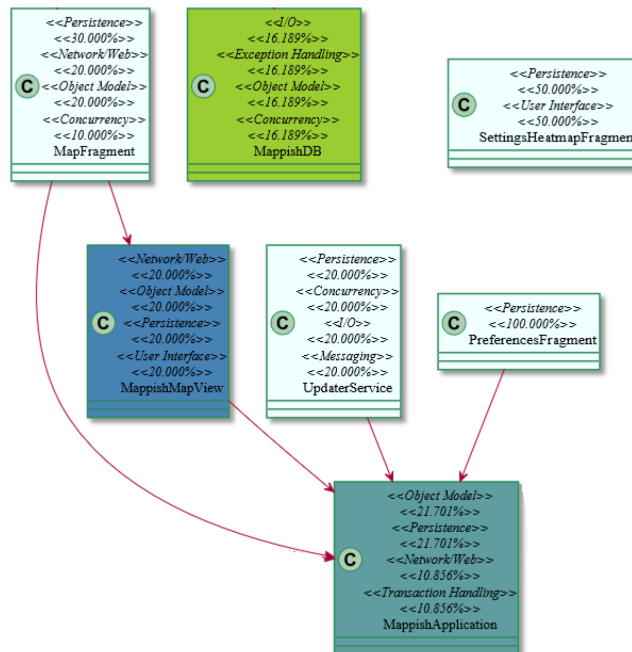
UNIVERSITY OF GOTHENBURG



Reverse Architecting: Automatic labelling of Concerns in Reverse Engineered Software Systems

Bachelor of Science Thesis in the Programme Software Engineering and Management

BERIMA K. ANDAM

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2015

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Reverse Architecting:
Automatic labelling of Concerns in Reverse Engineered Software Systems

BERIMA K. ANDAM,

© BERIMA K. ANDAM, June 2015.

Examiner: JAN-PHILIPP. STEGHÖFER

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Reverse Architecting: Automatic labelling of Concerns in Reverse Engineered Software Systems

(*Student*)
Berima Kweku Andam
Department of Computer Science and Engineering
Chalmers Univ. of Technology and Gothenburg University
Gothenburg, Sweden
vision.ami4@gmail.com

(*Supervisors*)
Michel R. V. Chaudron,
Truong Ho Quang
Department of Computer Science and Engineering
Chalmers Univ. of Technology and Gothenburg University
Gothenburg, Sweden
{truongh,chaudron}@chalmers.se

***Abstract*—A significantly large fraction of time during development and maintenance is spent on understanding unfamiliar parts of software systems. The existence of software documentation, such as software architecture design documentation can significantly reduce the amount of time spent on this task. However in reality, few software systems have an up-to-date documentation because project time pressure makes it impractical to do so. During comprehension therefore, software engineers often try to recover these lost design documentation through reverse engineering. However, current reverse engineered diagrams show only one perspective of a software system; the components that exist in the system and the relationship between them. Often, software engineers require additional perspectives in order to understand how a system works. In this research, we aim to solve this problem by providing one such perspective on top of reverse engineered diagrams. We provide a framework and tool for automatically identifying common system concerns that are found in modern software systems, and then map them back to the software components in the system that implement them. An example of such system concerns are user interfaces, persistence, security etc. A regular question that comes up during comprehension is which software components in a system implement these concerns. We propose a taxonomy of these common concerns, and a framework and tool for automatically identifying and labelling system components that implement them. Our framework is based one lightweight static analysis. It calculates three metrics that are then used during identification. An evaluation of the Concern Detector tool (and in essence the framework) on 4 software systems showed that, authors of the systems agreed 65.5% - 76.8% with the tool's classification of components in their systems. This indicates that, the tool is useful for describing the roles of these components in terms of implementing these concerns. The current implementation is for the java programming language; however the approach is designed to be generalizable for most object oriented programming languages.**

Keywords*—Reverse engineering, program comprehension, concern, static analysis, software metrics

I. Introduction

Comprehending large software systems is often a non-trivial task[1]. This task is made even harder when an up-to-date documentation of the system is unavailable. In practice however, very few software systems have an up-to-date documentation. This is due in part to the fact that most useful software systems undergo a brief period of development in which time pressures make it impractical to do so[2][3].

This period is then followed by a longer period of maintenance, feature addition and adaptation, during which software engineers spend a significant portion of their time trying to comprehend unknown parts of the software system [4] [5]. With inadequate documentation and a large system, this comprehension task can become even more time-consuming, expensive and difficult [6] [7].

Reverse engineering source code into class diagrams, is one way developers try to recover lost design in order to simplify this comprehension task. Researchers[8] have however found that developers find the use of reverse engineered class diagrams limited for system comprehension. During an experiment[9], developers were found using strategies such as interacting with the user interface to test program behaviour and using debuggers to elicit runtime information. This is so that they can identify starting points in the code for maintenance tasks and then manually read and filter out the code, based on experience. By doing this they also tried to map software features they needed to fix to components in the system that implement them.

The problems with the above strategies, however are that, attempting comprehension through manual code reading can be both tiring, expensive or even impossible if the system is really large[9]. It is also just as inefficient to try to figure out which software components implement what system "concerns" by manual code reading[9].

Even though reverse engineered class diagrams do a fairly good job of giving an overview of the components in a system and the relationships between them, its failure to provide other perspectives of the system that software engineers look for when trying to understand unknown software systems, may be the reason why developers avoid using them. Previous research in the area of software visualization has shown that, providing multiple perspectives of software systems can improve its comprehensibility [10].

In this work, we aim to provide an automatic way to add one such perspective on top of reverse engineered class diagrams, in order to improve its usefulness for program comprehension. We aim to do this by abstracting a systems reverse

engineered class diagram, to show cross project, cross platform concerns that are common in modern software systems.

These concerns include common system properties, for example User Interfaces, databases, multi-threading, security and authentication and so on. These properties are so common that most programing languages provide libraries to help build them.

Modern applications may provide all or a subset of these properties depending on their purpose. When trying to comprehend software systems, regular questions that come up are centred around which software components in the system implement these properties. In the context of program comprehension, the possibility to have this automatically mapped to software components that implement them, can save time that would otherwise be used reading source code in order to get similar information. The fact that these broad functionalities are found in most programming languages and across software platforms, could thus provide a very system independent and cross platform way to provide a useful perspective of software systems for program comprehension.

The aim of this research work is thus, to provide a cross project, cross platform, abstraction mechanism that would give a software engineer the ability to view a system, from the perspective of which of these concerns a system has, and which classes in the system implement them. We do this by providing a tool supported framework that uses light-weight static analysis of source code properties to automatically identify these concerns. The tool then visualizes this by labelling each class in a reverse engineered diagram (produced by another tool in our previous research)[1] with the concerns in the system that it implements. We envision two scenarios where such a perspective could be useful for code comprehension:

- A new developer has to maintain a system with 5000 classes. He understands that the system has a graphical user interface, saves user data to a server, and provides security and authentication. But has the difficult task of figuring out which of these 5000 classes implement each of these concerns. This perspective could help with this mapping.
- Our developer is now faced with the task of fixing a bug that results in faulty entries to the database. Instead of manually opening and reading each of the files to find the source of the fault, with our approach it is possible to narrow down to only classes that perform database entries.

In this work, we have mostly focused on realizing the approach for one object oriented programming language (Java). Even though we have designed the approach to be largely generalizable to most object oriented programming languages. It is worth mentioning that our tool is built on-top of an already existing software abstraction too (SAAbs)[11]. More about this tool is presented in section III.

The **contributions** of this paper are to:

- Introduce of a taxonomy of common cross-project, cross platform concerns that software engineers look for during comprehension.
- Propose 3 software metrics (based on light weight static analysis) for automatic identification of these concerns.
- Provide an implementation of this framework and an evaluation of the approach that can serve as a benchmark for further studies.

The remainder of this paper is structured as follows: Section II defines concerns, and introduces our taxonomy of common concerns found in modern software systems. It also talks about the software properties that are used by our metrics for automatic identification of these concerns. Section III discusses related research and Section IV Indicates our research questions. In Section V, a description of our approach is given whiles Section VI describes our experiment. We present the analysis of results in Section VII and discuss our finding in Section VIII. Finally, conclusion and talk about future work is presented Section IX.

II. Concerns and properties for identifying them

We follow Sutton and Rouvellou's[12] definition of concern as "any matter of interest in a software system". In the context of our research, the matter of interest is any one of these cross project, cross platform software properties that programmers look for when trying to understand a software system.

In table I we present a taxonomy of these concerns. This list was generated by analysing all the packages that are provided in the JavaSE, JaveEE and Android API. The concerns are however generalized so that they are not specific to the java platform. In fact most of them are concerns that other researchers have attempted to automatically identify using other approaches[12][13].

Concern	Description
User Interface	Display of interface to interact with end user
Persistence	Saves data to permanent storage, such as in Databases or files.
Object Model	Defines an object in the application domain (e.g. Room in a hotel booking system)
Security	Performs activities related to security such user authentication; password checking, encryption etc.
I/O	Input output operations such as reading and writing to files, printing
Concurrency	Performs actions related to multithreading
Exception/Exception Handling	Actions related to creating, throwing, catching and handling of errors.
Network/Web	Performs actions related to accessing Network and web resources.
Parser/Interpreter	Responsible for parsing interpreting files from one format to another, for example converting XML to objects.
Messaging	Performs actions related to messaging
Event/Event Handler	Performs actions related to Creating, throwing and handling of events in the system.
Geo Location	Performs actions related to location tracking

TABLE I. Concerns and Descriptions

We propose two metrics to automatically identify these concerns in software systems. A third metric is deduced from these initial two.

- Library Imports and variable types used (LIB metric)
- Text Analysis (Text metric)
- Deduced (Combined metric)

[1] More about this in III

The subsections II-A, II-B and II-C describe source code properties extracted to calculate these metrics and section V presents the algorithms used in the actual calculation of the metrics.

A. The LIB metric

Modern object oriented programming languages such as Java, C++ have native and 3rd party libraries that are designed to be very cohesive. For instance, the Java package java.swing is a known package for creating user interfaces. This cohesion is also true for other Java packages for example java.security which provides an API for introducing security into an application. It is possible therefore, from looking at the list of packages that a class uses, to deduce what concerns a class implements to a reasonable degree. To be more accurate with this deduction, we can also analyse the proportions of different types of known variables that are declared inside the classes. For instance the proportion of actual objects declared that are associated with a certain concern. This should give a much more accurate picture of the class since it is possible to find scenarios where packages are imported and yet not used in the body of a class definition.

B. The Text Metric

Another property that we identified as useful for identifying concerns in a class is the type of words used in the source code of the class. It is quite intuitive to try to figure out what a class does by looking at the class' name and the names of methods inside it. Words found inside the definition of a class thus, may be a good indication of concerns that are implemented by the class. In our approach we therefore analysed such words found in a class, for links with commonly used words associated with each concern.

In order to assemble such a list of words that are commonly associated each concern, we analysed the descriptions of packages that are associated with these concerns and extracted the most frequently used words in the descriptions. This word list is then cleaned of regular English stop words as well as java key words that have no meaning in the context of concern identification. Examples of such words include class, public etc. A complete list of the stop words can be made available on demand. Figure 1 shows a diagrammatic view of this process.

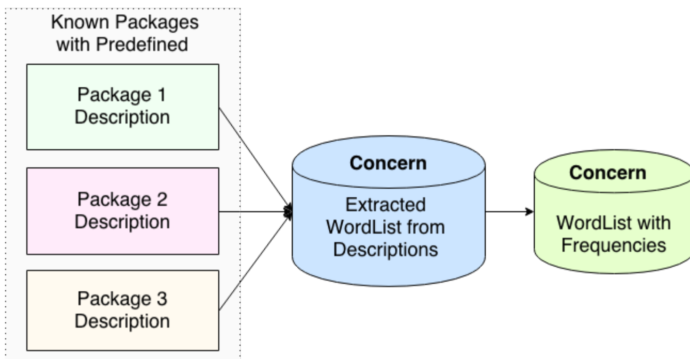


Fig. 1. Extraction of regularly used word for a concern

C. The Combined Metric

In order to mitigate the weakness of the two identified metrics we came up with a third metric which combines the predictive power of the two metrics identified above. This metric is basically an average of the two previous metrics, therefore halve of its classification information comes from each of the previous metrics.

The approach to calculating these metrics is presented in section V.

D. Extracting Source Code elements

In order to easily extract these source code properties used in the calculation of the above metrics, described in the subsections II-A, II-B and II-C above, we built an infrastructure to support querying and fact extraction that are based on srcML (Source Code Mark-up Language), XPath and WordNet.

srcML is an XML representation of source code that supports both data and document views of source code. The srcML format supports querying of source code elements using standard XML tools. A useful and efficient tool that was used for translating Java source code to scrML is freely available[2][14].

In order to find the source code elements within the transformed srcML file, we used XPath; an XML standard for addressing locations in XML.

During, extraction of words for calculating the Text metric II-B, it was common to find multiple forms of the same word used in a class. We solved this problem by stemming each word in order to get their root form using WordNet[3][15].

III. Related work

The core subject of our research is the abstraction of reverse engineered class diagrams in order to improve its usefulness for program comprehension. Thus in this section we discuss previous research work in the area of abstracting reverse engineered class diagrams and the approaches used in relation to our own.

Dragan[16] worked on the classification of classes into Boundary, Control and Entity. His classification was based on the prominence of certain method stereotypes in the classes, which tell their main role in the system. Validation of their approach was carried out on 5 open source projects. 95% of the classes in the system were stereotyped by the approach and developers via manual inspection agreed with the results.

Aditya Budi et al [17] also worked on a similar classification but focused on the identification so that they could give feedback on design flaws that may exist in the system. Evaluation of their approach was done by analysing programs written by novices and expert developer to show robustness of their approach. Another researcher, [18] worked on the classification of classes into roles such as: Service Provider, Controller, Coordinator, Interface etc. but also used lexical analysis of class names and other cues in the code as the basis for the classification. Zaidman et al. [19] worked on

[2]See: www.sdml.info for translator tool download
[3]Available at: https://wordnet.princeton.edu/

a technique based on coupling and web mining to identify classes that had a lot of ”control” in an application.

Other researchers have shifted entirely from the identification of known patterns to condensing the often huge outputs of reverse engineering tools. The goal is to give the user the ability to scale the class for abstraction based on the ”most important” classes in the system. One such research[11] produced a tool supported framework that uses a machine learning algorithm to rank classes based on a score of predicted importance. This ranking is then used as the basis for software architecture abstraction and visualization. The developer is able to interactively explore a reverse engineered class diagram at scalable levels of abstraction. Hence enabling them to understand and learn the software architecture from the bottom up view or from top to bottom view depending on what is useful for performing a specific task.

These afore mentioned work attempt to recover lost design of software systems by identifying patterns, ”important classes” and class intents. They are then used to re-document or abstract reverse engineered class diagrams in order to improve their comprehensiveness. In contrast our approach to abstraction is to use broad software features to provide a feature perspective of the system that can be used during maintains or for general comprehension. We also build on the tool produced by Osman et al[11] in order to add the concern perspective to the tool.

IV. Research Question

In this section we describe our main research question and our 2 sub-questions. The main research question is: ***How can concerns be automatically identified in source code.*** In order to provide an answer to this research question the following sub-questions need to be answered:

RQ1: What is the performance of the LIB metric, text metric and combined metric for automatically labelling class concerns?

RQ2: How does the performance of the 3 metrics compare?

V. Approach

Our approach to conducting this experiment is described in this section.

A. Overall Framework

The overall framework for identifying the concerns a class implements, regardless of the concern metric chosen is shown in figure 2.

The process begins with a systems source code as the input (Step 1). The source code is then transformed to a srcML file representation (Step 2). For each class in the system, library imports and object types or class and method names are then extracted depending on the concern metric selected (Step 3). The output of this phase depending on the metric selected is either a list of words or a list of libraries and variable types used.

In the concern extraction phase (Step 4), concerns implemented by each class are identified by using different algorithms depending on the concern metric selected. The algorithms map the properties extracted from the classes in the previous phase to either the list of words describing a concern(in the case of the text metric) or a list of known libraries (in the case of the LIB metric). In this step, the proportion of each concern found is also recorded as a percentage of the total class.

Finally the class is labelled with the concern with the biggest percentage when the result is rendered as a class diagram. However the class retains the list of all identified concerns so that it is possible to re-label the diagram depending on what concern a user is interested in. Figure 4 shows the results of such a classification. The subsections V-B, V-C, V-D describe the process for each individual metric in detail.

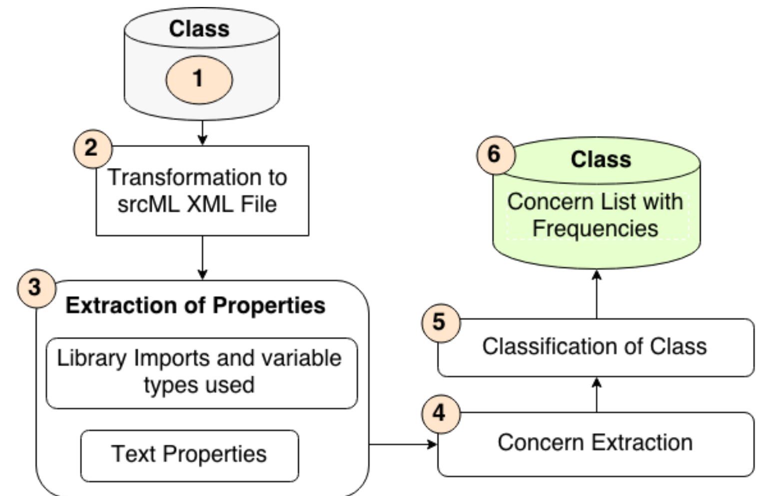


Fig. 2. Overall framework

B. Identifying concerns using the LIB metric

The process for automatically detecting concerns using the LIB metric approach is summarized in the steps below.

1) The process begins with the source code of the system.
2) Convert the source code to a srcML file representation to make it easy to query the code to retrieve specific code structures.
3) Run XPath queries on the srcML to extract the package imports and types and frequency of variables declared in each class.
4) Load a list of known libraries and variable types associated with each concern.
5) Compare the extracted packages and types from the class to the known libraries and types.
6) Record each identified concern along with it frequency in the class.
7) If a Variable type refers to a type defined in the system, add all concerns found in that class, to the list of found concerns in this class.
8) Finally, divide each concerns total by the total of all concerns to derive how much of the implementation of the class goes towards each identified concern.

A visualization of the process is shown in figure 3 below:

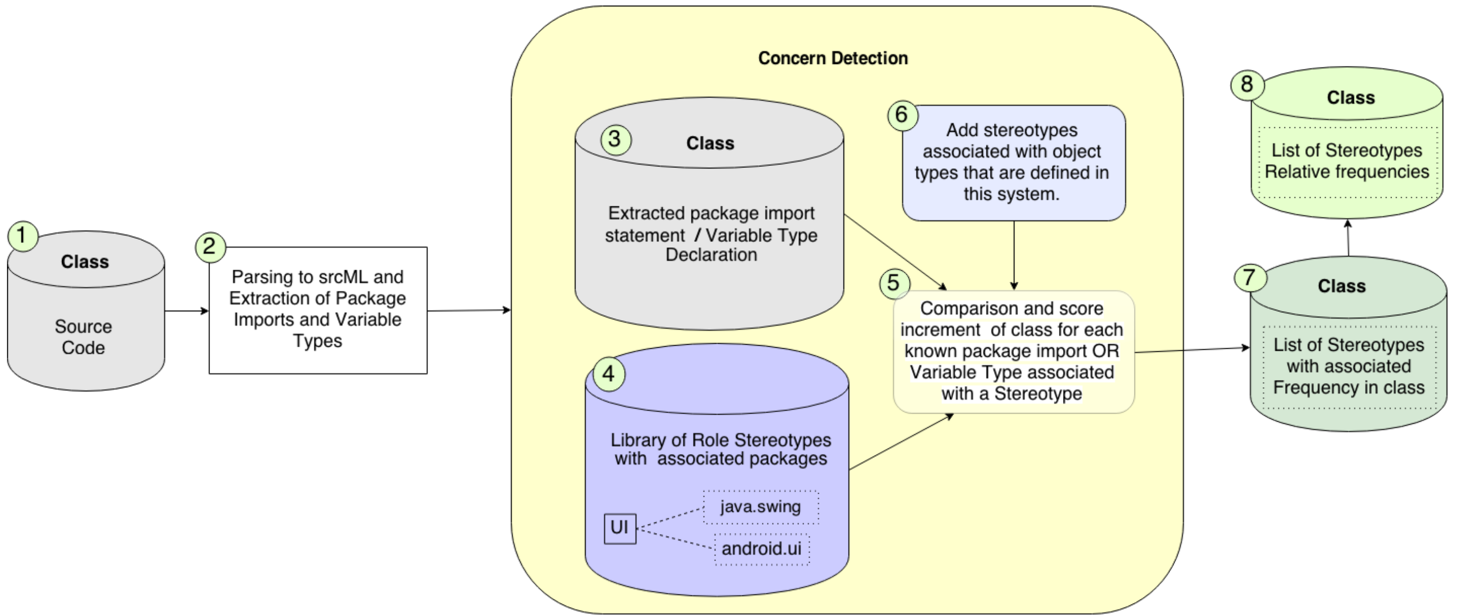


Fig. 3. Identifying Concerns using LIB metric

C. Identifying Concerns using the Text Metric

Just as for the package imports, analysing the classes using the Text analysis approach follows these steps:

1) The process begins with the source code of the system.
2) Convert the source code for the system to a srcML format.
3) Execute XPath queries on the XML representation to extract the class name and variable names for each class.
4) Convert the list of names into a wordlist by splitting camel cased words into multiple words, according to java naming conventions.
5) Remove English stop words and other java key words are that have no meaning in this context.
6) Stem the words using WordNet[15].
7) Record unique words alongside their frequencies in order to establish the weight of each word in the list.
8) Load a list of words that describes each concern.
9) In order to find concerns present in each class, compare the list of words found in the class with the list of words that describe each concern. If a word matches a concern then the weight of the word in the concern is multiplied by the frequency that it occurs in the class to get the weight of the concern in the class.
   An equation to describe the calculation of the weight of each concern in a class as described is given below.

$$WCIC = \sum_{w_1 \ldots w_i} (WWIC * FWIC) \quad (1)$$

   w - Word in class that matches word in concern description list.
   WCIC - Weight of concern in class.
   WWIC - Weight of word in class.
   FWIC - Frequency of Word in Class.
10) Finally, the list of identified concerns is normalized by dividing the weight of each concern found by the total weight of all concerns. Just as for the package import classification the class is decorated with the colour of largest concern found when a class diagram visualization of the project is generated. The list of all identified concerns is also maintained for each class.

A graphical visualization of the process is show in figure 5 below.

D. Identifying Concerns using the Combined Metric

Since the combined metric is an average of the classifications of the LIB metric and the Text metric the process for calculating it includes first calculating the metrics for the afore mentioned and then averaging the results. The process is described in the following steps.

1) Calculate the LIB metric
2) Calculate the Text metric.
3) For each class add all Concerns found by both the Text metric and the LIB metric along with their frequencies to the list of concerns found for the Combined metric. If a concern is found by both, only combine their frequencies but add the concern once.
4) For each concern in the list divide the frequencies by two to get the average weight of the concern in the class.

An equation to describe the calculation of the weight of each concern in a class as described is given below.

$$WCIC_{cm} = (\frac{WCIC_{lm} + WCIC_{tm}}{2}) * 100 \quad (2)$$

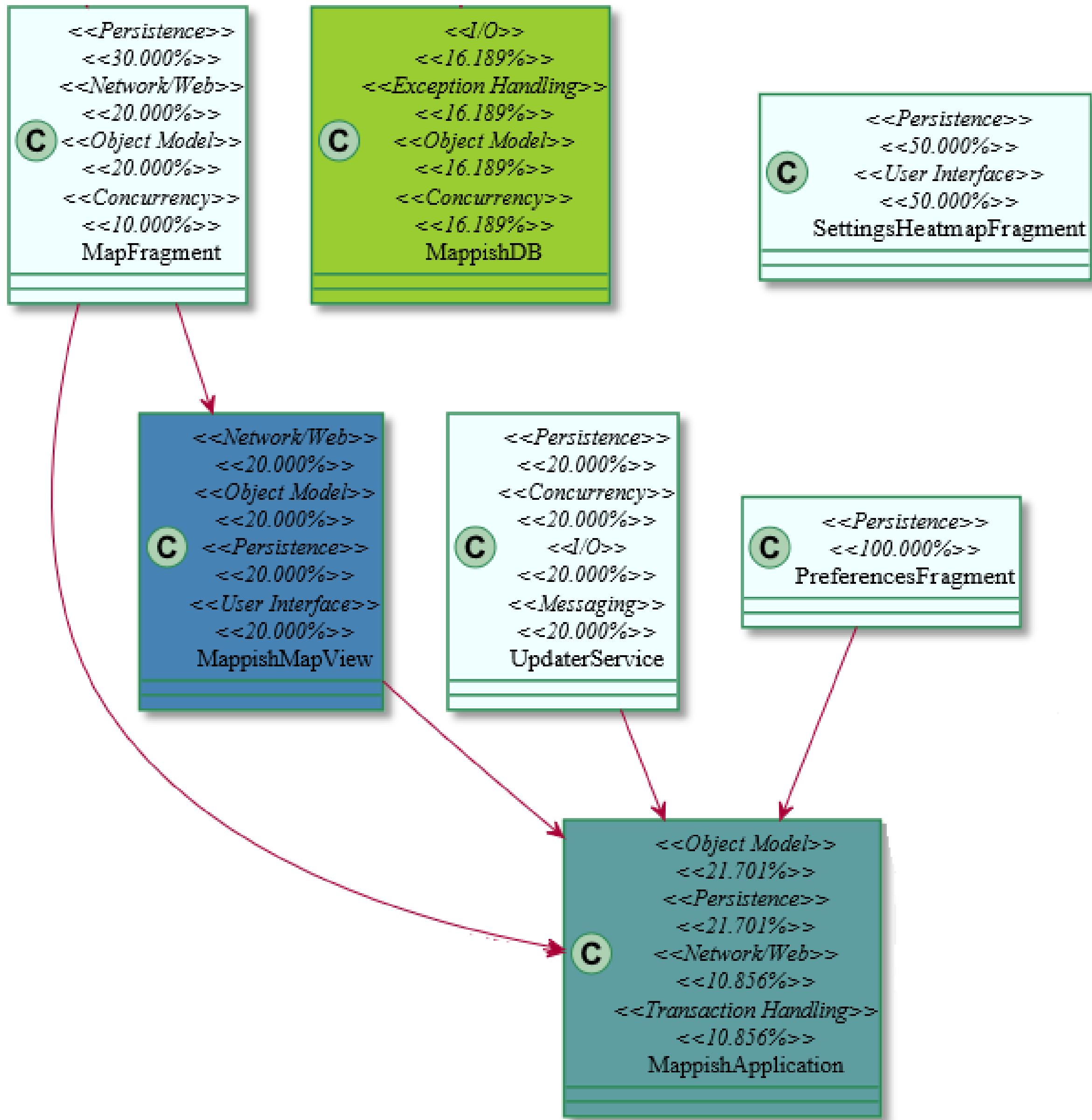


Fig. 4. System with concern perspective visualization

WCIC - Weight of concern in class.

cm - Combined metric.

lm - LIB metric.

tm - Text metric.

E. The Concern Detector Tool

The Concern Detector tool is implemented as a component on top of the SAAbs tool [11]. Its general mechanism for identifying concerns is implemented in accordance with the steps described in the overall framework; figure 2, section V.

In order to extract data for the calculation of the metrics, the source code must be converted to a srcML file outside the tool. Extraction of the source code properties is then done using XPath and WordNet as described in section II-D. At run time, the user has the option to select which of the metrics to use for concern detection. Implementation of the identification of concerns using each of the metrics was also done as described in subsections V-B, V-C and V-D respectively.

Finally, the Concern Detector labels the classes with the identified concerns and passes it on to the SAAbs tool for visualization as a class diagram.

VI. Experiment Description

In this section we describe our case studies and the evaluation measures used for analysing the results of the classifications.

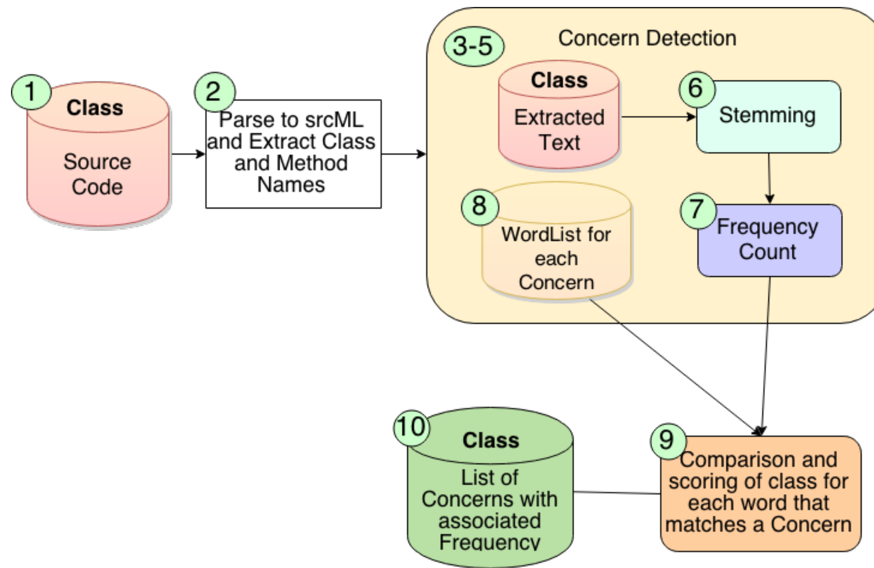


Fig. 5. Identifying Concerns using Text metric

A. Case Studies

In order to assess the performance of the ”Concern Detector” tool and indirectly the framework we applied the ”Concern Detector” tool to 4 projects implemented in Java and developed for two different platforms. Two of the projects are Android apps and the other 2 are stand-alone desktop apps. We tried to analyse these different applications in order to assess the generalizability of the framework. The Size of the projects and the number of classes used in the validation are shown in table II. The projects include:

- Finite Automata Simulator - UML case tool
- SAAbs - Reverse Engineering tool [4]
- WineTracker - Android App
- Mappish - Android App

Project	Total Classes	Total Classes Validated
Mappish	28	12
Finite Auto	36	12
SAAbs	19	7
Wine Tracker	36	31

TABLE II. CLASSES VALIDATED FOR EACH PROJECT

B. Evaluation Settings

The validation study applies a semi-structured interview method. Subjects are first asked questions from the first part of the questionnaire (subject background). They are then asked to describe the purpose of their software. Then, they are introduces to a demonstration of the tool (Concern Detector) and then to the classification the tool has made of their software. An explanation of what each concern means is also given to the subject. Then they are asked to explain how their software works to the researcher, starting from the most important classes that a developer needs to know in order to be able to understand their system. We had budgeted for one hour per interview so the subjects were asked to start from ”the most important” classes in the system. After they explain the purpose of a class, they are asked to look at the results of 3 diagrams that represent the classifications from the 3 concern metrics. They then have to select which of the classifications best describes what concerns the class implements and to what degree the classification is accurate from a scale of 0-100%. They are also allowed a choice of none of them, if none of them describes the class’s responsibility. The entire period of the interview is voice recorded.

[4]Available online at: https://github.com/aislimau/SAAbs

C. Subject Selection

In order to get as accurate feedback as possible from the validations, subjects selected were developers who implemented the systems that are analysed. Three of the subjects were researchers, and one was a student. All subject had at least 2 years experience developing systems with the Java programming language. With regards to experience with UML, one subject had more than 8 years experience, 2 had 5-6 experience and 1 had 1-2 years experience. All subjects were members of the same department as the researchers but were not directly involved in the implementation, development or discussion of this research.

There were two other case studies that were bigger systems; however we weren’t able to get an expert who was part of the development team or who knows the system well. Consequently those systems were not included in our study.

D. Data Collection Instrument

Data collection was done by administering a semi-structured questionnaire specially designed for this study [20]. The questionnaire was divided in two parts and includes:

1) Part A: Respondents Background: In this part we gathered information about the subjects knowledge of reverse engineering, their experience of using UML and whether they were authors of the system under analysis or not.

2) Part B: Concern Classification: The final part of the questionnaire is aimed at gathering information about the author's evaluation of the classifications of the metrics. The author is asked to select and explain important classes in the system. After they explain the purpose of a class, they are asked to look at the results of 3 diagrams that represent the classifications from the 3 concern metrics. They then have to select which of the classifications best describes what metrics the class implements and to what degree the classification is accurate from a scale of 0-100. They can also select none if none of them best describes the class. A sample of the questionnaire is shown in the appendix; section X.

3) Questionnaire Administration: Questionnaire administration was done in person by the researcher after the participant had been introduced to the tool.

VII. Analysis of Result

In this section we describe the analysis of results of our experiment. Each subsection is structured to answer the research questions specified in Section IV. In order to find out the performance of the metrics, we recorded for each metric, the frequency with which the author agreed with its classification out of the total of all classes validated in each project. We also recorded for each agreement if the author agreed with the classification 100% or lower. We then compare the results of the metrics' scores to each other.

A. RQ1: What is the performance of the LIB metric, text metric and combined metric for automatically labelling class concerns?

The overall results of the evaluation of the metrics on our case studies are shown in the figures that follow. Figure 6 shows the average levels at which the authors agreed with the results of each metric's classification for all projects. The average levels of agreement with each metric's classification for individual projects is shown in figure 7.

The average frequency with which the authors selected each individual metric as better than the others is show in figure 8. The average frequencies for individual projects are show in figure 9 below.

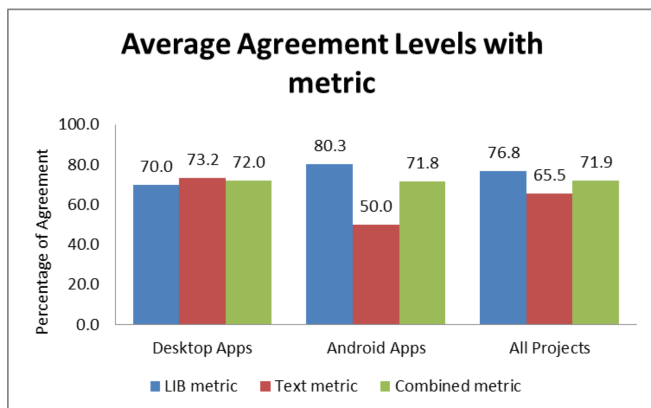


Fig. 6. Average Levels of Agreement - All Projects

As seen in figure 6, overall all the concern metrics were decently accurate at identifying concerns that a class implements

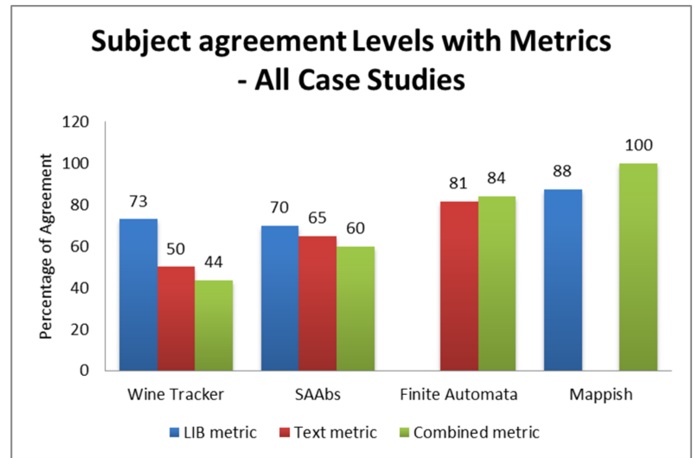


Fig. 7. Subject Agreement Level - All Projects

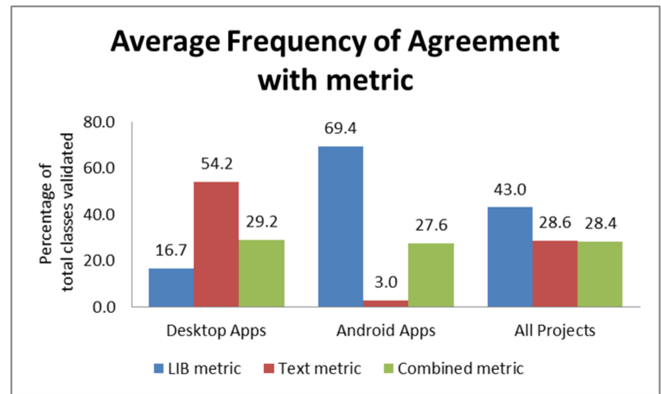


Fig. 8. Average Frequency of Subject Agreement

(Average Agreement>60%). Across all the projects in our case study, the LIB metric was the most accurate at identifying concerns with 76.8% author agreements with its classifications. The combined metric was the second most accurate with 71.9% agreements and finally the Text metric with the least accuracy of 65.5% agreements. Average agreement levels for each metric for individual projects is shown in figure 7.

It is worth noting that the average accuracies stated above are affected by the number of times a metric is chosen as the best classifier for a class. As seen in figure 8, the authors agreed 43% of the time with the LIB metric's classifications. The Text metric follows with 28.6% of total analysed classes and the Combined metric is the least with the number of best classifications with 28.4%. Average frequencies of agreement with each metric's classification for individual projects is shown in figure 9.

B. RQ2: How does the performance of the 3 metric compare?

The metrics' accuracies were also found to vary slightly for each of the software platforms. For the systems implemented for the android platform (figure 6), the LIB metric was still the most accurate with 80.3% average agreement with its classifications for the classes in the android systems. Again the combined metric followed closely with 71.8% agreement, and the text metric was the least accurate with 50% agreement.

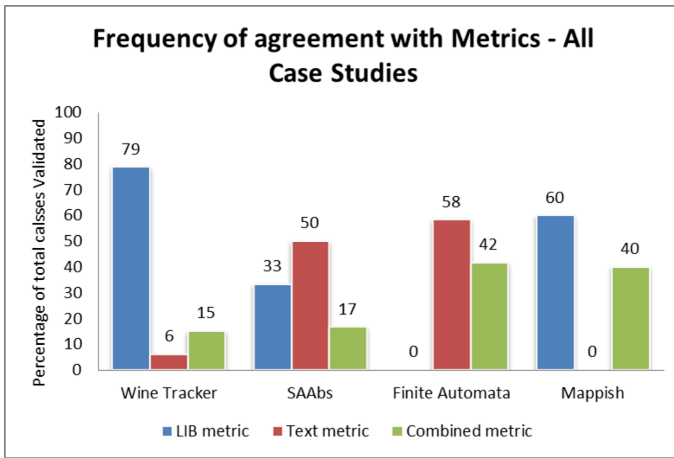


Fig. 9. Frequency of Subject Agreement with Metric - All Projects

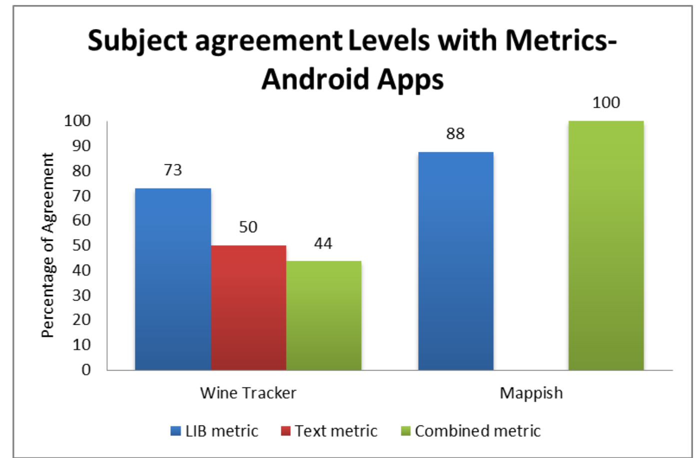


Fig. 10. Subject Agreement Level - Android Apps

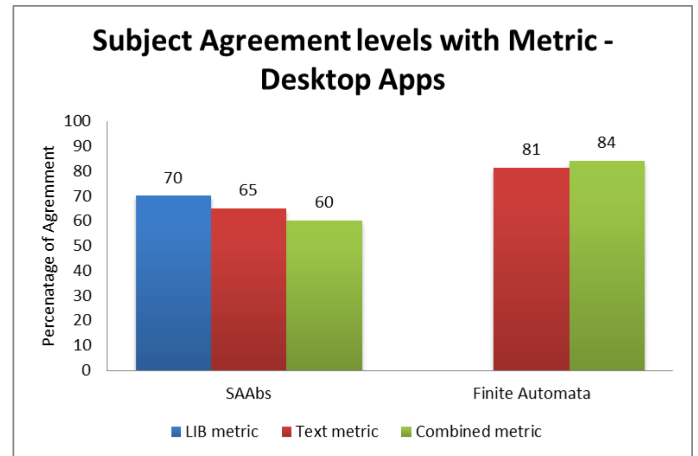


Fig. 11. Subject Agreement Level - Desktop Apps

Average agreement level for each metric for individual projects in this domain is shown in figure 10.

However, for the desktop applications (figure 6), the text metric performed slightly better than the combined and LIB metrics with agreement of 73.2%, 72 and 70 on average respectively. Agreement level for each metric for individual projects in this domain is shown in figure 11.

Again it is worth noting that the average accuracies of the metrics were affected by the number of times a metric is chosen as the best classifier for a class. For the systems developed for the android platform as seen in figure 8, the authors agreed 69.4% of the time with classifications done by the LIB metric. The Combined metric follows with 27.6% of total analysed classes and the Text metric is the least with 3%. Average frequencies of agreement with each metrics classification for individual projects in this domain is shown in figure 12.

For the Desktop Applications (figure 8), the Text metric was the most frequently agreed with metric with 54.2% of analysed classes. The combined metric was next most frequent with 29.2% and the LIB metric was least frequent with 17% of analysed classes. Average frequencies of agreement with each metrics classification for individual projects in this domain is shown in figure 13.

VIII. Discussion

A. *Results from validation*

Results from the analysis of our case studies, from Section VII show that by applying our light-weight analysis framework, we obtain accurate information about the concerns present in a system and also about which classes provide the implementations of these concerns. We also obtain useful information about the percentage of each class's implementation that goes towards the implementation of each concern.

Our simple graphical visualization of this concern perspective of the system on top of a reverse engineered class diagram, gives a useful brief overview of the system that can aid developers during program comprehension. In addition to this, it also provides the user with information concerning what combination of concerns each individual class implements, and what percentage of each class' implementation goes towards each of these concerns. Just as discussed in the introduction, the concern perspective is also useful for maintenance tasks where the developer has to find the source of a bug related to a specific concern. Equally it will also help them if they are looking to extend a system feature related to a specific concern.

Performance of our 3 proposed metrics for automatic identification of these concerns were generally very good, with accuracies ranging between 65% and 77% across all case studies. From analysing the reasons given by the authors for not agreeing entirely with the results of the metrics, we realized that there was a correlation between when classes had a relatively high number of concerns detected and when authors disagreed with the results. From figure 15 we can see that across all projects, the authors selected the metrics with the least number of concerns 57.3% of the time. However, they also selected the metrics that detected the highest number of concerns for each class 42.7% of the time. This shows that the performances of the metrics were not severely affected by how well concerns were separated in a class. A conclusion that can be drawn from this is that, the metrics perform slightly better in cases where the separation of concerns in a system is

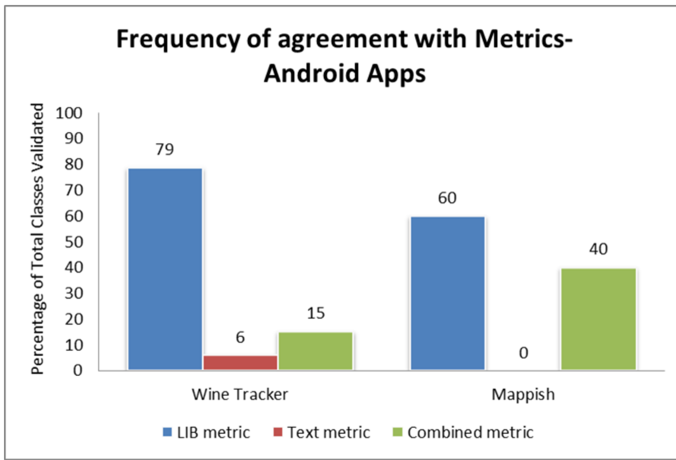


Fig. 12. Frequency of Subject Agreement - Android Apps

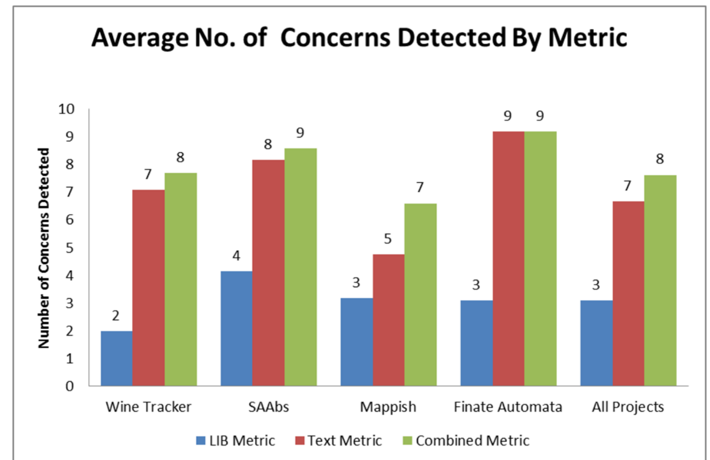


Fig. 14. Average Number of Concerns Detected by Metric

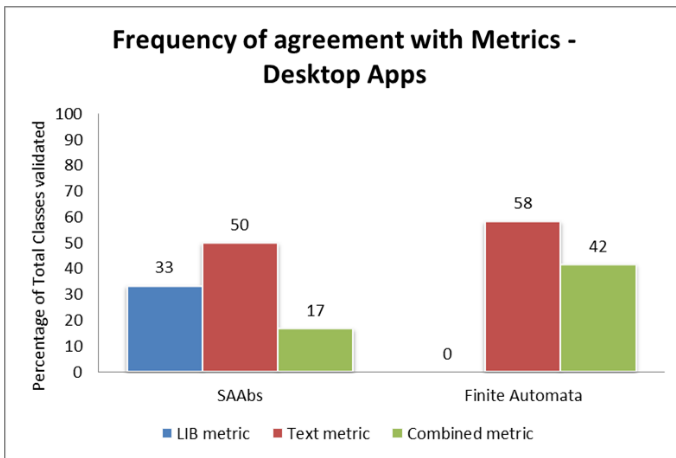


Fig. 13. Frequency of Subject Agreement - Desktop Apps

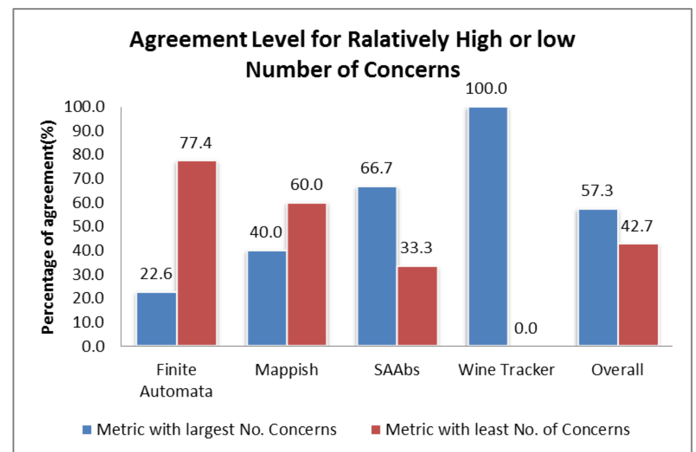


Fig. 15. Number of times authors chose the metric with the least or highest number of detected concerns

clear, even though its performance is not severely affected if this is not the case. Generally, the Text metric identified more concerns per class than the LIB metric as shown by figure 14. Authors found during validation that the Text metric also falsely identified concerns. Weakness of the metric that could cause such a problem is discussed in the next paragraphs.

Overall the LIB metric performed best in terms of accuracy of concern identification. However, it is worth noting here also that, the performance of the LIB metric relies on knowing the libraries and object types used in a system. Therefore, when a substantial amount of the libraries used in the system are unknown, the results of this metric could be significantly affected. It may result in unreliable or even non-existent results. Even though a fair amount of libraries were loaded to the lib metric during classification of the systems in our case study(as shown in figure 16), in a realistic context, there will always be new libraries that are unknown to the tool. This is because new libraries are implemented all the time. Therefore, we have provided a mechanism to feed additional mappings (of libraries and associated concerns) to the LIB metric through a simple text file. A user adds new libraries by inserting additional lines to the text file. Each line must contain the name of the library/object type, and the associated concern separated by a comma. A complete list of the libraries feed to the lib metric is provided in tables III, IV,V,VI in the appendix.

The Text metric was the best metric for the desktop apps but overall the least most accurate metric. Its strength lies in using the semantic meaning of words used in the class, such as class and method names. Similar to the weakness of the LIB metric, the accuracy of the classification of the Text metric could be significantly affected if words used in the method names do not match the wordlist that are regularly used to describe a concern(As described in section II-B). We believe that the existence of some ”noise” words in the wordlist generated for each concern affected the performance of the Text metric. The words were detected upon manual inspection of the list (The complete list of words used for each concern are found in tables VIII, IX, X, XI, XII, XIII, XIV, XV, XVI, XVII, XVIII, XIX in the appendix). Since the problem is related to the amount of descriptions of concerns analysed to generate this list, a natural solution to this problem would be to include more description text for each concern, in the analysis process. However, we werent able to do this due to time constraints. In future work this, could be done to improve the quality of the wordlist and in turn the quality of the results of the Text metric.

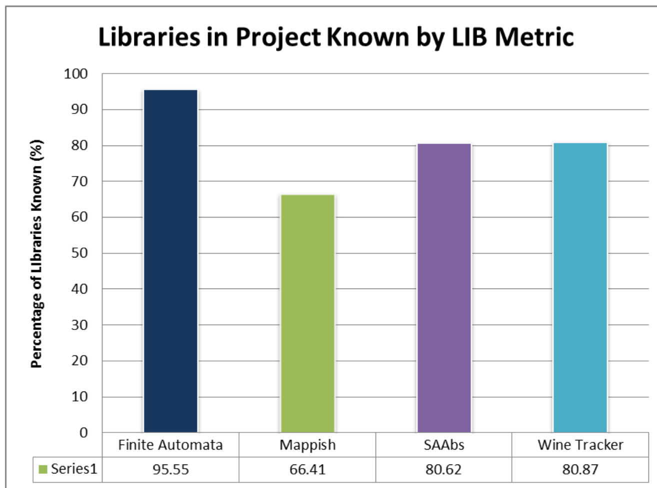


Fig. 16. Percentage of Libraries known by the LIB metric

The Combined metric as expected was the most stable of the metrics. This is no surprise since it an average of the other two metrics. Even though the two other metric can be very accurate in specific domain or when their perquisites are met, the combined metric is probably the safest bet for mitigating the weakness of the two.

B. Threats to Validity

The assessment of the Concern Detector tool is subject to a number of threats to validity.

1) Internal Validity: The classification of the LIB metric relies on the list of known libraries and object. In the case when this is not available, it relies on the users knowledge of the concerns these libraries implement and thus being able to add it to the list of know libraries. If this is not the case, the results of the classification can be adversely affected. In order to mitigate the effect of this threat we have included a feature to download from an online repository where all other users contribute such knowledge.

Also one of the subjects used for the validation is a close acquaintance of the researcher. Even though subjects were made to understand that feedback could help improve the work. This relationship might still affect the results of the validation.

Also, all subject involved in the validation were either researchers or student. This may have affected the style of programming and thus the separation of concerns in the class and then in effect the results of the metrics' classification. Since the objective is to be able to generalize the results to even code in commercial software developed by professional software engineers this might be a validity threat. We aim to solve this in future research by including more professional developers.

2) External Validity: Only 4 systems of relatively small size were used in the validation as such no statistical analysis to significance of the results could be employed. However in order to reduce this threat and increase the generalizability of the approach, we tried to analyse projects from different domains and those implemented for different software platforms. In future research we plan to reduce this threat further by including more projects in the validation.

The approach has only been implemented for the java programming language in this study even though it has been designed to generalizable to other object oriented programming languages. Until it has been implemented in other languages, claims of the approachs generalizability to other programming maybe subject to validity threat.

3) Construct Validity: During validation we asked the user to select one metric that was best for each class, this resulted in some data lose as we are not able to tell how much worse the others were. Thus accuracy predictions were affected by how many times a metric is chosen as best. In future work we plan to reduce this threat by collecting agreement data for each metric for all classes and then deciding which is best from it.

IX. Conclusion and Future Work

Current static reverse engineering techniques produce huge, complete class diagrams that tend to focus on system components and the relationship between them. Even though this is useful in some contexts, researcher realised that developers find the use of the diagram limited for program comprehension. One reason for this is because they don't provide other perspectives of system that are vital when trying to comprehension unknown software systems. According to researchers, one such perspective that frequently comes up during maintenance and extension tasks is the mapping of system concerns to the software components that implement them.

In this work, we present a framework and tool(Concern Detector) for automatically detecting these common concerns that are found in modern software systems (E.g. User interfaces, databases, concurrency, security & authentication) and map them to software components that implement them. The goal is to provide a concern perspective of the system on top of reverse engineered class diagrams in order to increase the diagram's usefulness for program comprehension. To do this, we proposed 3 metrics for automatically identifying these concerns in software systems. Our assessments show that our metrics were effective at this task. Our taxonomy of concerns was also found to be useful for describing the overall functionality of the system in our case study as well as the responsibilities of each class in the system. The Concern Detector tool based on light weight static program analysis is efficient and useable whiles still giving good results.

The accuracies however of the 3 metrics were affected by a number of factors, which we believe when improved in future work, could increase the performance of the metrics and in effect the result produced by the Concern Detector tool as well. The Text metric's performance was affected by the small number of descriptions of concerns that were used to generate the word list to describe each concern. This could be improved by analysing more descriptions to improve the word list. The LIB metric's performance is dependent on knowing the libraries and object types used in a class. Therefore, it suffers when such knowledge is not available. A way to improve this is it to keep an online database of library and object types with associated concerns, which would be updated each time a user labels an object or library to be of a certain concern type.

Even though the framework is designed to be general, our current implementation is designed for the Java programming language. Therefore, the performance of the framework for systems implemented in other programming languages is still open. We also believe that, for a user to benefit from the concern perspective that the tool provides, it is important that they have previous knowledge of the domain and purpose of the system being studied. If this precondition is met, then the concern perspective can provide more insight into the internal mapping of the system components to the concerns that they implement.

One other area that could be improved in future work is the visualization of the concerns found on the class diagram. Currently, the tool labels each class with the concern with the biggest weight and then lists other concerns found along with their weight on top of each class, in order to give the user an overall picture of the responsibilities of the class. In future work we would like the user to have the possibility to show only classes that implement a certain concern, or to group classes into packages based on the concerns that they implement.

ACKNOWLEDGMENT

I would like to thank my supervisors Dr. Michel Chaudron, and Mr. Truong Ho Quang for the wonderful support and guidance throughout the duration of this work. I would also like to thank Mr. Grischa Liebel and Mr. Einar Sundgren for being kind enough to share their projects that are used in the validation and for the time they took out of their busy schedules to give us feedback.

REFERENCES

[1] K. Mens and T. Tourwé, "Delving source code with formal concept analysis," *Comput. Lang. Syst. Struct.*, vol. 31, no. 3-4, pp. 183–197, Oct. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2004.11.004

[2] B. Boehm, "Software engineering," *IEEE Transactions on Computers*, vol. 25, no. 12, pp. 1226–1241, 1976.

[3] M. M. Lehman and L. A. Belady, Eds., *Program Evolution: Processes of Software Change*. San Diego, CA, USA: Academic Press Professional, Inc., 1985.

[4] A. Dunsmore, M. Roper, and M. Wood, "The role of comprehension in software inspection," *Journal of Systems and Software*, vol. 52, no. 23, pp. 121 – 129, 2000.

[5] H. B. O. H. Mohd, M. Chaudron, and P. v. d. Putten, "Interactive scalable abstraction of reverse engineered uml class diagrams," 2014.

[6] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, Jan. 2001.

[7] T. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," New York, NY, USA, 2006//, pp. 492 – 501, developer work habits;design documents;code duplication;code snippets;code ownership;agile software development;.

[8] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 255–265.

[9] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *Software Engineering, IEEE Transactions on*, vol. 32, no. 12, pp. 971–987, Dec 2006.

[10] P. Caserta and O. Zendra, "Visualization of the static aspects of software: a survey," *IEEE transactions on visualization and computer graphics*, vol. 17, no. 7, pp. 913–933, 2011.

[11] H. B. O. H. Mohd, M. Chaudron, and P. v. d. Putten, "Interactive scalable abstraction of reverse engineered uml class diagrams," 2014.

[12] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, pp. 3:1–3:37, Dec. 2007. [Online]. Available: http://doi.acm.org/10.1145/1314493.1314496

[13] A. Mesbah and A. van Deursen, "Crosscutting concerns in j2ee applications," in *Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, ser. WSE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 14–21. [Online]. Available: http://dx.doi.org/10.1109/WSE.2005.4

[14] M. L. Collard, J. I. Maletic, and A. Marcus, "Supporting document and data views of source code," in *Proceedings of the 2002 ACM Symposium on Document Engineering*, ser. DocEng '02. New York, NY, USA: ACM, 2002, pp. 34–41.

[15] G. A. Miller, "Wordnet: A lexical database for english," *COMMUNICATIONS OF THE ACM*, vol. 38, pp. 39–41, 1995.

[16] N. Dragan, "Emergent laws of method and class stereotypes in object oriented software," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, Sept 2011, pp. 550–555.

[17] A. Budi, D. L. Lucia, L. Jiang, and S. Wang, "Automated detection of likely design flaws in layered architectures," *Research Collection School of Information Systems*, 2011.

[18] R. J. Wirfs-Brock, "Characterizing classes," *IEEE Software*, vol. 23, no. 2, pp. 9–11, 2006.

[19] A. Zaidman, "Automatic identification of key classes in a software system using webmining techniques," in *Journal of Software Maintenance & Evolution, 20(6):387417. (Cited on*. IEEE Computer Society Press, 2008, p. 158.

[20] E. R. Babbie, *The practice of social research*. Wadsworth Thomson Learning, 2007.

X. APPENDIX

A. Example of concern classification of class shown to authors during validation

TABLE III. LIB METRIC CONCERN LIBRARY

Library	Concern
java.applet	User Interface
java.awt	User Interface
java.awt.color	User Interface
java.awt.datatransfer	User Interface
java.awt.dnd	User Interface
java.awt.event	Event/Event Handling
java.awt.font	User Interface
java.awt.geom	User Interface
java.awt.im	User Interface
java.awt.im.spi	User Interface
java.awt.image	User Interface
java.awt.image.renderable	User Interface
java.awt.print	User Interface
android.view.LayoutInflater	User Interface
android.view.View	User Interface
android.view.ViewGroup	User Interface
android.widget.BaseAdapter	User Interface
android.widget.TextView	User Interface
java.beans	Object Model
java.beans.beancontext	Object Model
java.lang.annotation	Program Logic
java.lang.reflect	Program Logic
java.net	Network/Web
org.apache.http.client.HttpClient	Network/Web
android.net.ParseException	Network/Web
org.apache.http.client.entity.UrlEncodedFormEntity	Network/Web
org.apache.http.client.methods.HttpPost	Network/Web
org.apache.http.impl.client.DefaultHttpClient	Network/Web
org.apache.http.params.BasicHttpParams	Network/Web
org.apache.http.params.HttpParams	Network/Web
org.apache.http.params.HttpProtocolParams	Network/Web
org.apache.http.protocol.HTTP	Network/Web
org.apache.http.util.EntityUtils	Network/Web
org.apache.http.HttpResponse	Network/Web
org.apache.http.HttpVersion	Network/Web
org.json.JSONArray	Parser/Interpreter
org.json.JSONException	Parser/Interpreter
org.json.JSONObject	Parser/Interpreter
java.rmi	Third Party System Call
java.rmi.activation	Third Party System Call
java.rmi.dgc	Third Party System Call
java.rmi.registry	Third Party System Call
java.rmi.server	Third Party System Call
java.security	Security/Validator
java.security.acl	Security/Validator
java.security.cert	Security/Validator
java.security.interfaces	Security/Validator
java.security.spec	Security/Validator
java.sql	Persistence
java.text	Data Manupulator
java.text.spi	Data Manupulator
java.time	Time/Date/Zone
java.time.chrono	Time/Date/Zone
java.time.format	Time/Date/Zone
java.time.temporal	Time/Date/Zone
java.time.zone	Time/Date/Zone
java.util.Date	Time/Date/Zone
java.util.Calendar	Time/Date/Zone
java.util.GregorianCalendar	Time/Date/Zone
java.util.concurrent	Concurrency
java.util.concurrent.atomic	Concurrency
java.util.concurrent.locks	Concurrency
java.util.function	Logic
java.util.jar	application packaging
java.util.regex	Logic
java.util.stream	Data Manupulator
java.util.zip	application packaging
javax.activation	Security/Validator
javax.activity	Exception Handling
javax.annotation	Program Logic
javax.annotation.processing	Program Logic
javax.crypto	Security/Validator
javax.crypto.interfaces	Security/Validator
javax.crypto.spec	Security/Validator
javax.imageio	User Interface
javax.imageio.event	User Interface

TABLE IV. LIB METRIC CONCERN LIBRARY

Library	Concern
javax.imageio.metadata	User Interface
javax.imageio.plugins.bmp	User Interface
javax.imageio.plugins.jpeg	User Interface
javax.imageio.spi	User Interface
javax.imageio.stream	User Interface
javax.jws	Network/Web
javax.jws.soap	Network/Web
javax.lang.model	Object Model
javax.lang.model.element	Object Model
javax.lang.model.type	Object Model
javax.lang.model.util	Object Model
javax.management.remote	Network/Web
javax.management.remote.rmi	Network/Web
javax.management.timer	Event/Event Handling
javax.net	Network/Web
javax.net.ssl	Network/Web
javax.print.attribute	I/O
FileWriter	I/O
FileInputStream	I/O
FileChannel	I/O
javax.print.attribute.standard	I/O
javax.print.event	I/O
javax.rmi	Network/Web
javax.rmi.CORBA	Network/Web
javax.rmi.ssl	Network/Web
javax.script	Parser/Interpreter
javax.security.auth	Security/Validator
javax.security.auth.callback	Security/Validator
javax.security.auth.kerberos	Security/Validator
javax.security.auth.login	Security/Validator
javax.security.auth.spi	Security/Validator
javax.security.auth.x500	Security/Validator
javax.security.cert	Security/Validator
javax.security.sasl	Security/Validator
javax.sound.midi	User Interface
javax.sound.midi.spi	User Interface
javax.sound.sampled	User Interface
javax.sound.sampled.spi	User Interface
javax.sql	Persistence
javax.sql.rowset	Persistence
javax.sql.rowset.serial	Persistence
javax.sql.rowset.spi	Persistence
javax.swing	User Interface
javax.swing.border	User Interface
javax.swing.colorchooser	User Interface
javax.swing.event	User Interface
javax.swing.filechooser	User Interface
javax.swing.plaf	User Interface
javax.swing.plaf.basic	User Interface
javax.swing.plaf.metal	User Interface
javax.swing.plaf.multi	User Interface
javax.swing.plaf.nimbus	User Interface
javax.swing.plaf.synth	User Interface
javax.swing.table	User Interface
javax.swing.text	User Interface
javax.swing.text.html	User Interface
javax.swing.text.html.parser	User Interface
javax.swing.text.rtf	User Interface
javax.swing.tree	User Interface
javax.swing.undo	User Interface
android.graphics.Canvas	User Interface
android.graphics.Color	User Interface
android.graphics.Paint	User Interface

TABLE V. LIB METRIC CONCERN LIBRARY

Library	Concern
android.graphics.Path	User Interface
android.graphics.Point	User Interface
com.google.android.maps.GeoPoint	User Interface
com.google.android.maps.MapController	Geo Location
com.google.android.maps.MapView	Geo Location
com.google.android.maps.Overlay	Geo Location
com.google.android.maps.Projection	Geo Location
javax.transaction	Transaction Handling
javax.transaction.xa	Transaction Handling
javax.xml	Parser/Interpreter
javax.xml.bind	Parser/Interpreter
javax.xml.bind.annotation	Parser/Interpreter
javax.xml.bind.annotation.adapters	Parser/Interpreter
javax.xml.bind.attachment	Parser/Interpreter
javax.xml.bind.helpers	Parser/Interpreter
javax.xml.bind.util	Parser/Interpreter
javax.xml.crypto	Parser/Interpreter
javax.xml.crypto.dom	Parser/Interpreter
javax.xml.crypto.dsig	Parser/Interpreter
javax.xml.crypto.dsig.dom	Parser/Interpreter
javax.xml.crypto.dsig.keyinfo	Parser/Interpreter
javax.xml.crypto.dsig.spec	Parser/Interpreter
javax.xml.datatype	Parser/Interpreter
javax.xml.namespace	Parser/Interpreter
javax.xml.parsers	Parser/Interpreter
javax.xml.soap	Parser/Interpreter
javax.xml.stream	Parser/Interpreter
javax.xml.stream.events	Parser/Interpreter
javax.xml.stream.util	Parser/Interpreter
javax.xml.transform	Parser/Interpreter
javax.xml.transform.dom	Parser/Interpreter
javax.xml.transform.sax	Parser/Interpreter
javax.xml.transform.stax	Parser/Interpreter
javax.xml.transform.stream	Parser/Interpreter
javax.xml.validation	Parser/Interpreter
javax.xml.ws	Parser/Interpreter
javax.xml.ws.handler	Parser/Interpreter
javax.xml.ws.handler.soap	Parser/Interpreter
javax.xml.ws.http	Parser/Interpreter
javax.xml.ws.soap	Parser/Interpreter
javax.xml.ws.spi	Parser/Interpreter
javax.xml.ws.spi.http	Parser/Interpreter
javax.xml.ws.wsaddressing	Parser/Interpreter
javax.xml.xpath	Parser/Interpreter
javax.validation	Security/Validator
Color	User Interface
URL	Network/Web
URI	I/O
JEditorPane	User Interface
JMenuBar	User Interface
JPanel	User Interface
JToolBar	User Interface
JMenuItem	User Interface
JTextArea	User Interface
Separator	User Interface
JToggleButton	User Interface
JScrollPane	User Interface
JButton	User Interface
JLabel	User Interface
JCheckBox	User Interface
GroupLayout	User Interface
Dimension	User Interface
JComboBox	User Interface
JTextField	User Interface
BufferedImage	User Interface
Graphics2D	User Interface
QuadCurve2D	User Interface
File	Persistence
ImageIcon	User Interface
Image	User Interface
JFileChooser	User Interface

TABLE VI. LIB METRIC CONCERN LIBRARY

Library	Concern
ObjectOutputStream	I/O
JSplitPane	User Interface
LayoutInflater	User Interface
EventListener	Event/Event Handling
Formattable	I/O
Observer	Event/Event Handling
Base64	Parser/Interpreter
Base64.Decoder	Parser/Interpreter
Base64.Encoder	Parser/Interpreter
EventListenerProxy	Event/Event Handling
EventObject	Event/Event Handling
Observable	Event/Event Handling
PropertyPermission	Security/Validator
Timer	Event/Event Handling
TimerTask	Event/Event Handling
JSONObject	Parser/Interpreter
HttpResponse	Network/Web
JSONArray	Parser/Interpreter
HttpClient	Network/Web
HttpParams	Network/Web
HttpPost	Network/Web
UrlEncodedFormEntity	Network/Web
android.content.Intent	Messaging
android.os.Environment	I/O
Intent	Messaging
GeoPoint	User Interface
Projection	User Interface
Path	User Interface
Paint	User Interface
Point	User Interface
android.location.GpsStatus	Geo Location
android.location.Location	Geo Location
android.location.LocationListener	Geo Location
android.location.LocationManager	Geo Location
android.media.MediaPlayer	User Interface
android.os.Bundle	Messaging
LocationManager	Geo Location
LocationListener	Geo Location
Location	Geo Location
Notification	User Interface
PendingIntent	Messaging
NotificationManager	Messaging
MediaPlayer	User Interface
android.widget.Button	User Interface
android.widget.EditText	User Interface
android.widget.Toast	User Interface
android.content.BroadcastReceiver	Messaging
android.content.DialogInterface	User Interface
android.content.IntentFilter	Messaging
RegisterResponseReceiver	Messaging
EditText	User Interface
Button	User Interface
TextView	User Interface
IntentFilter	User Interface
com.google.android.gms.maps. model.LatLng	Object Model
com.google.android.gms.maps. model.LatLngBounds	Object Model
com.google.android.gms.maps. model.PolygonOptions	Object Model
com.google.android.gms.maps. CameraUpdate	Geo Location
com.google.android.gms.maps. CameraUpdate-Factory	Geo Location

TABLE VII. LIB METRIC CONCERN LIBRARY

Library	Concern
com.google.android.gms.maps. GoogleMap	Geo Location
com.google.android.gms.maps. GoogleMapOptions	Geo Location
com.google.android.gms.maps. SupportMapFragment	Geo Location
com.google.android.gms.common. GooglePlayServicesUtil	Transaction Handling
com.google.android.gms.common. ConnectionResult	Network/Web
GoogleMap	Geo Location
Bundle	Messaging
GoogleMapOptions	Geo Location
FragmentManager	User Interface
Fragment	User Interface
SupportMapFragment	User Interface
PolygonOptions	Object Model
Builder	User Interface
DisplayMetrics	User Interface
Display	User Interface
CameraUpdate	Geo Location
AlertDialog	User Interface
android.app.AlertDialog	User Interface
android.widget.ArrayAdapter	User Interface
android.widget.Spinner	User Interface
Spinner	User Interface
RecordResponseReceiver	Network/Web
android.app.AlertDialog	User Interface
android.app.ListActivity	User Interface
android.view.ContextMenu	User Interface
android.view.ContextMenu. ContextMenuInfo	User Interface
android.view.MenuInflater	User Interface
android.view.MenuItem	User Interface
android.widget.AdapterView. AdapterContextMenuInfo	User Interface
MenuInflater	User Interface
AdapterContextMenuInfo	User Interface
android.app.TabActivity	User Interface
android.content.res.Resources	I/O
android.widget.TabHost	User Interface
android.util.DisplayMetrics	User Interface
android.view.Menu	User Interface
Resources	I/O
TabHost	User Interface
TabSpec	User Interface
android.os.AsyncTask	Concurrency
android.os.Message	Messaging
Message	Messaging
android.support.v4.app.Fragment	User Interface
android.support.v4.app.FragmentActivity	User Interface
android.support.v4.app. FragmentManager	User Interface
android.content.SharedPreferences	Messaging
android.preference.PreferenceManager	Persistence
Editor	User Interface
SharedPreferences	Persistence
android.app.DatePickerDialog	User Interface
android.app.Dialog	User Interface
android.widget.DatePicker	User Interface
CheckBox	User Interface
android.app.NotificationManager	User Interface
android.app.PendingIntent	Messaging
android.support.v4.app. NotificationCompat	User Interface
android.app.Notification	User Interface
android.widget.CheckBox	User Interface
JSeparator	User Interface
JMenu	User Interface
JTextPane	User Interface
JPopupMenu	User Interface
Font	User Interface
Polygon	User Interface
java.util.Observer	Event/Event Handling
java.util.Observable	Event/Event Handling

TABLE VIII. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Concurrency	
Word	Weight In Concern
lock	176
thread	99
executor	55
queue	52
task	44
concurrent	41
blocking	36
acquire	31
write	30
pool	29
time	28
interrupt	28
action	27
read	25
condition	25
service	24
join	23
map	23
fork	22
future	22
return	21
operation	21
happen	19
wait	19
current	19
synchronization	18
synchronize	17
reentrant	17
object	16
list	16

TABLE IX. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Event/Event Handling	
Word	Weight In Concern
listener	178
event	171
basic	117
ui	99
action	56
change	56
pane	53
tree	44
mouse	39
ui	38
editor	38
accessible	37
synth	34
naming	33
context	31
adapter	29
menu	27
receive	27
frame	24
combo	23
text	23
list	22
focus	22
awt	22
object	21
bar	20
property	20
box	20
kit	19
input	18

TABLE X. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Exception Handling	
Word	Weight In Concern
cause	61
message	55
detail	34
get	20
stack	18
construct	18
trace	18
throwable	16
specify	14
null	13
constructor	12
(which	11
new	10
public	10
retrieval	8
late	8
enable	8
save	8
since	7
print	7
unknown	7
parameter	7
value	7
object	6
writable	6
unsupported	6
suppression	6
wait	6
format	6
disabled	5

TABLE XI. FREQUENT WORDS USED TO DESCRIBE CONCERNS

I/O	
Word	Weight In Concern
file	224
stream	188
print	105
input	105
object	75
doc	61
byte	60
datum	56
read	55
writer	49
output	47
service	43
flavor	43
character	36
descriptor	28
write	26
job	26
skip	21
specify	19
buffer	19
array	19
return	18
attribute	17
exist	17
create	17
public	17
close	16
open	16
reading	16
reader	15

TABLE XII. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Messaging	
Word	Weight In Concern
message	54
jms	42
messaging	18
queue	18
service	15
web	14
provider	12
receive	10
model	9
client	8
server	7
subscriber	7
jump	6
publish	6
open	6
middleware	6
software	6
topic	6
create	5
process	5
register	4
management	4
bean	4
orient	4
publisher	4
send	4
community	3
scheme	3
jsr	3
term	3

TABLE XIII. FREQUENT WORDS USED TO DESCRIBE CONCERNS

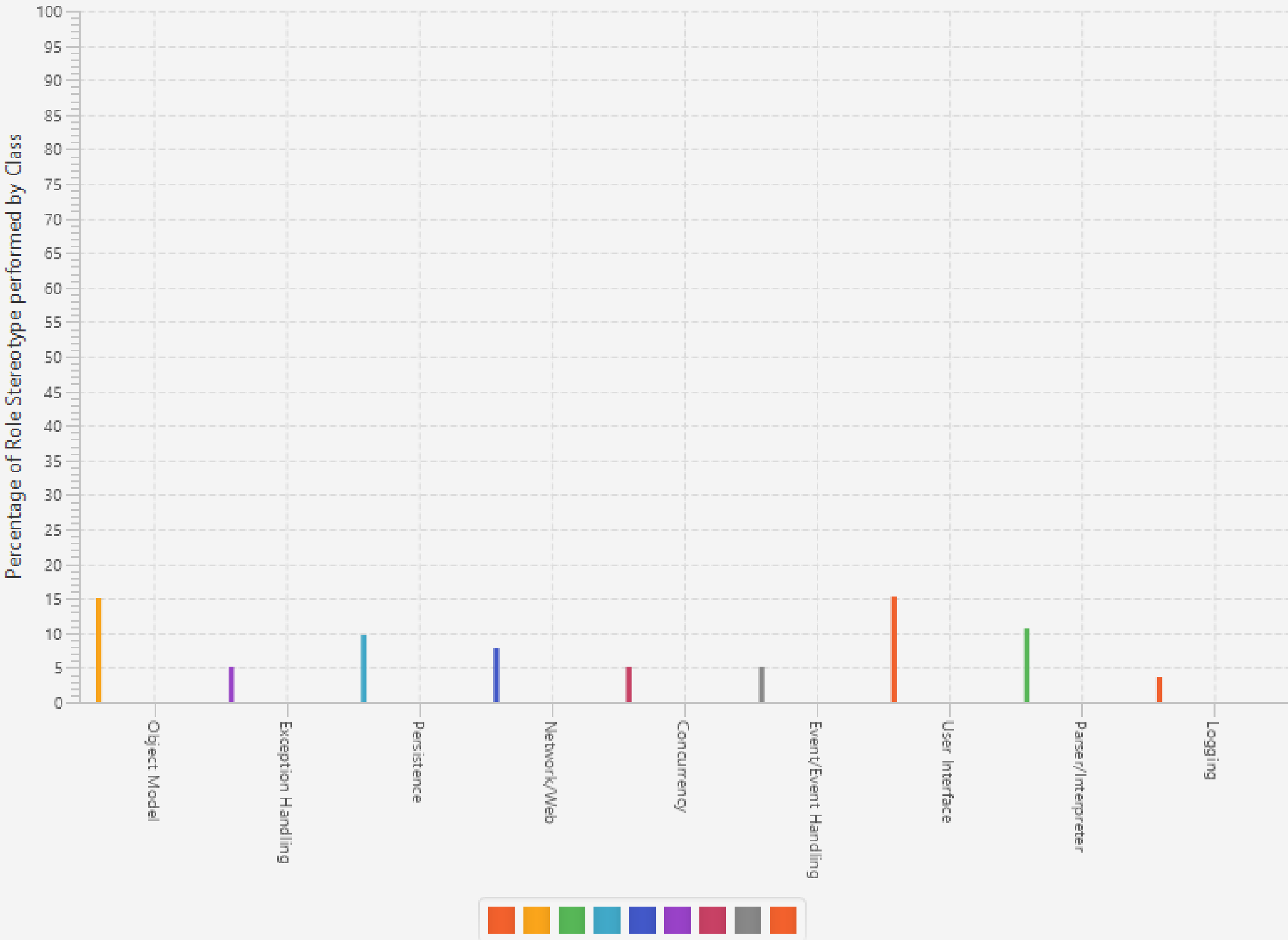
Network/Web	
Word	Weight In Concern
http	198
protocol	106
request	105
parameter	102
response	91
url	75
client	70
socket	64
connection	62
address	55
connector	55
server	52
jmx	52
execute	49
params	48
deprecate	47
set	46
network	46
context	45
value	43
action	40
entity	39
content	38
get	35
input	31
core	31
names	29
return	29
static	28
form	28

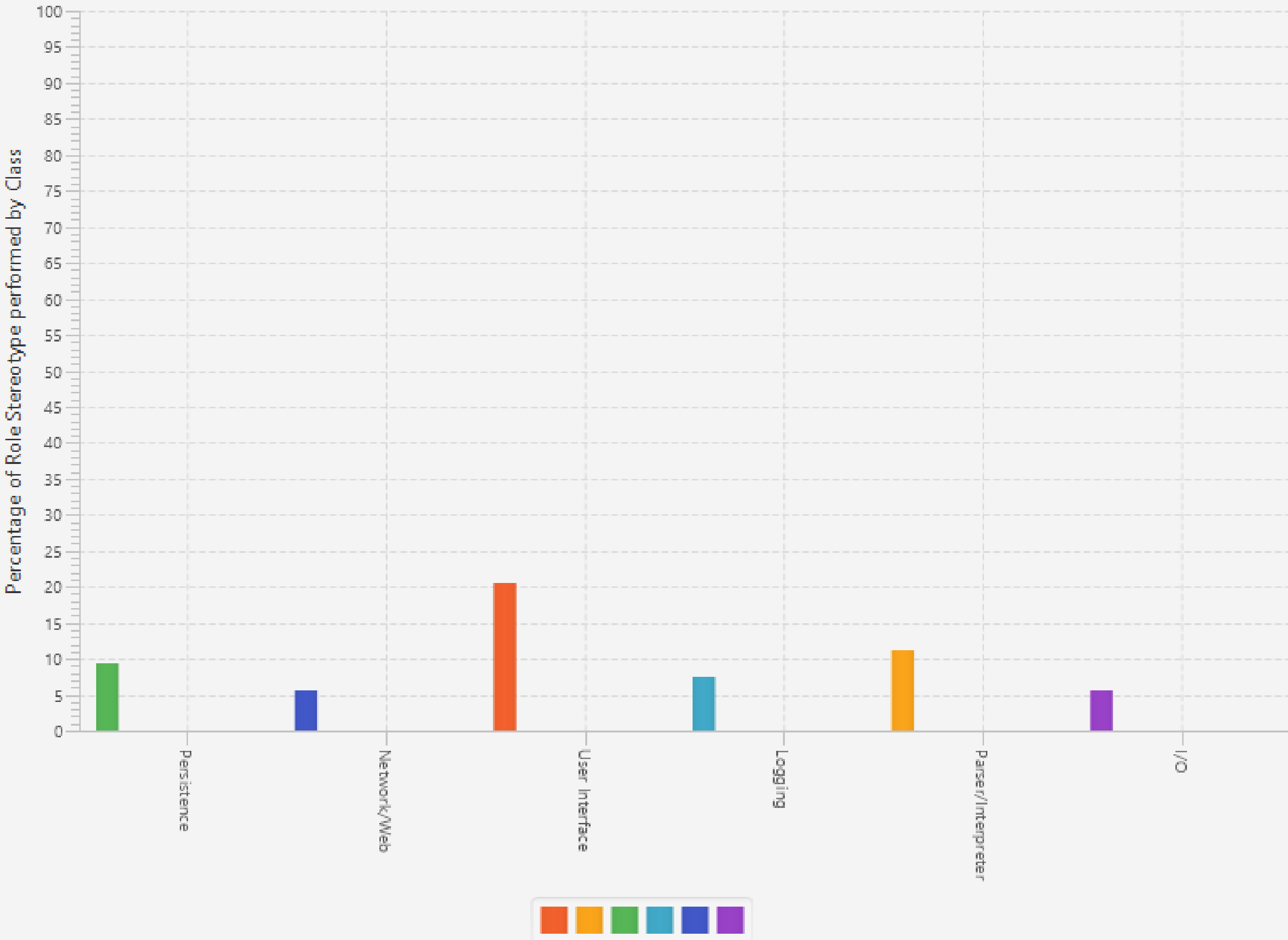
TABLE XIV. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Object Model	
Word	Weight In Concern
set	1
get	1

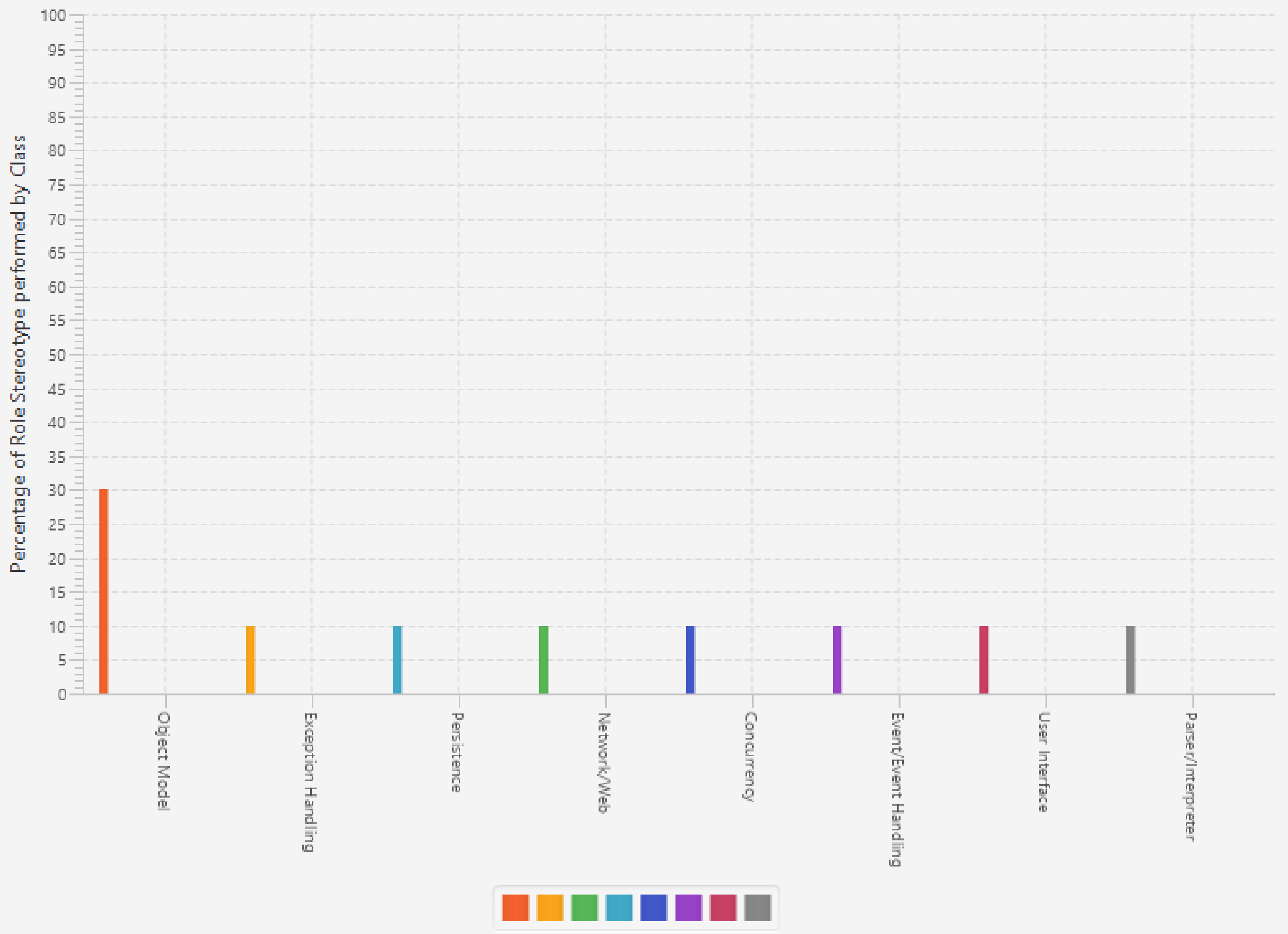
TABLE XV. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Parser/Interpreter	
Word	Weight In Concern
json	93
byte	65
object	59
script	58
array	39
model	35
engine	34
encode	33
stream	30
parser	28
base	25
output	24
datum	20
encoding	19
streaming	18
processing	18
result	15
create	15
factory	15
return	14
event	14
public	14
value	14
write	13
buffer	13
scheme	12
specify	12
encoder	12
code	12
description	11


PaintUML(Combined Stereotype)
Percentage of Role Stereotype performed by Class
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
Object Model
Exception Handling
Persistence
Network/Web
Concurrency
Event/Event Handling
User Interface
Parser/Interpreter
Logging



PaintUML(Package Import Stereotype)
Percentage of Role Stereotype performed by Class
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
Persistence
Network/Web
User Interface
Logging
Parser/Interpreter
I/O


PaintUML(Semantic Stereotype)
Percentage of Role Stereotype performed by Class
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
Object Model
Exception Handling
Persistence
Network/Web
Concurrency
Event/Event Handling
User Interface
Parser/Interpreter

Demographic Information

Project Name:	
Are you the author of this project(Yes/No):	
How would you rate your expertise of using UML models?	☐ 0 years
	☐ 1 - 2 Years
	☐ 3 - 4 Years
	☐ 5 - 6 Years
	☐ 7 - 8 Years
	☐ more than 8

Which Stereotype Classifier best describes what this class does?

Class name:	☐ Import Stereotype
	☐ Semantic Stereotype
	☐ Combined Stereotype
	☐ None

Class name:	☐ Import Stereotype

Fig. 17. Questionnaire Sample

	☐ Combined Stereotype
	☐ None

Class name:	☐ Import Stereotype
	☐ Semantic Stereotype
	☐ Combined Stereotype
	☐ None

Class name:	☐ Import Stereotype
	☐ Semantic Stereotype
	☐ Combined Stereotype
	☐ None

Class name:	☐ Import Stereotype
	☐ Semantic Stereotype
	☐ Combined Stereotype
	☐ None

Class name:	☐ Import Stereotype
	☐ Semantic Stereotype
	☐ Combined Stereotype
	☐ None

Fig. 18. Questionnaire Sample

TABLE XVI. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Persistence	
Word	Weight In Concern
set	895
parameter	776
sql	560
database	534
object	433
datum	357
value	299
row	276
stream	259
driver	239
void	206
designate	172
given	163
reader	148
access	148
index	144
get	138
connection	137
jdbc	132
statement	122
update	108
input	103
source	103
model	102
character	98
doe	90
command	90
call	82
length	77
object's	71

TABLE XVII. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Security/Validator	
Word	Weight In Concern
key	91
provider	43
constraint	33
datum	32
parameter	30
object	29
permission	26
description	25
algorithm	24
context	22
certificate	22
store	22
security	21
service	19
cipher	18
spi	18
access	18
stream	16
spi	16
message	15
entry	15
validator	15
file	14
configuration	14
generator	14
factory	14
digest	13
code	13
cryptographic	13
private	12

TABLE XVIII. FREQUENT WORDS USED TO DESCRIBE CONCERNS

Transaction Handling	
Word	Weight In Concern
transaction	48
manager	10
transactional	8
annotation	7
rollback	6
heuristic	6
description	6
manage	6
bean	6
commit	5
roll	5
update	5
back	5
server	5
datum	5
information	4
operation	4
request	4
make	4
decision	3
party	3
multiple	3
mark	3
context	3
status	3
scoped	3
report	3
relevant	3
boundary	3
synchronization	3

TABLE XIX. FREQUENT WORDS USED TO DESCRIBE CONCERNS

User Interface	
Word	Weight In Concern
view	285
layout	143
menu	103
listener	99
list	88
event	88
text	76
adapter	70
object	64
callback	63
window	58
item	58
display	56
model	56
color	49
change	49
control	49
button	47
line	47
manager	46
user	45
file	42
datum	42
filter	42
invoke	41
action	40
focus	38
container	38
scroll	38
definition	37