profiling/src

6.2 Rules

Each rule refers to a certain kind of layout. It has to inherit from the BaseRule class, which has the abstract method get_skill_string. That method returns the name of the skill defined by the rule. Besides that, they have to tell the base class via the constructor what logical expression that rule represents. The base class itself implements the match logic and returns the Skill object, if the match of the rule was successful.

Layout of skills: The skills are clustered in a hierarchical approach and hence each skill is assigned to a root category. This will be represented through a string similar to a path definition in a filesystem. It starts with the category and combines them with the other categories using a slash "/" and finally ends with a particular skill. So basically the form is category/subcategory/skill or with concrete values: Model/Data. Per definition the skills can always just be found at the end of the chain. That means that the category descriptions can't be a skill itself but rather describes the skills that are in its cluster.

The problem with that layout is that it is not normalized. in fact it is not in the 1NF, [12] and therefore it would be hard to update a database, if a change has to be applied in the structure of the skills. The data in the database, however, never claimed to be correct at any particular moment in the future but just represent the state of skills for a specific amount of rules at a specific point in time. Every change in a rule would result in a different output on a new run of the program. Therefore, an easy to change dataset is not really a requirement for the design. The layout provides the benefit that a user can easily understand and read the database. A rule just has to return a string which is convenient. The database itself, as well as the Python standard environment, provides a lot functions to manipulate and search in or for a string. As a result, a request can be easily made that searches for a specific skill or all skills of a category e.g. database.

Experience calculation: The rules themselves use the filters to extract the experience of a user in a specific domain. Those filters can be found in the library folder and will return the experience a user has in a specific context, measured by the amount of experience atoms. So the type of an experience atom is an integer.

There are different options how the amount of atoms can be calculated by the filters. They can be calculated by the number of

1. found strings or regular expressions in a file,
2. found strings or regular expressions in a diff log,
3. lines added in the diff or
4. added lines to a specific area of the source file e.g. to a specific method

A filter can be configured to use a specific calculation method. The configuration will take place over a parameter in the construction. Figure 11 shows the mapping between filters and calculation methods. The logical operators will get these values and compose a final result.

Filters: The filters provide an easy interface to call the libraries (Chapter 5.3) that collect functions that logical fit to it. One of the arguments of the filter construction specifies, through a string, what function of the library the

Filter	Measurement of experience atoms
Parse Tree	4
Content	1, 2
Filename	3
Directory	3

Figure 11: Filters and the way the calculate EAs

filter should execute. Therefore, the libraries can change their implementation and even names without causing any problems in the rules. In fact, just the filter has to adapt to the changes. As a consequence, the project is easier to maintain.
Furthermore the rules become a lot easier to read since the logical operation will be composed through the creation and chaining of different objects. That would not be possible in a not object oriented way. Figure 12 shows the implementation of an example rule.
During the construction other filter specific options could be set as well. For example the *ParseTree-Filter* provides two options that specify its internal behavior.
To summarize, the filters encapsulate the logic that calls the subroutines inside a library. As a result, that code stays outside of the rules and thus the rules stay small and easy to comprehend.
Another information a rule has to set up is on what Git operations it should be invoked. There are three kind of options which are add, modified, renamed or deleted.
The following code (Figure 12) shows an example of a rule that describes the skill *Migration* which is a subtype of the database category. It used the logical operation to combine the *Content-Filter* and the *Parse-Tree-Filter*. Of course you could combine it with more in detail filters as well.

6.3 Libraries

The Inheritance Tree library has the purpose to build up a tree structure that aggregates the inheritance structure of the system. Internally it is basically a Python dictionary, whose key is the name of the class and its value is a list of all the classes it inherits from. If we want to know if a class X is the subtype of another class, it simply walks recursively over the Inheritance Tree starting with the key X and adding all values to the execution queue till it reaches an end or finds the searched base class in it. The Inheritance Tree, however, doesn't parse or is able to read the Python source code. For that the ParseTree library + comes into play. It uses the Python AST module[9], which is part of the Python standard library, to parse Python source code and finally extract the class name and base class names. Those will either be stored in the Inheritance Tree or be given as an argument to it, to find out if it inherits from a specific class. Another use of the parse tree library is to find functions with a specific signature in a provided Python code. A side effect of this implementation is that it does not always correctly tells the end of the class but rather says the class ends at the beginning of the last expression.

[9]https://docs.python.org/2/library/ast.html

```
import sys
sys.path.append("../")


from library.Rules import BaseRule
from library.Rules.Operations.And import And
from library.Rules.Filters.ParseTreeFilter import ParseTreeFilter
from library.Rules.Filters.ContentFilter import ContentFilter

class MigrationRule_(BaseRule.Rule):
    def __init__(self):
        regex = r'((import (django.db.)?migrations)|(import (.)+, migrations))'
        ptf1 = ParseTreeFilter("Migration")
        ctf1 = ContentFilter(regex, "regex_anywhere")
        expr = And(ctf1, ptf1)
        super(MigrationRule_, self).__init__(expr,["A", "M"])

    def get_skill_string(self):
        return "Database/Migration"
```

Figure 12: An example of a rule

That library itself uses our Diff library. The Diff library has the purpose to read the diff format and returns all the added lines to a file. That result will then be used to determine if the person has added something to an interesting function or class.
As we have discussed in the Design chapter the second way to search for content is by using a plain content search without bothering about the structure of the code. To achieve this the Content library has implemented a few functions to search in a string. One is by regular expressions another is by the use of a simple string search. Two different algorithms are available for an efficient search. The Python standard search and an implementation of the Knuth-Moris algorithm[10]. There exist two versions of each function. One that returns a boolean value and another that returns the number of matches.
The Filename library provides some functions to analyze the filename. It provides the ability to search for an regular expression or for the occurrence of a string in a specific part of the name e.g. the start, end or middle of it.
As mentioned earlier, another possibility is to connect the location of a directory to a specific skill. That is why the Directory library provides a function to check if a path is a subpath of another.
All classes that save data that have to be persisted into a relation database like Skill, Commit, File or Contributor objects have to inherit from the RawData class. This class has mandatory methods that describe the layout of the relational table like name, columns and what values to be saved. This is an easy way to provide an interface between mapping data from relational to object form. In the program itself just objects will be used and it will never be asked

[10]http://www.cs.jhu.edu/ misha/ReadingSeminar/Papers/Knuth77.pdf

for just a specific column in the table.
The *Content-Filter* and logical operations will be discussed together with the rules. They indeed are libraries as well but they logically fit in the rule location better.

6.4 Modules

The Git logs are accessed by the GitModule through the GitPython[11] library. The repository will first of all be pulled or if it does not exist cloned. There were two reasons to work on a local repository. On the one hand it is faster because it does not have to use the, compared to a disk, slow network to get all the data and second the GitHub Web API is limited to a to small amount of requests per hour. The module will first obtain all commits and save important informations into a stack. That data structure was chosen to assure that the oldest commits are looked at first. The order of execution is important to assure that the Inheritance Tree is valid.
To deal with added and deleted lines of code it will simply split commits, which modified files, into two parts. First it calculates the normal diff and then it calculates the deleted lines by changing the order in the diff calculation. Thus the ‚+ ‘will appear as - and vice verse. The advantage is that the code now logically just has to deal with adds and hence can apply the same logic of skill assignment for deleted and added content. Another hazard, that occurred during the work with Git, were merge and pull requests or other constructs that could lead to a commit with multiple parents. Once that happens, GitPython returns more changed files than the GitHub webpage revealed. Since Github should always reflect the result of the tool some fixes had to be applied. So the subroutine check_file_in_commit(...) was added. As a downside, even through caching, the speed of the application decreased. The correctness of the application has a greater value, though. Finally the stack will be decreased by returning all files and their content and diff for each commit.
The RuleEngine[12] simply scans the directory, which is specified in the configuration file, for all filenames that end with ".py" and writes them into a "__init.py__" file, assigning as "all" the list of filenames. Afterwards it simply imports the complete directory. The class name of the rules is by definition the filename appended with an underscore. Hence, after importing from the directory we can use that information to instance all the classes, which inherit from BaseRule and call the match method.
The result of the program will be stored into a relational database and plotted into different HTML files. Thus the result can easily be examined in a browser. The data is plotted in the HTML format using the CreateOutput module and the jinja2[13] library. It provides an interface and a template language for Python. To visualize all those different fragments this module has a dependency to the SQLClient module. The latter one implements specific functions to retrieve those data from the database.
The SQLClient module as mentioned earlier, provides the functionality to access a SQLite3 database by using the Python sqlite3 module[14]. Furthermore it pro-

[11] https://pypi.python.org/pypi/GitPython/1.0.1
[12] https://goo.gl/mqPMQn
[13] http://jinja.pocoo.org/docs/dev/
[14] https://docs.python.org/2/library/sqlite3.html

vides functions to save all objects that inherit from RawData (see Chapter 5.3) into the database or update those. Optimizing the combination of a relational and object oriented form is the guiding principle for the implementation. Thus the objects will also have attributes that are not persisted into the relational database. The main purpose of that is to create an object layout that is similar to a linked list and hence allows a quick navigation. In general the design tries to fit the tables into an object and not the other way around which is the reason why the layout is still normalized.

6.5 Environment and Tests

The program uses Make for build and dependency management. It further uses pip to install the required Python packages. To get all the dependencies to Python libraries a simple *make install* execution is enough. Besides that there is also a script that setups the table structure into the database file. It executes internally a .sql file that contains all the SQL commands to create the tables. That file can also be used to create the schema for any other relation database. Furthermore, a script runs all the test cases that it finds by simply walking over all existing Makefiles and executing the label test. The whole project contains about 180 test cases. Most of these are unit tests that are basically just testing a function. Some of the tests setup an own database executing the table schema and finally saving the objects into it.
Another interesting part is that all rules implemented for the case study are tested as well. Some of those even use the input of real source files of the django-oscar project as input, ensuring that the rules work in a real world scenario. For the implementation of the test cases a library which uses the Test Anything Protocol (TAP)[15] is used.
A configuration file[16] customizes the program. It further creates logs to track down errors after or while executing the program.

[15] https://testanything.org/
[16] https://goo.gl/iR8Z6c

7 Case Study

7.1 The Project

Django Oscar[17] is an open source e-commerce framework implemented using the Django framework. It aims to have an easy to customize core functionality and hence be a good choice for all kind of e-commerce applications. It has gained up to 145 contributors in five years of development. During all that time 6349 commits were made, most recently in Sept 2015. Thus we can say that the development process is still ongoing. The size of the project right now is 135,3 MB and it contains 1852 files.

7.2 Skills

Figure 13: Identyfied skills in a Django project

As discussed in the methodology chapter we have collected a few skills and their category by examining the Django framework. Figure 13 shows the rough layout in a mind map were a child node (leafs) should be interpreted as a skill. **Naming of the skills:** The name of the skills were found and defined with

[17]https://github.com/django-oscar/django-oscar

the help of the Django documentation and tutorial. This sources were used to create a (long) list of filenames, folders and classnames. The long list was reduced to to a short list of candidates by applying criteria like number of mentions, probability that it is used in a variety of projects and with the goal to get skills that represent a spectrum of different domains. The final short list contained follow words: *ModelAdmin, Model, TestCase, fixture, Form, Migration, ModelForm, MultiWidget, validate, Templates, Sql, url.py, ReadMe.mb, .git, Wsgi.py and Settings.py.* Some of the elements in the list are a bit messy and inconvenient like for example MultiWidget. Therefore a more proper name was found for some of those candidates with the help of the documentation. As a result, the elements of the leafs were selected for the name of the skills. Each of that elements corresponds to one of the names in the list.
After the skills were found a way to group or categorize them was needed. The six groups were defined as follow:

1. Model: The class has Model in it and rule 2 does not apply
2. Forms category: If the class that the skill inherits from imports from *forms*
3. Test category: If the skill has something to do with testing code
4. Database: An operation on the database is performed that is loosely abstracted by Django
5. Administration: They deal with the configuration of Django or another tool in development
6. Other: It stores anything that does not fit to the above rules

Definition of the rules: Due to the design of Django and the Python language itself many skills are defined by the inheritance of a specific class. For example models or migration issues always have to inherit from a specific class to achieve something in the Django framework. That is the reason why many rules work over the *ParseTree-Filter* with the inherit option.
Setup for the case study: For the Case Study we have defined 17 rules[18]. Of those 14 were specific to the Django domain and 3 could be applied to any Python project. The created rules can be found in the Appendix. An underlying assumption of the case study is that the deletion or renaming of a specific file would not indicate any skill. Those operations can be performed for different reasons but do not require any knowledge of the underlying implementation. The file does not even have to be opened. Therefore, it was not concerned as an analyzing case. If we would take those cases into account every delete would give the committer a tremendous amount of experience atoms. In the worst case the amount of lines. That does not seem to be fair. Note that this case differs from removing just a few lines of code. That would be classified as a modification and hence would execute the rules. After all the code was changed by a contributor for sure. The same applies to the adding of a file to Git. That way it can be sure, that the contributor, who added the file, added lines to it as well. Otherwise it would be empty and would return zero skills anyways. The Figure 14 sums up that paragraph.

[18]https://goo.gl/PjO2P1

Operation	Analyse
Adding File	yes
Deleting File	no
Renaming File	no
Modifying File	yes

Figure 14: Overview about the operations and how the tool deals with them

7.3 Validation

After the program and all the unit tests were successfully executed several approaches were followed to validate the result. First of all, the unit tests of the rules were run with examples of the project to verify that the right skills have been discovered. That way the rules can be tested based on real input and we can be assured that the rules work properly.
Besides the automatic unit tests the manual verification of the program was important. The first project to be tested with the code was a Django tutorial[19]. Once it was pretty sure that it works on that small project, the tool was executed on the Django Oscar project. By picking several commits and files, it was checked manually if the result in the database was identical to the expected value. The GitHub visualization feature supported this approach. Thus basically a few interesting files were selected and checked first if all the commits were into the database and second if the changed lines really implicit the found skill. In case the result is valid for all eighteen tested files it is pretty safe to say that the program has no serious bug at the current time. Those manually checked files were afterwards coded as Python test cases[20]. Therefore, it can easily be checked if the result is still valid after a change in our tool . Furthermore those test cases can be seen as a documentation of tested files.
Another approach would be to code an oracle like program that simply uses some kind of grep for several keywords to find an independent reference value. A closer look to that approach, however, revealed some weaknesses. For example we could either be too optimistic or too pessimistic with the assignment of a skill through the search of grep. If we search in every diff for the keyword "Model", we would get by far fewer hits because only the added lines count and the base class was defined in a previous commit. On the other hand, if we always look in the complete content of the file, that was touched during the commit, we would get a far too positive result. Therefore, that approach is just able to give a rough boundary and nothing that has a greater worth for the validation. A more complex oracle would increase the probability of a bug in it as well. Consequently, it won't be a valid source and hence we decided to test manually if the results are valid. To make sure that for every person it is easy to verify that the test cases are correct, all of those follow a specific schema and directly link to the corresponding parts on GitHub. With that transparence I want to make sure that it is easy for everyone to validate the result. The implementation of the program can lead to unexpected behavior in some special scenarios. If that is the case for a testcase a detailed explanation will be given as comment. For example, a commit that just moves a file won't be counted as

[19] https://github.com/Chive/django-poll-app
[20] https://bitbucket.org/Freddy92/bsc.-developer-profiling/src/master/DjangoDevProf/test/t/

commit hence it does not identify a skill. See the following code for an in depth example. A list of tested files can be found in the appendix and now we will show the example layout of such a file:

```
"""
The file that will be tested is 'tests/...../reviews/model_tests.pyy'

    URL: https://github.com/django-oscar/django-oscar/blob/master/tests/
        .../reviews/model_tests.py
"""

from TAP.Simple   import *
from lib import help

plan(10)

filename = 'tests/integration/catalogue/reviews/model_tests.py'
help.load_database()

"""
The file was was touched by five commits and three users.
"""

results = help.get_commits(filename)

eq_ok(len(results), 5 , "Did we got the commits as expected")

dict = {}

for (id, date, name, user) in results:
    if user in dict.keys():
        dict[user] += 1
    else:
        dict[user] = 1

"""
Three users have committed to that file
"""
eq_ok(len(dict.keys()), 3 , "Three users have commited to that file")

values = sorted(dict.values())

eq_ok(values[-1], 2, "Make sure that one use has two commits")


"""
The file has the skill Test/Generell . Eight relevant lines were dropped
and eight lines were added.

    Github: https://github.com/django-oscar/django-oscar/commit/0a1b4a60[..]09e930b9505c7b
```

```
"""

_, name, experiece, _ = help.get_skills(4030 , filename)[0]

eq_ok(name, "Test/Django" , "Check if the right skill was found")
eq_ok(experiece, 8 , "There should be eight ea")

_, name, experiece, _ = help.get_skills(4030 , filename)[1]

eq_ok(name, "Test/Generell" , "Check if the right skill was found")
eq_ok(experiece, 8 , "There should be eight ea")

#Just two skill found Generell and Django as expected?
ok(len(help.get_skills(3305 , filename)) == 2)
```

7.4 Result

That are the collected values of skills for the David Winterbottom

Skillname	Count	Proportion of skill
Administration/VCS	134	0.881578947368
Forms/HTML	1205	0.559684161635
Validation	10	0.113636363636
Forms/Model	1827	0.543426531826
Administration/Buildmangament	300	0.660792951542
Other/DjangoConfiguration	3005	0.880715123095
Database/Fixture	4923	0.926595143986
Other/Templates	27140	0.424168542136
Test/Django	12703	0.728424795
Model/Admininterface	770	0.820895522388
Database/Migration	58	0.148717948718
Test/Generell	16091	0.671409496787
Model/Data	20683	0.612920432657
Other/KnowledgeManagement	4962	0.576039006269

Figure 15: Skills of the contributor David Winterbottom

RQ: What can be found out about the contributors?
After an execution time of about 80 minutes the program has filled the database with the corresponding values. Let's have a look to the profile of the project owner David Winterbottom. He has a total number of 3159 relevant commits. We define relevant commits as the ones that did not just move a file from a folder to another because that operation does not really need a skill. 5688 is the count of relevant commits. As we can see, that user has made about 55 percent of the commits of the whole project. Looking at the results we can easily calculate the average experience atom count per skill of David Winterbottom: the average is 0.6042 and standard deviation is 0.241. It is now interesting to see that

the Migration and Validation skills from him are below the standard deviation (Figure 15). So in that skills he seems to be not as dominant as in the other. On the other hand, he has done more than 90 percent of the fixture related stuff. In general, we can see a correlation between a user who has contributed the most commits and a user who happens to have the most skills. To get those values you can run the SQL command shown in Figure 16.

```
Select Skill.name, count(Skill.experience)
        from Gitcommit,FileVersion, Skill
where
        Gitcommit.UserId IN (
        Select UserId from Contributor where name='David Winterbottom'
        )
        and FileVersion.CommitId = GitCommit.CommitID
        and Skill.FileVersionId = FileVersion.FileVersionId
group by Skill.name;
```

Figure 16: SQL command to get all the skills of David Winterbottom

Most surprising of the discussed things is that just about 15 percent of all the skills in the migration domain were collected by him. That is an interesting fact because it shows that there seems to be some kind of task sharing and another expert exists for the migration domain. If we have a look at the developer 'Maik Hoepfel' we can see that he has collected about 74 percent of the migration experience atoms. As a result, it is obvious to say that he is the Migration expert in that project.
To summarize an interesting insight was gained due to the data because task sharing or domain experts could be identified which would not be possible by simply analyzing the commits. After having a look to the contributors more details about skills itself will be discussed now.
RQ: What skills do the most contributors nominal have?
Another interesting aspect is to look how many people posses a certain skill (Figure 18). That is an interesting question because it could help to identify important persons that are not really replaceable so quickly by another member of the team. To do so we can run the SQL command in Figure 17.

```
Select Skill.name, count(DISTINCT Contributor.UserId)
from Contributor,Gitcommit, FileVersion, Skill
where
        Gitcommit.UserId = Contributor.UserId
        and FileVersion.CommitId = GitCommit.CommitID
        and Skill.FileVersionId = FileVersion.FileVersionId
group by Skill.name;
```

Figure 17: SQL command to identify how many users have a skill

The result shows that the complete database related skills are just distributed between a few users. Only 5 respectively 7 users worked on the ex-

```
Administration/Buildmangament|6
Administration/VCS|11
Database/Fixture|7
Database/Migration|5
Forms/HTML|25
Forms/Model|32
Model/Admininterface|14
Model/Data|66
Other/DjangoConfiguration|20
Other/KnowledgeManagement|57
Other/Templates|77
Test/Django|34
Test/Generell|44
Validation|6
```

Figure 18: The distribution of the skills

pertise field of migration and fixtures. On the other hand, the Model/Data skill that describes the data was used by 66 users.
In fact, that is a pretty interesting thing. A lot of people tend to know or have experience in working with the database model but wouldn't be able to migrate the changes to the database or/and providing interesting start values. An explanation for that is that the most people worked on the surrounding methods of the model objects and less on the definition of the data layout. In the next step we want to analyze this a bit more in depth and check how it relates to the number of commits.
RQ: Do more commits lead to a diverse or different experience distribution?
The majority of contributors have only one commit which is typically in many other open source projects of the same size. In that project 164 out of 190 contributors have less than five commits. Comparing the distribution of the skills with the people contributing very frequently we define those very active people as developers with more than 40 commits to the project i.e. 27 users.
We now plot two tables ordered by the most nominal used skills of that group (Figure 19). After that is done the Levenshtein distance [13] will be used to calculate how similar the output of those two queries is. If they are similar we can say that the number of commits is not correlated to the skill distribution.

To get a value between 0 and 1 we have to normalize the similarity function by using:
$1 - \frac{(LevenshteinDistance(abcdefghijklmn, abdcefghijlnkm)}{12} = 0.58$
12 is the number of used characters. Furthermore an online calculator[21] was used to calculate the value. The result is that the similarity is 58 percent. Thus more than the half is equal for both groups. So it can basically be concluded that there are in general skills that are more and less popular. There does not seem to be a skill that is in particular possessed by people who work very long

[21] http://planetcalc.com/1721

a	Other/Templates	58	Other/Templates	20	a
b	Model/Data	50	Model/Data	16	b
c	Other/KnowledgeManagement	46	Test/General	12	d
d	Test/General	32	Other/KnowledgeManagement	11	c
e	Test/Django	23	Test/Django	11	e
f	Forms/Model	22	Forms/Model	10	f
g	Forms/HTML	15	Forms/HTML	10	g
h	Other/DjangoConfiguration	14	Other/DjangoConfiguration	6	h
i	Model/Admininterface	9	Model/Admininterface	5	i
j	Administration/VCS	8	Administration/VCS	3	j
k	Administration/Buildmangament	4	Database/Fixture	3	l
l	Database/Fixture	4	Validation	3	n
m	Database/Migration	3	Administration/Buildmangament	2	k
n	Validation	3	Database/Migration	2	m

Figure 19: The result of the distribution around the skills

on the project. Furthermore the maximal difference between position of letter in the one and the other word is 2 or in percent 17. To sum this up: In this project there is no strong correlation between using a certain skill and having done a lot commits. Finally, we want to identify how skills and commits are related.

RQ: How many commits carry a skill?

Before closing this discussion lets look how many commits exist that do carry a skill of any kind. The result of the SQL statement in Figure 20 returns the numbers 3768 and 5727. Hence about 65 percent of the commits do carry a skill of some kind. Therefore we can conclude that a way to analyze and classify the other commits had to be found in the future.

```
Select count(Distinct x.CommitId)
        from
                Gitcommit as x,
                FileVersion as y,
                Skill as z
        where
                x.CommitId = y.CommitId
                and z.FileVersionId = y.FileVersionId
Union Select count(Distinct x.CommitId)
        from
                Gitcommit as x,
                FileVersion as y
        where
                x.CommitId = y.CommitId;
```

Figure 20: SQL command to identify how many commits are connected to a file and how many commits exist

8 Concluding remarks

First of all, we used the Django documentation and some successful Django projects to find interesting rules, that define if a user has experience with a certain skill. Those rules operate on four different kind of filters which are *Filename-, Directory-, Content- and ParseTree-Filter*. The first two operate on the path of the file to determine, if it indicates a specific skill. The two following work on the content of the file. It either takes the syntactical structure into account or just the plain uninterpreted content.
Those rules were implemented using Python and classes that were defined in libraries. Besides that the tool defines other helpful libraries that could be helpful for the process of analyzing. Additional modules to deal with Git, Databases, the loading of rules and the creation of output were implemented. Finally a case study on the Django Oscar project was performed. The result was validated using specific test cases to determine, if the result matches our expected observation. As a result, it can be said that it was possible to automatically extract skills from a Django project on GitHub.

8.1 Thread to validity

The approach is based on the assumption that a user is experienced with a certain skill, if he did many changes in a file . Although many papers are based on that theory, it has to be verified for this case. Thus Django developers in a large project should be asked for a statement about their knowledge, which will be compared to the actual result of the program. If we find a correlation between the result of those two approaches it can be stated that the assumption is correct.
Logical operations have to merge the results of their filters. That could take place through defining a dominant side like it is done with the left or right version of an operation. However, if the dominant side of an Or operation does not apply the other side will be returned. If that side returns in average larger values than the dominating one you will get a falsified result. That commit seems to be more relevant than the others but in fact it is not. Therefore, almost equal strong filters should be combined. Another bias is that the equal weight operations use the average. That can in some scenarios lead to the return of unexpected values.
In addition, the rules are usually static. They can't consider all possibilities. For example the directory fixture_is not considered, but the folder fixture would be. Therefore, an adaption of rules for a specific project have to be made to keep precision. Otherwise some skills wouldn't be detected.
Moreover, the output just gives absolute values. There is no real comparison over a complete domain. We can just say in that particular project that person seems to have the most skills of all users. But it cannot be determined how this relates to the average. Therefore, a lot of projects should be scanned, so that we can see the collected experience atoms in a larger picture and finally see what a reasonable value of experience would be.
Moreover, the work is just limited to Git, so that in the end we just see the commits a person has done. Some parts of the code could be auto generated by a plugin or be copy pasted from some other sources, though. Those would still be counted as skill, although the developer does not really need to have any

knowledge of that particular skill.
Last but not least the rules could be a lot more restrictive. Not every modification of a class that is of the type Model really indicated that there would be a skill of the model domain. An example for that could be the simple renaming of a variable or even just the correcting of a grammatical error in the comment section. Thus the rules and commits should be analyzed in a previous step to drop those biases before the execution of the rules takes place.

8.2 Future Work

Some skills like the installing of plugins to the IDE, setting some environment variables or installing software packages tend to change the state of the machine without modifying the VCS. Docker [22] is a new tool used to create containers to make the developing and deploying process easier. In fact it provides an additional layer of abstractions. More and more projects and companies use Docker for the deployment of their applications. It further provides a repository[23] where those containers can be uploaded quickly. A future work could be to have a more in depth look to this software and decide if valuable skills could be mined by analyzing those container repositories and who made what changes to it. Perhaps that approach can help to define and identify more skills and roles in a software project. Especially for deployment, administrations and configuration of IDE plugins.
Although we have extracted a lot interesting data of the VCS and the source files, we in fact just displayed the data. A more interesting task would be to take the data and apply machine learning or data mining techniques to it. Some questions that could be applied in that context could be:

- What skills do developers usually have combined?
- Can we say what skills are most needed due to the specific phase of the development
- Detect more skills by finding patterns in files and commits with the help of the data

Furthermore the rules should be able to learn. Instead of just statically applying the rules it should be able to learn dynamically for example by comparing how similar a file is to a file that actually has that skill. That way the rules could be applicable to projects without modifications for the specific context. In the best case it would automatically detect that a fixture and fixture_folder are similar and therefore should be treated in the same way.
The current state of the program is stable. There is still a lot room for improvement, though, if it comes to the way the filters work. The *ParseTree-Filter* could for example work more detailed and in fact even filter out the lines in a class that really identify a skill. To make that filter mechanism perfect a significantly more developing time is required. The return of that investment would be more precise results.
To find all the functions that determine a skill would be a tremendous work.

[22] https://www.docker.com/
[23] https://hub.docker.com/

Perhaps there is a way to extract automatically interesting variables or functions out of the Django documentation. If a method will be called in a class that inherits from Model and in fact that method has a match in the Django API description of Model, it can be stated that it would be a skill in that experience category. Furthermore an efficient way of data cleaning could be applied to delete content that is leading to wrong conclusions. One of those could be to detect copy pasted or generated code. That should be deleted by the program before the actual analysis starts and hence make sure that the results will just assign the right skills. To do so also in that context data mining and machine learning techniques could be applied to identify "bad" parts in the code and drop them.

9 Appendix

9.1 Rules

Value	Filter	Skill	Link
Settings.py	Filename	Configuration Django	Docu
Wsgi.py	Filename	Administration /Deployment	Docu
Model LeftAnd import models	Parse Tree Content	Model/Data	Docu
Admin.py Or ModelAdmin	Filename Parse Tree	Model/Admininterface	Docu
Templates/ RightOr {%	Directory Content	Other/Template	Docu Docu
url.py	Filename	Other/Router	
Import forms RightAnd Form	Content Parse Tree	Forms/HTML	Docu
Testcase LeftAnd import TestCase	Parse Tree Content	Test/Django	Docu
def test_....(self, ..) Test or Test	Parse Tree Filename	Test/General	Docu
import migrations RightAnd Migration	Content Parse Tree	Database/Migration	Docu
Fixture Or initial_data.(xml or yaml or json)	Directory Filename	Database/Fixture	Docu
.sql RightAnd Insert ... VALUES()..	Filename Content	Database/SQL	Docu
(MultiWidget Or def decompress(self, value)) And import Widgets	Parse Tree ParseTree_method Content	Forms/Widget	Docu
Import ValidationError RightAnd def validate_	Content ParseTree_method	Other/Validation	Docu
Docs	Directory	Other/Knowledge Managmen	
.gitignore Or .gitattributes	Filename Filename	Administration/VCS	
Makefile	Filename	Administration/Buildmangament	

Figure 21: All rules we defined

9.2 Tested Files

Filepath
tests/functional/basket/manipulation_tests.py
sites/_fixtures/comms.json
tests/integration/shipping/model_method_tests.py
src/oscar/apps/order/migrations/0003_auto_20150113_1629.py
src/oscar/apps/dashboard/reports/forms.py
src/oscar/apps/promotions/models.py
src/oscar/apps/catalogue/admin.py
src/oscar/templates/oscar/checkout/shipping_address.html
src/oscar/core/compat.py
src/oscar/apps/shipping/abstract_models.py
src/oscar/apps/dashboard/partners/forms.py
src/oscar/apps/dashboard/catalogue/forms.py
sites/sandbox/fixtures/books.computers-in-fiction.csv
sites/sandbox/fixtures/multi-stockrecord-product.json
tests/functional/customer/auth_tests.py
tests/integration/catalogue/reviews/model_tests.py
src/oscar/apps/offer/abstract_models.py
src/oscar/apps/dashboard/partners/forms.py

Figure 22: All files we tested for the validation of the case study

References

[1] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, 2002, pp. 503–512. [Online]. Available: http://doi.acm.org/10.1145/581339.581401

[2] C. De Roover, "A logic meta programming foundation for example-driven pattern detection in object-oriented programs," Ph.D. dissertation, Vrije Universiteit Brussel, August 2009.

[3] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Software Eng.*, vol. 37, no. 6, pp. 772–787, 2011. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2010.81

[4] A. C. MacLean and C. D. Knutson, "Apache commits: social network dataset," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 135–138. [Online]. Available: http://dx.doi.org/10.1109/MSR.2013.6624020

[5] S. Onoue, H. Hata, and K. Matsumoto, "A study of the characteristics of developers' activities in github," in *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2*, 2013, pp. 7–12. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2013.104

[6] F. V. Rysselberghe and S. Demeyer, "Studying software evolution information by visualizing the change history," in *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, 2004, pp. 328–337. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICSM.2004.1357818

[7] O. Alonso, P. T. Devanbu, and M. Gertz, "Expertise identification and visualization from CVS," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*, 2008, pp. 125–128. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370780

[8] P. Bhattacharya, I. Neamtiu, and M. Faloutsos, "Determining developers' expertise and role: A graph hierarchy-based approach," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 11–20. [Online]. Available: http://dx.doi.org/10.1109/ICSME.2014.23

[9] C. Teyton, M. Palyart, J. Falleri, F. Morandat, and X. Blanc, "Automatic extraction of developer expertise," in *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014*, 2014, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/2601248.2601266

[10] P. Laplante, *What Every Engineer Should Know About Software Engineering.* CRC Press, 2007.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[12] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970. [Online]. Available: http://doi.acm.org/10.1145/362384.362685

[13] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, p. 707, February 1966.

List of Figures