Measuring Software Ticket Quality using Statistical Analysis

Andrei-Mihai Nicolae

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Masters project proposal

December 18th 2017

Contents

1 Introduction

1.1 Background

Software engineering, compared to other engineering fields, such as mechanical or electrical engineering, is much more abstract - while performing maintenance on a car engine or on an electric panel, you can actually touch the components, they are physical. However, in computer science and the engineering of software, you cannot see the whole picture. Of course one can actually open up a laptop or PC and see and touch the components, but the software, what is actually displayed to the user, is a multitude of layers on top of each other: it starts with a number of characters in some file; then, that file gets compiled or interpreted to machine code by other components which run on the operating system; then, these components can communicate with the kernel (i.e. the component that sits exactly between the OS and the hardware) and eventually perform the final step - send different 1s and 0s encoded into electrical signals to the CPU and various other hardware components.

As previously mentioned, there are many levels of abstraction that computers expose to their human operators, thus developing software that runs and performs as expected in agreement with its requirements is not trivial. Therefore, developers have created software called *issue tracking systems* that can manage and track *tickets* of various forms such as bug reports, feature requests etc. The aforementioned tickets can vary greatly and issue tracking systems usually allow developers to include many types of information, such as:

- stack traces;
- steps to reproduce a bug;
- summary and description;
- story points (e.g. how many units of time does it take one developer to mark the ticket as resolved or completed).

In order to create such tickets, the end users or the developers themselves notice certain flaws in the software or propose new features to be implemented. Then, they head to the issue tracking system and create tickets containing the necessary information. However, we are facing a main issue here, and that is understanding the key elements the developers look for when fixing a bug or implementing a new feature.

Therefore, a software engineering issue still not completely resolved is how can one create valuable, high quality tickets.

1.2 Problem Statement

This research is investigating the features of a high quality ticket. More specifically, it is looking into the various components of a ticket (e.g. summary, description, comments) and

their characteristics in order to infer what combination of such components or what specific attributes help developers close the ticket quicker (i.e. time between ticket creation and marked as CLOSED or RESOLVED).

We believe that this study is valuable to the software engineering community because solving the issue/ticket quality problem will help reduce both cost and effort in companies worldwide. Having a clear understanding of what attributes make for a high quality ticket will lead the project maintainers to write specific guidelines in the ticket creation form (maybe even create an automated system) that will help reporters include the most important details and structure them accordingly (e.g. for a specific type of project it might be essential to include stack traces so that developers can locate the bugs quicker), which will in turn reduce overall costs and developers/triagers' effort.

Furthermore, our study might also increase the likelihood of people reporting bugs or asking for more specific features more often. Not having specific guidelines or an automated system when creating a ticket many non-technical users might be put off and give up creating the report. However, if the company would put up a form where all the information needed can be very easily inserted, users might be more inclined to report bugs and create feature requests, thus increasing the software's overall quality on the long term.

1.3 Research Questions and Contributions

The main research question this study is trying to answer is what makes for a valuable, high quality ticket?

However, as this is a broad research question, we have divided it into three sub-research questions which, when combined, would yield the answer to our main research question:

- are there components which singularly affect the quality of a ticket (e.g. well written summary) or only combinations of them (e.g. stack traces together with steps to reproduce)?
- how much do specific components or combinations of them affect the quality of a ticket?
- can quality be inferred solely from the ticket or do we also need to consider the surrounding context (e.g. details related to the version control system)?

Even though there have been many studies related to issue/ticket quality, most of them have looked at it qualitatively, not quantitatively (e.g. Bettenburg et al. [2008a], Bettenburg et al. [2007]). This is the first contribution of our study - we will explore the area of ticket quality from a statistical point of view, not through interviews or other human-centered data gathering (i.e. we want to inspect several open source repositories and collect tickets through the issue tracking system's APIs).

Moreover, another contribution is that, even though most studies so far have looked solely at ticket features/components, we want to inspect both tickets and other components of the software project, such as version control system and how various attributes affect the time-to-close for a particular ticket.

The third contribution is a proposed model for bug reports and/or feature requests. After collecting data and finding which components together with their afferent attributes make for a high quality ticket, we will propose a model for best practices in terms of creating tickets.

1.4 Proposal Structure

The rest of this project proposal is structured as follows:

- Section 2 presents both the study design and the background survey, which in turn is divided into three subsection:
  - subsection 2.2 describes the literature review on data quality and metrics;
  - subsection 2.3 describes the background survey on issue/ticket quality;
  - subsection 2.4 elicits literature on waste in software projects;
- Section 3 describes how we would like to conduct the research;
- Section 4 shows how we broke down the work so that we keep a consistent pace throughout the lifecycle of the research.

2 Background Survey

2.1 Study Design

In order to conduct a thorough background survey, several topics needed to be explored, among which the most important were:

- data quality and metrics;
- issue/ticket quality;
- waste in software projects;
- sentiment analysis.

Firstly, we will deal with a large volume of tickets and information extracted from software projects and the tools that revolve around them, such as issue trackers and version control systems. Thus, *data quality* is an important topic that needed to be explored in order to infer appropriate conclusions.

We analyzed data quality mainly in software organizations and looked at what metrics are most useful when performing measurements. We also found different methodologies that either measure data quality or calculate the impact of poor data quality on the software project.

Secondly, probably the topic of most interest to our research is *issue quality*. This field is mainly concerned with software tickets and their characteristics: how important are issue trackers to a software project, what makes for a good bug report, what impact does bad triaging have etc. As we are interested in finding out what features are most important in improving the quality of a ticket, the issue quality literature reviewed proved to be beneficial and provided valuable insight into how other researchers tried to approach this problem.

The third topic was waste and its impact on software projects. We needed to look into various communication and tool-related waste in software projects because having a project with a considerable amount of waste might affect our results and compromise the evaluation process. We investigated researches that looked into what the main types of waste are and how it can be avoided so that we could apply the findings into selecting the projects.

Last but not least, we did a literature review on sentiment analysis as well because we will inspect, among other components of a ticket, comments - we want to analyze the textual representation of comments and see if the overall sentiment influences the time it takes to close a ticket (i.e. a conversation might unnecessarily delay solving the ticket).

The main sources of information when searching for papers were Google Scholar, ACM digital library and IEEE Xplore digital library. The first step in the literature review process was finding relevant papers that could aid us in our research and splitting them into the categories mentioned above. The key search phrases used (with slight variations) are as follows:

- software data quality;
- software data quality metrics;
- data quality in software projects;
- issue quality in software projects;
- ticket quality software;
- sentiment analysis;
- software project communication waste.

The snowballing research technique was also applied when retrieving papers for the background review. Citation count and publication journal were two important aspects when selecting papers - the most cited papers, published in the top quality journals were reviewed.

2.2 Data Quality and Metrics

Several authors have investigated the *meaning* of data quality and what characteristics define it.

Bachmann and Bernstein [2009] conducted a thorough investigation of several software projects, both open source (5 projects) as well as closed source (1 project), in order to infer what determines their quality. They selected various sources of information, among which bug tracking databases and version control systems logs, and examined SVN logs, CVS logs and the content of the bug tracker databases, in the end trying to link the logs with the bug tracker contents as they are not integrated by default. A different approach was taken by Strong et al. [1997] who conducted a qualitative analysis instead of quantitative by collecting data from 42 data quality projects by interviewing custodians, customers and managers. They analyzed each project using the data quality dimensions as content analysis codes.

The first study came to several conclusions, among which:

- closed source software projects usually exhibit better data quality (i.e. better average number of attachments, better average status changes, better average number of commits per bug report);
- reduced traceability between bug reports and version control logs due to scarce linkage between the two;
- open source projects exhibit reduced quality in change logs as, for example, the Eclipse project has over 20% empty commit messages.

However, the second study reached the conclusion that representational data quality dimensions are underlying causes of accessibility data quality problem patterns. The authors also found out that three underlying causes for users' complaints regarding data not supporting their tasks are incomplete data, inadequately defined or measured data and data that could not be appropriately aggregated.

The work of Kitchenham and Pfleeger [1996] looks specifically at defining quality in a software project, as well as finding who the people in charge of this are and how they should approach achieving it. They tried to define quality in software projects and analyze techniques that measure such metrics by looking at other models proposed in different researches, such as McCall's quality model or ISO 9126. However, they learned that quality is very hard to define and there are various factors which need to be taken into consideration, such as the business model of the company, the type of the software project (e.g.

safety critical, financial sector) or the actors who are involved and how they coordinate the software activities.

One other piece of data that is probably the most crucial in a software project is code, and Stamelos et al. [2002] tried to discuss and examine the quality of the source code delivered by open source projects. They used a set of tools that could automatically inspect various aspects of source code, looking at the 6th release of the OpenSUSE project and its components defined by C functions. The results show that Linux applications have high quality code standards that one might expect in an open source repository, but the quality is lower than the one implied by the standard. More than half of the components were in a high state of quality, but on the other hand, most lower quality components cannot be improved only by applying some corrective actions. Thus, even though not all the source code was in an industrial standards shape, there is definitely room for further improvement and open source repositories proved to be of good quality.

After defining what data quality is and its characteristics, the next step would be data measurement or information quality in a project.

There are several research papers that try to find the answer, among which the work of Lee et al. [2002]. The authors tried to define an overall model along with an accompanying assessment instrument for quantifying information quality in an organization. The methodology has 3 main steps that need to be followed in order to apply it successfully:

- 2×2 model of what information quality means to managers;
- questionnaire for measuring information quality along the dimensions found in first step;
- two analysis techniques for interpreting the assessments captured by the questionnaire.

After developing the technique, they applied it in 5 different organizations and found that the tool proved to be practical.

Compared to the methodology proposed by Lee et al. [2002], the solution found by Heinrich et al. [2007] is quite different. Even though the main goals were the same, the authors used a single metric, and that is the metric for timeliness (i.e. whether the values of attributes still correspond to the current state of their real world counterparts and whether they are out of date). Thus, they applied the metric at a major German mobile services provider. Due to some Data Quality issues the company was having, they had lower mailing campaign success rates, but after applying the metrics, the company was able to establish a direct connection between the results of measuring data quality and the success rates of campaigns.

Nelson et al. [2005] proposed another technique for measuring data quality in an organization by, firstly, setting 2 main research questions:

- identify a set of antecedents that both drive information and system quality, as well as define the nature of the IT artifact;
- explore data warehousing in general, especially analytical tools, predefined reports and ad hoc queries.

Then, the authors set up a model to define a tree-structured representation of system quality and data quality, as follows:

- data quality - defined by completeness, accuracy, format and currency;
- system quality - defined by reliability, flexibility, integration, response time and accessibility;
- then, data and systems available are evaluated to infer data satisfaction and system satisfaction (coming from customers), which in turn will compute the final satisfaction score of the product.

After conducting a cross-functional survey to test the model, they learned that the features they selected were a good indicator of overall information/data and system quality. They also found that accuracy is the dominant determinant across all three data warehouse technologies, followed by completeness and format. Last but not least, another discovery was that more attention needs to be given to differences across varying technologies.

Another important area of data quality is what methodologies or techniques can be applied by organizations in order to improve it. There are several studies that tried to propose such methodologies. A general overview of such techniques was presented by Pipino et al. [2002] - they wanted to present ways of developing usable data quality methods that organizations can implement in their own internal processes. After reviewing various techniques of assessing data quality in information systems, they reached the conclusion that there is no universal approach to assess data quality as it heavily depends on the context where it is analysed.

One such methodology examined by Pipino et al. [2002] is the technique proposed by Wang [1998] called Total Data Quality Management (abbreviated TDQM). Through this method, the authors wanted to help organizations deliver high quality information products to information consumers by following the TDQM cycle - define, measure, analyse and improve information quality continuously. In order to do that, the research proposes 4 steps: clearly articulate the information product, establish several roles that would be in charge of the information product management, teach information quality assessment and management skills to all information products constituencies and, finally, institutionalize continuous information product improvement. After developing the methodology, the authors conclude on a confident note that their technique can fill a gap in the information quality environment.

However, there are other aspects that can be investigated when trying to improve data quality. For example, Prybutok et al. [2008] tested whether leadership and information

quality can lead to positive outcomes in an e-government's data quality. They conducted a web survey to gather data and test their hypotheses. Afterwards, they assessed the City of Denton's e-government initiatives, including current plans and implementations. They learned that the MBNQA leadership triad (leadership, strategic planning and customer/market focus) had a positive impact on the IT quality triad (information, system and service quality). Moreover, they found out that both leadership and IT quality improved the benefits.

Finally, after defining, measuring and improving data quality, we also need to be able to assess the impact poor DQ has on a company. In this area, there are several authors that investigated various implications of low indices of data quality.

One such research is the one done by Gorla et al. [2010]. They wanted to examine the influences of system quality, service quality and data quality on an organization, as well as what effect does system quality have on data quality. After conducting a construct measurement for the information systems quality dimensions [Swanson, 1997] and collecting data through empirical testing, they learned that service quality has the greatest impact of all three quality constructs. Moreover, they found out that there is a linkage between system and information quality, fact that previous research did not reveal. Last but not least, their results indicate that the above-mentioned quality dimensions have a significant positive influence on organizational impact either directly or indirectly.

On the other hand, Redman [1998] had different findings. Even though they did not analyze multiple types of quality, but only data quality, they learned that there are three main issues across most enterprises: inaccurate data, inconsistencies across databases and unavailable data necessary for certain operations. Furthermore, they presented a number of impacts that poor data quality has on enterprises: lowered customer satisfaction, increased cost, lowered employee satisfaction, it becomes more difficult to set strategy, as well as worse decision making.

Another research, the one of Lee and Strong [2003], investigated another aspect: whether knowing-why affects work performance and whether knowledge held by people in different roles affect the overall work performance. After conducting various surveys with 6 companies that served as data collection points, they reached the conclusion that there is vast complexity of knowledge at work in the data production process and if these gaps are not bridged it might lead to decreased data quality.

2.3 Issue Quality

Issue quality is the main topic of this literature review as it revolves around software tickets and the characteristics of a well-written and informative bug report, how efficient are bug tracking systems, what defines an efficient bug triaging process etc.

The first and most important sub-topic of issue quality, which is the one we will address in our own research as well, is the quality of bug reports. There are several authors that have

tried to defined what makes for a good bug report and what components actually improve the overall quality.

Bettenburg et al. [2008a] analyzed what makes for a good bug report through qualitative analysis. They interviewed over 450 developers and asked them what are the most important features for them in a bug report that help them solve the issue quicker. They reached the conclusion that stack traces and steps to reproduce increased the quality of a bug report the most, followed by well-written, grammar error free summaries and descriptions. Last but not least, they also created a tool called CueZilla that could with an accuracy rate between 31 and 48% predict the quality of a bug report.

Strengthening the argument that readability matters considerably in bug report quality is the work of Hooimeijer and Weimer [2007]. They ran an analysis over 27000 bug reports from the Mozilla Firefox project, looking at self-reported severity, readability, daily load, submitter reputation and changes over time. After running the evaluation, they not only found out about the importance of readability in bug reports, but also that attachment and comment counts are valuable for faster triaging and that patch count, for example, does not contribute in the same manner as the previous two.

Another research that agrees that stack traces are helpful in solving bug reports faster is the one performed by Schroter et al. [2010]. They conducted the whole experiment on the Eclipse project. They first extracted the stack traces using the infoZilla tool([Bettenburg et al., 2008c]), followed by linking the stack traces to changes in the source code (change logs) by mining the version repository of the Eclipse project. The results showed that around 60% of the bugs that contained stack traces in their reports were fixed in one of the methods in the frame, with 40% being fixed in exactly the first stack frame.

After examining the work of Bettenburg et al. [2007], we noticed that the authors found very similar results to the ones presented in the research of Bettenburg et al. [2008a]. They performed the same type of evaluation method (i.e. interviews with developers) and confirmed that steps to reproduce and stack traces are the most important features in a bug report that help developers solve the issue quicker.

Other interesting findings regarding the quality of a software project's tickets are the ones elicited in Bettenburg et al. [2008b]. The authors examined whether duplicate bug reports are actually harmful to a project or they add quality. They collected big amount of data from the Eclipse project and ran various kind of textual and statistical analysis on the data to find answers. They reached the conclusion that bug duplicates contain information that is not present in the master reports. This additional data can be helpful for developers and it can also aid automated triaging techniques.

However, in order to model the quality of a bug report, one needs to be able to successfully extract various types of information from such a report and, if possible, link them to the source code of the project (i.e. source code fragments in bug report discussions should be linked to the corresponding sections in the actual code) or other software artifacts. There are several researches that tried to analyze such techniques and one of them is the work of Bettenburg et al. [2012]. The authors created a tool that could parse a bug report (using

fuzzy code search) and extract source code that could then be matched with exact locations in the source code, in the end producing a traceability link (i.e. tuple containing a clone group ID and file paths corresponding to the source code files). Evaluation showed an increase of roughly 20% in total traceability links between issue reports and source code when compared to the current state-of-the-art technique, change log analysis.

Bettenburg et al. [2012] also made use of a tool developed by Bettenburg et al. [2008c] called infoZilla. This application can parse bug reports and correctly extract patches, stack traces, source code and enumerations. When evaluating it on over 160.000 Eclipse bug reports, it proved to have a very high rate of over 97% accuracy.

However, Kim et al. [2013] propose a rather different technique than the one exposed in Bettenburg et al. [2012]. The authors employed a two-phase recommendation model that would locate the necessary fixes based on information in bug reports. Firstly, they did a feature extraction on the bug reports (e.g. extract description, summary, metadata). Then, in order to successfully predict locations, this model was trained on collected bug reports and then, when given a new bug report, it would try to localize the files that need to be changed automatically. Finally, the actual implementation was put into place, and that is the two phase recommendation model, composed of binary (filters out uninformative bug reports before predicting the files to fix) and multiclass (previously filtered bug reports are used as training data and only after that new bug reports can be analyzed and files to be changed recommended). The overall accuracy was above the one achieved by Bettenburg et al. [2012], being able to rank over 70%, but only as long as it recommended *any* locations.

One other type of information that could be extracted and used for valuable purposes is the summary of a bug report, as shown in the work of Rastkar et al. [2010]. The authors wanted to determine if bug reports could be summarized effectively and automatically so that developers would need only analyze summaries instead of full tickets. Firstly, they asked university students to volunteer to annotate the bug reports collected from various open sources projects (i.e. Eclipse, Mozilla, Gnome, KDE) by writing a summary of maximum 250 words in their own sentences. These human-produced annotations were then used by algorithms to learn how to effectively summarize a bug report. Afterwards, the authors asked the end users of these bug reports, the software developers, to rate the summaries against the original bug reports. They eventually learned that existing conversation-based extractive summary generators trained on bug reports produce the best results.

A third technique for linking source code files from the textual description of a bug report is the one proposed by Zhou et al. [2012]. The authors first performed a textual analysis, looking for similarities between the bug report description and the source code files. Then, it analyzed previous bugs in the repository to find the most similar ones, thus being able to find which files ought to be changed. Lastly, it assigned scores to similar files, the ones with bigger sizes obtaining higher scores as they are more likely to contain bugs. They collected the necessary data from Bugzilla projects and then performed the evaluation. The results of the evaluation phase were positive: the bug locator can locate a large percentage of bugs analyzing just a small set of source code files. Moreover, similar bugs can improve the localization accuracy only to a certain extent, while the locator outperformed every other

competitor on a multitude of projects.

Having discussed automated linkage between source code files and bug reports, Fischer et al. [2003] presents a way to track certain features from the data residing in tickets. The authors employed a method to track features by analyzing and relating bug report data filtered from a release history database. Features are then instrumented and tracked, relationships of modification and problem reports to these features are established, and tracked features are visualized to illustrate their otherwise hidden dependencies. They managed to successfully visualize the tracked features and illustrate their non apparent dependencies which can prove very useful in projects with a large number of components.

Even though we have discussed about quality in bug reports, how it can be identified and what state-of-the-art techniques can be used to extract and link various components from tickets, we must also investigate if the platform where these bug reports reside (i.e. bug trackers) are properly fitting the software environment. The work of Just et al. [2008] tries to find the answer to this question. The authors launched a survey to developers from Eclipse, Mozilla and Apache open source projects from which they received 175 comments back. Then, they applied a card sort in order to organize the comments into hierarchies to deduce a higher level of abstraction and identify common patterns. In the end, they ranked the top suggestions as follows:

- provide tool support for users to collect and prepare information that developers need;
- find volunteers to translate bug reports filed in foreign languages;
- provide different user interfaces for each user level and give cues to inexperienced reporters on what to report and best practices;
- reward reporters when they do a good job;
- integrate reputation into user profiles to mark experienced reporters;
- provide powerful and easy to use tools to search bug reports;
- encourage users to submit additional details (provide tools for merging bugs).

One other important aspect when analyzing the quality of a bug report is the time it takes to triage it. Triaging is the process of analyzing issues and based on the information available, determine their severity and assign them to the most qualified available developer to fix them. If the ticket was valuable and had an increased index of quality, the triaging process would be quicker and, thus, reduce the cost for the organization. There are several researches that are concerned with this field and how gaps can be bridged.

One such example is the work of Anvik [2006]. The authors automated the process of bug report assignment by employing a machine learning algorithm which, after receiving as input various types of information (e.g. textual description of bug, bug component, OS, hardware, software version, developer who owns the code, current workload of developers, list of developers actively contributing), would recommend the best developers to work on

the ticket. After the tool was evaluated, it was found that it could actually be deployed in the software industry so that it could be used by organizations.

Working on the previous tool, the same authors proposed a new solution to automatic bug report assignment through the work presented in Anvik and Murphy [2011]. They increased the precision and recall of the algorithm and then, in order to test it, they implemented a proxy to the actual web service that's providing the issue repository. Eventually, they also found out that the impact of poor software project management, including bug triaging, on software projects can reach alarming levels.

A similar research, but which looks at the free form of the ticket only, is the technique presented in Anvik et al. [2006]. The authors applied a supervised machine learning algorithm on the repositories to learn which developers were best suited for specific tasks, thus when a new bug report would come in, a small set of people would be selected. In order to train the algorithm, they looked at Bugzilla repositories and selected the free text form of tickets, trying to label similar ones based on textual similarities. Once the tickets were labeled and grouped for specific developers, the algorithm would then be able to present the triager the set of developers suitable to fix the bug. Even though this approach, compared to Anvik [2006] or Anvik and Murphy [2011], does not reach the same levels of accuracy, it is much simpler to implement and evaluate.

However, a completely different approach is the one presented by Jeong et al. [2009] that looks at both bug report assignment as well as bug tossing (i.e. re-assign tickets from one developer to another). They wanted, through their tossing graph approach based on the Markov property, to both discover developer networks and team structures, as well as help to better assign developers to bug reports. They analyzed 145,000 bug reports from Eclipse and 300,000 from Mozilla and then, using statistical analysis on the bug reports in order to find evolution histories and changes throughout the lifetime of the reports, they created the bug tossing graph. The research shows that it takes a long time to assign and toss bugs and, additionally, they learned that their model reduces tossing steps by up to 72% and improved the automatic bug assignment by up to 23%.

Another approach to bug report assignee recommendation is the one based on activity profiles, proposed by Naguib et al. [2013]. The authors employed an algorithm using activity profiles (i.e. assign, review, resolve activity of the developer) such that, after detecting the prior experience, developer's role, and involvement in the project, it could recommend who should fix a specific bug. The average accuracy was around 88%, much higher than other techniques, including the ones mentioned previously.

However, bug report assignment is not the only part of bug triaging that needs investigation. Lamkanfi et al. [2010] show a method through which one can predict the severity of a reported bug (i.e. assign story points to the ticket) automatically. The authors conducted their research on the Mozilla, Eclipse and GNOME projects and divided their method into four steps:

- extract and organize bug reports;

- pre-process bug reports - tokenization, stop word removal and stemming;
- train the classifier on multiple datasets of bug reports - 70% training and 30% evaluation;
- apply the trained classifier on the evaluation set.

The conclusions they drew from running the evaluation process were rather interesting:

- terms such as deadlock, hand, crash, memory or segfault usually indicate a severe bug; however, there are other terms that can indicate the opposite, such as typos;
- when analyzing the textual description they found that using simple one line summaries resulted in less false positives and increased precision levels;
- the classifier needs a large dataset in order to predict correctly;
- depending on how well a software project is componentized, one can use a predictor per component rather than an universal one.

2.4 Measuring Cost and Waste in Software Projects

Having talked about data and ticket quality and its characteristics, we have another issue that inevitably arises in every software project: software waste. There are different researches that try to define what waste in software development is as well as its main types. One such paper is the one of Sedano et al. [2017]. The authors conducted a participant-observation study over a long period of time at Pivotal, a consultancy software development company. They also interviewed multiple engineering and balanced theoretical sampling with analysis to elicit the following main types of waste in software project:

- building the wrong feature or product;
- mismanaging backlog;
- extraneous cognitive load;
- rework;
- ineffective communication;
- waiting/multitasking;
- solutions too complex;
- psychological distress.
- **Relevance to our work**: this paper complements the previous one on waste identificationKorkala and Maurer [2014].

Even though Sedano et al. [2017] look at waste generally in software development, Ikonen et al. [2010] try to specifically investigate waste in kanban-based projects. They controlled a case study research at a company called Software factory using semi-structured interviews. The authors reached two main conclusions:

- they couldn't explain the success of the project even though waste was found;
- they identified 7 main types of waste throughout various development stages: partially done work, extra processes, extra features, task switching, waiting, motion and defects.

Apart from building the wrong feature/product, the 2 works differ in what types of waste they identified.

However, defining waste and its main types is not enough - one needs also ways to identify such waste. Korkala and Maurer [2014] propose a technique to identify communication waste in agile software projects environments. The authors collaborated with a medium-sized American software company and conducted a series of observations, informal discussions, documents provided by the organization, as well as semi-structured interviews. Moreover, the data collection for waste identification was split into 2 parts:

- **pre-development**: occurred before the actual implementation begun (e.g. backlog creation);
- **development**: happened throughout the implementation process (e.g. throughout sprints, retrospectives, sprint reviews, communication media).

Eventually they reached two main conclusions:

- this proposed approach was efficient and the authors recommend it to companies if they'd like to conduct such processes internally;
- the research elicits 5 main categories for communication waste: lack of involvement, lack of shared understanding, outdated information, restricted access to information, scattered information.

Another work that is trying to analyze the waste around software coordination activities is the one of Aranda and Venolia [2009]. They are trying to understand common bug fixing coordination activities in terms of software projects coordination and propose different directions on how to implement proper tools. They conducted a field study which was split into two parts:

- led an exploratory case study of bug repositories histories;
- then, they conducted a survey with professionals (e.g. testers, developers).

After they finished the 2 phases, they learned that there are multiple factors which influence the coordination activities that revolve around bug fixing, such as organizational, social and technical knowledge, thus one cannot infer any conclusions only by automatic analysis of the bug repositories. Also, through surveying the professionals, they reached the conclusion that there are *eight main goals* which can be used for better tools and practices: summit, probing for ownership, probing for expertise, code review, triaging, rapid-fire emailing, shotgun emails and infrequent/direct email.

However, having discussed about software and communication waste, what work has been done around another major software activity, i.e. software cost estimation? The paper from Grimstad and Jørgensen [2006] showed that poor estimation analysis techniques in software projects will lead to wrong conclusions regarding cost estimation accuracy. Moreover, they also propose a framework for better analysis of software cost estimation error. They approached a real-world company where they conducted analysis on their cost estimation techniques. Eventually, they learned that regular, straight-forward types of cost estimation analysis techniques errors lead to wrong conclusions, thus increasing the risk of poor coordination activities.

But what happens when a project gains maturity? A work that investigates the effects of process maturity on quality, cycle time and effort is the one undergone by Harter et al. [2000]. The research proves and rejects several hypotheses, such as higher levels of process maturity lead to higher product quality, higher levels of process maturity are associated with increased cycle time, higher product quality is associated with lower cycle time, higher levels of process maturity lead to increased development efforts in the project and higher product quality is associated with lower development effort. The authors examined data coming from 30 projects belonging to the systems integration division from a large IT company and analyzed a couple of key variables and see how they interact (i.e. process maturity, product quality, cycle time, development effort, product size, domain/data/decision complexity and requirements ambiguity). They learned that the main features they inspected are additively separable and linear. Moreover, they found out that higher levels of process maturity are associated with significantly higher quality, but also with increased cycle times and development efforts. On the other hand, the reductions in cycle time and effort resulting from improved quality outweigh the marginal increases from achieving more process maturity.

As we want to analyze comments as well and how they influence the quality of a ticket, we need to investigate what state-of-the-art tools are available to perform sentiment analysis. One paper that tried to detect the overall sentiment transmitted through reviews of various types is the research performed by Turney [2002]. The author created an unsupervised machine learning algorithm that was evaluated on more than 400 reviews on Epinions on various kins of markets. The algorithm implementation was divided into three steps:

- extract phrases containing adjectives or adverbs;
- estimate the semantic orientation of the phrases;

- classify the review as recommended or not recommended based on the semantic orientation calculated at previous step.

The author learned that different categories will yield different results, such as the automobile section on Epinions ranked much higher, 84%, compared to movie reviews, which had an accuracy of 65.83%. Moreover, most pitfalls of the algorithm could be attributed to multiple factors, such as not using a supervised learning system or limitations of PMI-IR.

However, the work of Turney [2002] looks only at sentiment analysis without examining the context. The research from Wilson et al. [2005] found efficient ways to distinguish between contextual and prior polarity. They used a two step method that used machine learning and a variety of features. The first step classified each phrase which had a clue as either neutral or polar, followed by taking all phrases marked in the previous step and giving them a contextual polarity (e.g. positive, negative, both, neutral). Through the method the authors employed, they managed to automatically identify the contextual polarity. As most papers were only looking at the sentiment extracted from the overall document, they managed to get valuable results from looking at specific words and phrases.

3 Proposed Approach

3.1 Experimental Design

We would like to begin the research by conducting a statistical analysis over popular open source projects such as Mozilla or Apache and extract tickets most probably via REST APIs, however the list of projects will be finalized once some tests will be run, and that is because maybe some projects will have more valuable tickets for us to analyze and we want to analyze multiple issue tracking systems (e.g. Apache uses Jira [Atlassian] while Mozilla uses Bugzilla [Mozilla]).

Then, as most issue tracking systems respond with JSON payloads through their REST APIs, we can easily extract components such as summary or description and analyze them. We have defined that the initial list of components that we want to inspect into three main categories:

- regular text - summary, description, story points, text in comments, status (e.g. OPEN, WONTFIX, CLOSED);
- non-regular text (i.e. special structures) - stack traces, steps to reproduce;
- non-text - attachments in comments (e.g. images, screenshots when trying to show steps to reproduce).

Extracting text from components is rather trivial, but extracting and analyzing non-text data such as images or non-regular text will be more complex.

For images, there are two very popular REST APIs that we can use for analysis: Google Cloud Vision [Cloud] and AWS Rekognition [AWS]. When an image is sent via the API, the servers respond with detailed information about what can be found in the image, thus we can detect if the image was for example a simple screenshot or if it contained multiple screenshots that were aimed at visually showing steps to reproduce a bug.

In terms of non-regular text, steps to reproduce are usually trivial to retrieve (if they are not images, for which we have presented a solution in the above paragraph), they will most probably be written as bullet lists or a sequence of lines starting with *Step [number] ...*, thus a simple regex will probably suffice. However, for stack traces, as they are more complex, there are no commercial or open source applications that could help them extract them easily, thus we will compose a special regex depending on the language of the project (e.g. Apache heavily uses Java, thus we can look for lines starting with java.*.*Exception ...).

Implementation wise, we would like to develop the application that will perform both scraping, extraction and inspection in either Go or Java.

In order to detect what features make for a high quality ticket, we need to compare two *similar* tickets and then analyze their components for differences. Thus, one of the biggest issues we will be facing is comparing and contrasting tickets. We have two initial approaches that we want to test to see which one is more efficient:

- look solely at the issue tracking system and use story points to detect similar tickets; one limitation of this approach is that either the triager is not experienced and might assign wrong story point values or the initial story point values are not properly assigned and they will be corrected during the ticket's lifecycle;
- look at the version control system and compare changesets for similar work effort; however, this approach has one limitation - solely looking at source code changes might not encapsulate the whole effort needed to solve the task (e.g. the developers on the team might spend extra hours only on designing the fix).

However, this list is not final, so if we find some other method while running the experiment, we shall switch to that.

Finally, grouping similar tickets together allows testing the variances in component features they possess and see which features affected the time-to-close the most.

3.2 Variables Tested

The independent variables proposed for testing are:

- presence and number of stack traces in a ticket;
- steps to reproduce (as described above - images or text);

- presence of screenshots in tickets;
- text components (e.g. summary, description) being either succinct or detailed;
- grammar-free textual description;
- overall sentiment drawn from comments (e.g. angry or calm);
- complex or light discussion in comments.

In terms of dependent variables, there is only one - the time it takes between when the ticket was created and marked as OPEN until when it is finished and marked as CLOSED or RESOLVED.

4 Work Plan

The work on the actual project in the second semester will start off with deciding and analyzing which repositories to use for ticket data. There are several popular open source options, such as Apache, Mozilla or Eclipse, so settling on a couple of options will be the first step in our research. We believe that this step will be finished *by the end of January.*

The next step will be to create the scraping, extraction and inspection tool which will be implemented in either Go or Java. This will be a vital component of the whole research so we believe that it will be ready to run *by the end of February.*

Then, concomitantly with the previous work package, the method used to detect similar tickets will be decided on. This step will be performed during the creation of the tool because efficiency can only be evaluated once ideas have been validated via execution of several mock tickets. Thus, the proposed deadline for this work package is *the end of February* as well.

Following, tickets collected will be run in the tool created as per the previous steps. The expected deadline is *by* 4^{th} *of March.*

Then, the actual write-up of the paper will be structured as follows:

- write abstract, introduction, related work - this is expected to be completed *by mid March*;
- write methodology (collecting data, running analysis) - expected to be finished *by the beginning of April*;
- write results, contributions and conclusion for the paper - we expect to complete it *by* 18^{th} *of April.*

References

John Anvik. Automating bug report assignment. pages 937–940, 2006.

John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10, 2011.

John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? pages 361–370, 2006.

Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. pages 298–308, 2009.

Atlassian. Jira. `https://www.atlassian.com/software/jira`. [Online; accessed 7-December-2017].

Amazon AWS. Rekognition API. `https://aws.amazon.com/rekognition/`. [Online; accessed 12-December-2017].

Adrian Bachmann and Abraham Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. pages 119–128, 2009.

Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. Quality of bug reports in eclipse. pages 21–25, 2007.

Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? pages 308–318, 2008a.

Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful really? pages 337–345, 2008b.

Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. pages 27–30, 2008c.

Nicolas Bettenburg, Stephen W Thomas, and Ahmed E Hassan. Using fuzzy code search to link code fragments in discussions to source code. pages 319–328, 2012.

Google Cloud. Vision API. `https://cloud.google.com/vision`. [Online; accessed 12-December-2017].

Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. 3:90, 2003.

Narasimhaiah Gorla, Toni M Somers, and Betty Wong. Organizational impact of system quality, information quality, and service quality. *The Journal of Strategic Information Systems*, 19(3):207–228, 2010.

Stein Grimstad and Magne Jørgensen. A framework for the analysis of software cost estimation accuracy. pages 58–65, 2006.

Donald E Harter, Mayuram S Krishnan, and Sandra A Slaughter. Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science*, 46(4):451–466, 2000.

Bernd Heinrich, Marcus Kaiser, and Mathias Klier. Metrics for measuring data quality foundations for an economic data quality management. pages 87–94, 2007.

Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. pages 34–43, 2007.

Marko Ikonen, Petri Kettunen, Nilay Oza, and Pekka Abrahamsson. Exploring the sources of waste in kanban software development projects. pages 376–381, 2010.

Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. pages 111–120, 2009.

Sascha Just, Rahul Premraj, and Thomas Zimmermann. Towards the next generation of bug tracking systems. pages 82–85, 2008.

Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.

Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: the elusive target [special issues section]. *IEEE software*, 13(1):12–21, 1996.

Mikko Korkala and Frank Maurer. Waste identification as the means for improving communication in globally distributed agile software development. *The Journal of Systems and Software*, 95:122–140, 2014.

Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. pages 1–10, 2010.

Yang W Lee and Diane M Strong. Knowing-why about data processes and data quality. *Journal of Management Information Systems*, 20(3):13–39, 2003.

Yang W Lee, Diane M Strong, Beverly K Kahn, and Richard Y Wang. Aimq: a methodology for information quality assessment. *Information & management*, 40(2):133–146, 2002.

Mozilla. Bugzilla. `https://www.bugzilla.org/`. [Online; accessed 8-December-2017].

Hoda Naguib, Nitesh Narayan, Bernd Br"ugge, and Dina Helal. Bug report assignee recommendation using activity profiles. pages 22–30, 2013.

R Ryan Nelson, Peter A Todd, and Barbara H Wixom. Antecedents of information and system quality: an empirical examination within the context of data warehousing. *Journal of management information systems*, 21(4):199–235, 2005.

Leo L Pipino, Yang W Lee, and Richard Y Wang. Data quality assessment. *Communications of the ACM*, 45(4):211–218, 2002.

Victor R Prybutok, Xiaoni Zhang, and Sherry D Ryan. Evaluating leadership, it quality, and net benefits in an e-government environment. *Information & Management*, 45(3): 143–152, 2008.

Sarah Rastkar, Gail C Murphy, and Gabriel Murray. Summarizing software artifacts: a case study of bug reports. pages 505–514, 2010.

Thomas C Redman. The impact of poor data quality on the typical enterprise. *Communications of the ACM*, 41(2):79–82, 1998.

Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? pages 118–121, 2010.

Todd Sedano, Paul Ralph, and Cécile Péraire. Software development waste. pages 130–140, 2017.

Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, and Georgios L Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12 (1):43–60, 2002.

Diane M Strong, Yang W Lee, and Richard Y Wang. Data quality in context. *Communications of the ACM*, 40(5):103–110, 1997.

E Burton Swanson. Maintaining is quality. *Information and Software Technology*, 39(12): 845–850, 1997.

Peter D Turney. Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews. pages 417–424, 2002.

Richard Y Wang. A product perspective on total data quality management. *Communications of the ACM*, 41(2):58–65, 1998.

Theresa Wilson, Janyce Wiebe, and Paul Hoffmann. Recognizing contextual polarity in phrase-level sentiment analysis. pages 347–354, 2005.

Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. pages 14–24, 2012.