The Effect of “*Good First Issue*” Indicators upon Newcomer Developers

Identifying Improvements for Newcomer Task Recommendation

Jan Willem David Alderliesten

The Effect of "*Good First Issue*" Indicators upon Newcomer Developers

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jan Willem David Alderliesten
born in Washington, District of Columbia, United States of America

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

The Effect of “*Good First Issue*” Indicators upon Newcomer Developers

Author: Jan Willem David Alderliesten
Student id: 4368703
Email: `j.w.d.alderliesten@student.tudelft.nl`

Abstract

The recommendation of tasks for newcomers within a software project through *good first issue*s is being done within the domain of software development, such as on *Github* platform. These issues aim to help newcomers identify tasks that are suitable for them and their level of expertise within the project. This thesis report investigates the effectiveness regarding developer onboarding and task completion of *good first issue*s by data mining a set of 105 repositories and manually analyzing at most 30 *good first issue*s and 30 initial commits per sampled project. It was found that, although *good first issue*s are effective at developer onboarding, and developers perceive *good first issue*s as being useful, changes can be made to the types of tasks suggested as *good first issue*s to match the types of initial contributions made by newcomers. It was also found that developers with less than a year of experience favored *documentation*-related contributions for their first commit to a project.

Preface

It feels somewhat strange writing the preface. For you, as reader, it should be known that this is the final component for this thesis that was written before it was sent off to be inspected and critiqued by the committee. I had not expected that my thesis would be investigating *good first issue*s, nor did I expect that I'd enjoy this work as much as I did. What started out as a thesis born out of an interest in repository mining ended-up being, what I hope, a contribution for projects and repositories to help with developer onboarding. An attempt at finding a method of numerically evaluating a process that has already existed for quite some time. It feels doubly strange that I will not have a formal defense but an online defense, due to the ongoing situation surrounding the Wuhan strain of the Corona virus, and only physically saw my supervisor twice during the entire thesis process, but an uncountable amount of times online. Interesting stories to tell when I'm older to an interested ear, perhaps.

I mostly hope that you, the reader, enjoy reading the thesis report and are able to extract information that may help in organizing your projects or determining your next contributions to projects. If you have any remarks or questions, you are welcome to send me an e-mail at *david.j.w@hotmail.com* to discuss, even if it were to be years later. Perhaps a bit unorthodox to share my private e-mail in the preface, but then again, this entire procedure was less than normal and electronic communication has become the standard during this time. There are numerous people I'd like to thank in this preface.

I'd like to begin by thanking my parents, Liza and Jan, for their continued support and advice. Many evenings were spent with me discussing my progress, ranting about problems I encountered, and going through concept versions of my thesis that came back with feedback and advice. I am very grateful for all your support, the thesis is much stronger due to your advice and support and I would not have been able to complete my master's program without the many, many conversations we've had.

I would also like to thank the many friends that kept me sane during the largest parts of this thesis. Although I am not able to name everyone, I would especially like to thank (in no particular order) Alex Molenberg, Cas Buijs, Kasper Kop, Hugo Meeldijk, Jesse Tilro, Niels Warnars, Floris Doolaard, and many others who are not named for countless nights of gaming, walks, and phone talks during these times. It was a much needed break from the usual, and I cannot thank you enough for this. Once this whole situation is over, we need to come together and celebrate! I would also like to thank the many students who I taught in the capacity of teaching assistant during this time. Many of you remained interested in my progress after I was done, and these chats were a very friendly interaction that has led, in many groups, to sustaining social groups. Thank you all.

Thirdly, I wish to thank my thesis evaluation committee, consisting of professor Georgios Gousios and professor Rafael ("*Rafa*") Bidarra for their willingness to evaluate my thesis work and to provide their critiques despite these unconventional and busy times. I also wish to thank the developers who responded to the survey I sent out, your kind responses helped motivate me to finish this research.

Finally, I wish to thank my supervisor, Professor Andy Zaidman. I was always dreading my thesis work, and had heard from many people that the thesis was perhaps the most painful and difficult part of a master's degree program. Thanks to your guidance, I found it to be one of the smoothest and most enjoyable components of my master program. You were always readily available to provide commentary, help me out with an issue I was facing, and provided a lot of advice that improved the quality of this thesis and guided me to better research practices. When I began this thesis report, I could not have imagined that I would learn so much related to both the domain of software engineering and to research procedures and practices. I am very grateful for all of this, and I hope to shake your hand once this entire situation is over to thank you for all you've done.

I hope you enjoy reading this thesis as much as I enjoyed writing it, and hope this thesis report finds you in good health.

Jan Willem David Alderliesten
Delft, the Netherlands
June 18, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of open-source software (*OSS*) relies largely on contributions made by developers and contributors who work without compensation and on their own accord [38]. The cumulative effort of these distributed developers results in software that is utilized by a significant number of individuals and organizations [53] and within closed-source packages (such as the utilization of the *C++* "*Boost*" library[1] in video games and optimization-based software). Due to the distributed and asynchronous nature of open source software development, regulations and procedures are required to ensure that development occurs in a steady and predictable manner such that all contributors are working with an identical set of expectations. Developers are also expected to acquaint themselves with these regulations, procedures, and with the technical aspects related to software development. Due to differences in technical expertise and experience per developer, coupled with a high turnover rate for developers in open source software projects [43], this can be a difficult task.

As a result of these varying factors and procedures, developers in open source software projects are expected to take the responsibility of integrating within the project and identifying tasks that can be done that are appropriate to their level. To assist this process, existing developers in open source software projects recommend certain tasks by labelling them as appropriate for beginners or newcomers. Within the *Github*[2] ecosystem, tasks can be stored as *issues* and these issues can be given a label called "*good first issue*." These tasks are aimed to be of such a level that someone unfamiliar with the processes and technical intricacies of the particular project can be introduced to the project and gain a deeper understanding of the project. Since these labels are given to tasks and issues as indicated by existing developers, however, the question arises whether the indicated tasks are actually good introductory tasks for new contributors, and, if these tasks are taken and completed, what types of tasks tend to be preferred by and for newcomers.

This thesis aims to analyze and identify trends and practices related to the indication of so-called *good first issue*s and their effect on the process of developer onboarding, if such an effect exists. This thesis also aims to identify the types of tasks commonly related to *good first issues* and the adoption rate of features allowing the labelling of these tasks within the

[1]Additional information can be found at: `https://www.boost.org/`
[2]Website can be found at `https://github.com`

open source *Github* ecosystem. Within this introduction, the motivation for the research and an indication of its importance is provided in Section 1.1. An overview of the domain and background knowledge required for an understanding of the research is provided in Section 1.2. The research questions for this thesis are stated and justified in Section 1.3, the contributions made by this thesis work in Section 1.4, and the structure of the remainder of the thesis report is outlined in Section 1.5.

1.1 Motivation for Research

The onboarding of developers within open (source) projects is a well-studied sub-domain of software engineering research, with research having been done investigating the social factors affecting developer onboarding [6], the impact of previous development experience upon project selection and onboarding [34], and other such aspects. Additional information regarding onboarding can be found in Chapter 2. It can be seen, however, that many papers study the effects of direct recommendation of repositories as opposed to the recommendation of tasks within a repository.

Within open (source) projects, new developers will form the next backbone of the project and are a project's most important resource [22]. These developers will guide development for the project in the coming years, and the transfer of knowledge and acquisition of newcomers is therefore important. As a result, not only is the onboarding of developers into a project important, but ensuring that developers are guided within the project ensures that they remain active within the project and gain a better (technical) understanding of the project. This will then, as a result, lead to a developer that is capable of taking increasingly complex tasks and resulting in more contributions to the project alongside the acquisition of deeper technical knowledge.

Thus, studying possibilities to onboard and guide developers from within a project is warranted. When considering the possibilities, labels that directly steer newcomers to possible development tasks can be considered as a prime contributor to this process. When considering the literature as done in Chapter 2, no papers seem to have investigated the effects of labelling possible tasks for newcomers directly. By investigating the types of tasks that tend to be picked-up by newcomers, and by studying the effectiveness of labelling and suggesting tasks directly to newcomers, possible adjustments can be made to improve the onboarding of new developers within projects and their retention rate within the project.

1.2 Background

The background section aims to provide an overview of content related to the domain of this thesis report, with the goal of providing the necessary background to the reader such that the contributions of this work can be understood and analyzed. This includes an overview of open source projects in Section 1.2.1, a primer on software repositories in Section 1.2.2, and an overview of version control systems in Section 1.2.3. Additionally, background is provided on *Github* and its services in Section 1.2.4, the idea of labels and the *Good First Issue* label is discussed in Section 1.2.5, and some information on software repository *mining* is introduced in Section 1.2.6. Additional terminology and an overview of all terminology in this section and within the thesis report can be found in the glossary provided in Appendix A.

1.2.1 Open Source & Openly Developed Software

The development of software can take place through upon demand, in which programmers develop software and are compensated for their work in the form of a wage or perhaps through equivalently-valued goods. This software is then sold at a price determined by the developer or releasing company for end users that will use the software product. The inner workings of the software product are not publicly available as to protect the trade secrets of the developed product. An exemption to this practice, however, exists within the open source software community. Open source software is software that has been developed and is released with its codebase public, meaning end users can view, modify, or change any aspect of the codebase if they wish[3]. Open source software is also usually released for free, meaning no compensation is given to the developer of the software.

In recent years, many professional software companies have released software that was once *closed source* software as open source software. Examples of this include *Google*'s *Kubernetes*[4] and *Facebook*'s *React*[5]. On some occasions, this open sourcing allows inspection and free-use of the codebase by external parties but retains the rights of software product in hands of the company responsible for its original development. This adds the benefit that the company itself can rely on code review and create possible enhancements to its original product from developers that it does not have to monitor or compensate.

It is estimated that open source software development has a beneficial impact for both end users of software products [31] and companies or individuals relying on this software [7] due to the increased quality resulting from public code analysis and due to the increased rate at which development occurs resulting from the larger developer base behind open source software projects.

1.2.2 Software Repositories

The development of software through means of multiple developers commonly relies on the utilization of repositories for code storage and development. This is required due to the fact that exchanging codebases (a current version of code) between developers through means of electronic mail or a physical storage medium transfer requires analysis of the entire codebase for each iteration given to a developer. This would mean that each change made by another developer would require one to iterate over all the code, in the hopes of finding the areas that were changed and integrating one's own work into this changed codebase. Repositories aim to alleviate and avoid such practices by providing a centralized location for all code to be stored, and through means of a version control system (such as *Git*) that

[3]As defined by `https://opensource.com/resources/what-open-source`
[4]Website can be found at `https://kubernetes.io/`
[5]Website can be found at `https://reactjs.org/`

Name	Type
Bazaar	Open-Source, Distributed
Concurrent Versions System	Open-Source, Centralized
Git	Open-Source, Distributed
Mercurial	Open-Source, Distributed
Subversion	Open-Source, Centralized
Visual Studio Teams	Proprietary, Distributed

Table 1.1: An overview of some existing version control systems.

provides a localized copy of the repository to each developer and manages the integration of all changes for these developers. This version control process is outlined in Section 1.2.3.

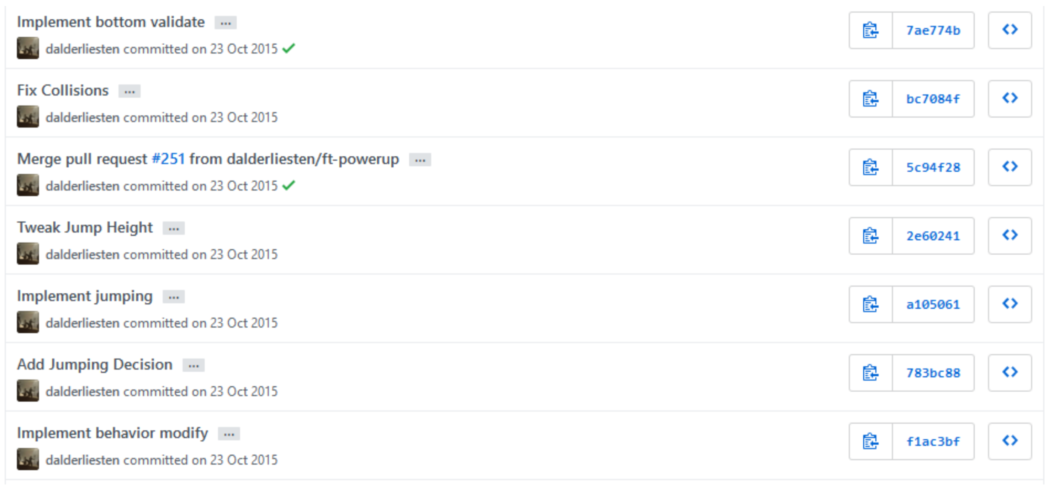
Within these repositories, a history of all contributions for each developer can be found, alongside a history of the codebase itself. Many repositories also contain the tracking and organization of other software product related aspect such as bug tracking [10][11][23], issue tracking (which tasks are assigned to which developer) [11][23], and the ability to review changes to the codebase that are submitted to the *master* version of the codebase, which contains the definitive codebase given to consumers and aimed at release. Software repositories can also contain an overview or archive of communication between developers related to that project [10][23].

1.2.3 Version Control & Git

When utilizing a software repository as described in Section 1.2.2, a set of standards should be introduced that allow developers to contribute their codebase changes within the repository. Version control systems feature certain standards and methodologies which allows software to be developed simultaneously between developers. This is achieved by tracking changes to the codebase and updating the codebase across all developers such that only the actual changes made by a developer are propagated to all developers. In simpler terms, this means that the codebase is only updated at the location in which the developer made actual changes (if a developer changed line seven in a 12-line class within a repository's codebase, the change of line seven would be the only update propagated). Version control systems either employ a central server which stores all changes and development, or require each developer to store a copy of the software repository locally and then submit changes to a centralized *master* copy. The former methodology is known as a centralized model, whereas the latter is known as a distributed model.

There are many version control systems for software development, of which an overview of a few is given in Table 1.1. One of these version control systems, named *Git*, has seen widespread utilization within the software development community[6]. Developed by Linus Torvalds, *Git* is a distributed version control system which allows developers to make *commits* containing codebase changes upon a local copy of the repository. When looking at a

[6]Experimental data viewable at: `www.openhub.net/repositories/compare`



Implement bottom validate ...
dalderliesten committed on 23 Oct 2015 ✓ — 7ae774b

Fix Collisions ...
dalderliesten committed on 23 Oct 2015 — bc7084f

Merge pull request #251 from dalderliesten/ft-powerup ...
dalderliesten committed on 23 Oct 2015 ✓ — 5c94f28

Tweak Jump Height ...
dalderliesten committed on 23 Oct 2015 — 2e60241

Implement jumping ...
dalderliesten committed on 23 Oct 2015 — a105061

Add Jumping Decision ...
dalderliesten committed on 23 Oct 2015 — 783bc88

Implement behavior modify ...
dalderliesten committed on 23 Oct 2015 — f1ac3bf

Figure 1.1: An example of a commit log for a software repository.

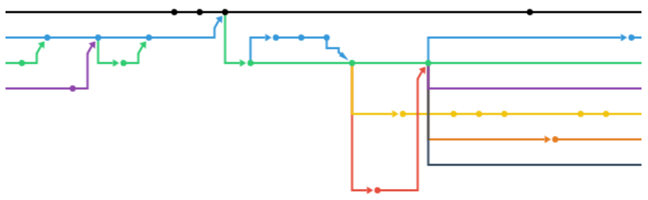


Figure 1.2: An example of a branch network within a software repository. The black line represents the *master* branch. Arrows represent *merges* of one branch into another.

log of all commits, such as the log shown in Figure 1.1, a complete history of the development of a software product or codebase can be obtained. Once a developer wishes to submit their commits, they *push* their work to the central repository storage.

All commits are added to a *branch*, which is a specific version of the codebase that contains changes related to a certain issue or task. An example of changes that warrant a branch could be a bug fix or a the development of a new feature. All branches are eventually *merged* (copied into) into a branch called *master*, which contains the release or user ready version of the codebase. It is also possible to merge non-*master* branches into other non-*master* branches, such as having one branch for a major feature development which itself has multiple branches containing smaller iterations of relevant components for that feature. This generates a branching network, of which an example can be seen in Figure 1.2.

Storing the centralized software repository for *Git* can be done by hosting a repository on-site, or hosting providers can be utilized. Free options for *Git* hosting include services such as *Github* and *Gitlab*[7].

[7]Found at: `www.about.gitlab.com`

1.2.4 Github & Github Issues

Github is an online software repository hosting service relying on the *Git* version control system. Users are able to host their repositories either publicly (allowing global visibility to any visitor of the site) or privately (only approved developers gain access to the repository). *Github* provides multiple auxiliary tools alongside each repository that aim to assist in the organization of tasks and assist in code review. Such features include *pull requests* (code reviews in which comments and specific lines of code can be highlighted for evaluation and discussion as shown in Figure 1.3), security notifications for dependencies that are outdated or are found to contain security vulnerabilities, textual markdown (.md) files outlining contribution guidelines within a repository[8], and *issues* (tasks that can be assigned to developers and classified based on a number of factors).

Github's issues are commonly utilized by projects to indicate tasks that developers in the project should or could perform. Each individual issues has a title or a description, can have additional textual information describing what must be done, can contain an assignee who is held responsible for completion of the task, and labels indicating the type of task. These labels can be customized per repository, resulting in label collections ranging from no labels to 100s of labels within a repository. Issues also have a status, indicating whether they are *open* (meaning the need to be done or are in-progress) or closed (indicating they are finished and have been merged into the *master* branch of a repository through a pull request). An example of a list of issues is shown in Figure 1.4.

1.2.5 The Good First Issue Label

As described in Section 1.2.4, issues can be labelled with custom labels. Upon creation of a repository, *Github* provides a number of default labels (such as "*bug*", "*feature*", and "*need help*") which can be utilized[9]. One of these suggested labels is "*good first issue*[10]." The good first issue label indicates tasks within a repository or open source project that are good for newcomers and first-time contributors to a repository. These issues should thus provide developers with a task that helps introduce them to a project or provides them with a greater understanding of the software project, as to stimulate them to take more difficult tasks down the line.

*Good first issue*s are also shared in other locations, sometimes off of the *Github* platform. Websites such as "*Up For Grabs*[11]" and *GoodFirstIssue.dev*[12] aim to provide alternative means of newcomer developers to identify tasks suitable for their level.

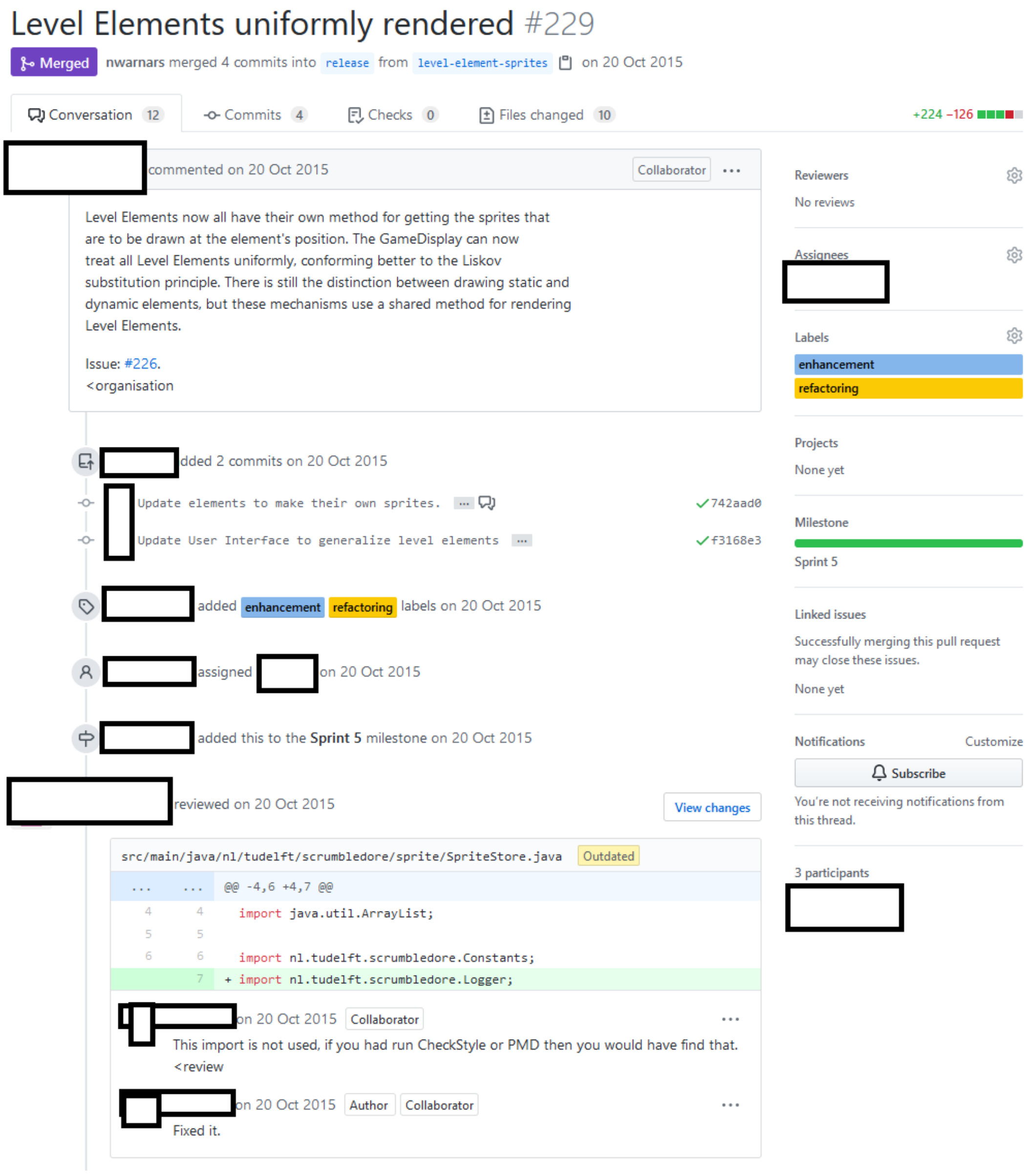
[8]Additional information can be found at: `https://www.github.blog/2012-09-17-contributing-guidelines`

[9]More information available at `https://help.github.com/en/github/managing-your-work-on-github/about-labels`

[10]Read more at: `https://help.github.com/en/github/building-a-strong-community/encouraging-helpful-contributions-to-your-project-with-labels`

[11]Website can be found at: `https://up-for-grabs.net/#/`

[12]Website can be found at: `https://goodfirstissue.dev/`



Level Elements uniformly rendered #229

Merged nwarnars merged 4 commits into `release` from `level-element-sprites` on 20 Oct 2015

Conversation 12 Commits 4 Checks 0 Files changed 10 +224 −126

[redacted] commented on 20 Oct 2015 Collaborator

Level Elements now all have their own method for getting the sprites that are to be drawn at the element's position. The GameDisplay can now treat all Level Elements uniformly, conforming better to the Liskov substitution principle. There is still the distinction between drawing static and dynamic elements, but these mechanisms use a shared method for rendering Level Elements.

Issue: #226.
<organisation

[redacted] dded 2 commits on 20 Oct 2015

[redacted] `Update elements to make their own sprites.` ✓742aad0
`Update User Interface to generalize level elements` ✓f3168e3

[redacted] added enhancement refactoring labels on 20 Oct 2015

[redacted] assigned [redacted] on 20 Oct 2015

[redacted] added this to the **Sprint 5** milestone on 20 Oct 2015

[redacted] reviewed on 20 Oct 2015 View changes

`src/main/java/nl/tudelft/scrumbledore/sprite/SpriteStore.java` Outdated

```
@@ -4,6 +4,7 @@
 import java.util.ArrayList;

 import nl.tudelft.scrumbledore.Constants;
+import nl.tudelft.scrumbledore.Logger;
```

[redacted] on 20 Oct 2015 Collaborator
This import is not used, if you had run CheckStyle or PMD then you would have find that.
<review

[redacted] on 20 Oct 2015 Author Collaborator
Fixed it.

Reviewers
No reviews

Assignees
[redacted]

Labels
enhancement
refactoring

Projects
None yet

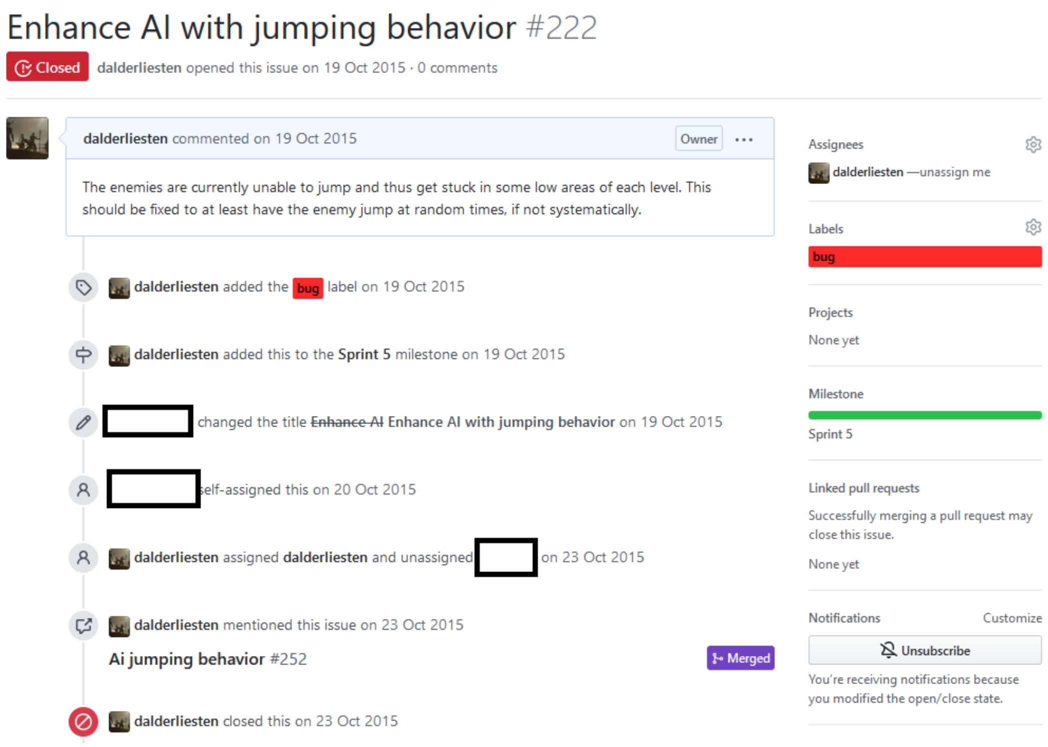
Milestone
Sprint 5

Linked issues
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
Subscribe
You're not receiving notifications from this thread.

3 participants
[redacted]

Figure 1.3: An example of a pull request within a repository on *Github*. The names and identifiable material have been removed to protect the contributor's privacy.



Enhance AI with jumping behavior #222

Closed dalderliesten opened this issue on 19 Oct 2015 · 0 comments

dalderliesten commented on 19 Oct 2015 Owner ···

The enemies are currently unable to jump and thus get stuck in some low areas of each level. This should be fixed to at least have the enemy jump at random times, if not systematically.

dalderliesten added the bug label on 19 Oct 2015

dalderliesten added this to the Sprint 5 milestone on 19 Oct 2015

[redacted] changed the title ~~Enhance AI~~ Enhance AI with jumping behavior on 19 Oct 2015

[redacted] self-assigned this on 20 Oct 2015

dalderliesten assigned dalderliesten and unassigned [redacted] on 23 Oct 2015

dalderliesten mentioned this issue on 23 Oct 2015

Ai jumping behavior #252 Merged

dalderliesten closed this on 23 Oct 2015

Assignees

dalderliesten —unassign me

Labels

bug

Projects

None yet

Milestone

Sprint 5

Linked pull requests

Successfully merging a pull request may close this issue.

None yet

Notifications Customize

Unsubscribe

You're receiving notifications because you modified the open/close state.

Figure 1.4: An issue on *Github* describing a bug that has been fixed and, as a result, closed the issue. The names and identifiable material related to one contributor have been removed to protect their privacy.

1.2.6 Software Repository Mining

Software repositories contain and store a lot of data related to software development, as outlined in Sections 1.2.2 and 1.2.3. The field of *software repository mining* revolves around the process of obtaining data from existing repositories and using this data to extract information related to software development and engineering purposes. This allows analysis of software development procedures and processes and can provide a basis for analysis and improvement of these processes. Initial mining aims to extract a significant amount of data and evidence [24], which is then commonly followed by manual analysis.

When mining software repositories, tools must be utilized to acquire the repository data. Multiple tools have been developed for the purposes of *Git* software repository mining and data archiving. Examples of such tools include *Boa* [11], *GHTorrent* [20], and *PyDriller* [46]. Each of these tools allows a user to *drill*, or search through, a software repository based on keywords or desired data. This data can consist of keywords, names, specific files, and other such queries. More specific queries allow drilling to identify certain patterns or trends up to the codebase level, meaning a driller could extract desired features within code, examples of which include identifying all lambda functions or the number of nested conditional statements within a software repository's codebase.

The resulting data can be utilized to identify trends or manually identify software engineering or development behavior that can lead to the improvement of software development practices. Results can be also be utilized to identify codebase trends or utilization statistics that can help identify regions of development that can be expanded upon in the future. If an analysis finds that many developers within a certain programming languages experience difficulties onboarding developers related to type *x*, then it might be worth investigating how to increase the number of new developers comfortable working on tasks of type *x* within that (open source) project.

1.3 Research Questions

To guide the research and software repository mining processes, a set of guiding research questions were devised which will be answered in this thesis report. One *main research question* was defined to guide overall research as given in Section 1.3.1. Multiple *secondary research questions* were then defined as given in Section 1.3.2 to more specifically tune the direction of research and to allow for conclusions to be made in other, perhaps more minor segments of the research.

1.3.1 Main Research Question

The main research question (MRQ) for this thesis requires the mining of software repositories with the goal of identifying the usefulness and level of utilization of the *good first issue* label and process. Hence, the main research question has been devised as:

> **Is the *good first issue* label effective in indicating tasks within an open (source) software project that are taken by newcomers?**

By focusing exclusively on the idea of a *good first issue* label (or a label with a different name performing the same function as the *good first issue* label), it is possible to analyze its effectiveness. By looking at the type of tasks that are recommended for newcomers, but also by comparing this to the actual contributions of newcomers, a baseline can be established to identify possible weaknesses or to indicate effectiveness.

1.3.2 Research Questions

This section describes the secondary research questions (RQs) that have been defined as a complement to the main research question described in Section 1.3.1. These research questions will also guide the research in such a manner that the main research questions can be answered and relevant aspects can be studied.

Level of Utilization The first set of secondary research questions aim to identify the level of utilization of *good first issues* within *Github* repositories and when considering the total number of issues within a repository. When considering the usefulness of *good first issue* labels it is important to consider whether the sample size considered is representative, and whether the feature itself sees widespread use among active *Github* repositories. Hence, the first RQ aims to identify the rate of adoption of good first issues as to allow us to draw conclusions about their actual adoption rate within *Github*.

> **RQ1**
> What percentage of projects within the *Github* ecosystem employ *good first issue* labels for their issues?

It is also important to consider the number of issues and tasks that are deemed to be a *good first issue* within individual repositories. This may provide an insight into how widespread the label is and what percentage of total issues within a repository could be considered as good for newcomers.

> **RQ2**
> What percentage of issues within repositories are labelled as *good first issue*s out of the total number of issues?

Labelling of *Good First Issues* The second set of research questions aim to investigate the types of tasks that are labelled as *good first issue*s and whether these suggested tasks are indeed taken and developed for initial contributions from new developers within a project. By classifying the types of tasks that are commonly deemed to be *good first issue*s and identifying common trends, it may be possible to provide suggestions to other projects regarding the types of issues to recommend for newcomers.

> **RQ3**
> What types of tasks and issues are generally recommended as *good first issue*s?

Beyond investigating and classifying commonly suggested *good first issue* tasks and issues, it is also important to investigate whether the labelled tasks are completed by newcomers. If the tasks are not completed by newcomers, this may suggest that the tasks being indicated are not suitable for newcomers. If the tasks are being completed by newcomers, then validation can be given to the classification as described in *RQ3* that these types of tasks are indeed generally suitable for newcomers.

> **RQ4**
> Do new developers complete the tasks labelled as *good first issue*s?

Newcomer Experience The third and final set of research questions aim to investigate the types of tasks new developers within an open (source) software project complete for their initial contribution and what the experience was during this initial contribution. *RQ4* considered whether or not new developers were actually completing issues labelled as *good first issue*s, but it is useful to consider what types of tasks new developers are contributing to as their initial commit within repositories. This may show a divide between suggested tasks and actual implemented tasks, perhaps highlighting improvements for the types of tasks given as suggested first contributions.

RQ5

What types of contributions are made by newcomer developers within a project?

Evaluating the newcomer developer experience within open (source) projects can also provide valuable insights as to the *good first issue*s and whether or not they assisted the newcomer with their goal of finding a suitable primary contribution. These insights allow greater understanding of non-technical experiences of the newcomer developer, such as their personal feelings and ease of use.

RQ6

How do (new) developers perceive the labelling of *good first issues* and their usefulness?

1.4 Contributions

This thesis report and the research performed lead to a number of contributions within the domain of task recommendation and developer onboarding. These contributions are outlined in this section to provide an overview of what can be expected from this report.

***Good First Issue* Dataset:** A dataset has been constructed, sampled, and analyzed which focuses on *good first issue*s and initial contributions. This dataset, spanning 105 repositories, contains 858 sampled **good first issue**s and 1.272 sampled commits. The dataset and its associated analysis are publicly available and presented for future work on *Github*[13]. Additional information is provided in Appendix B.

Task Classification Taxonomy: This report introduces a taxonomy that can be utilized to classify contributions, commits, and issues for future research. The classification consists of six labels that are broad enough to support multiple types of research and domains.

***Good First Issue* Adoption and Fulfillment:** Numerical data was obtained and investigated to determine the rate at which repositories adopt *good first issue*s and how many of these issues are fulfilled by new developers within a project or repository. A recommendation based on this data is made for projects to adopt task recommendation in the form of *good first issue*s as a result of these findings.

Analysis of Issue and Commit Differences: An analysis investigating the differences between initial contributions and *good first issue* task recommendation was performed, identifying that a discrepancy for multiple task types were identified. Additionally, combinations of labels for tasks and contributions were identified to determine whether associations exist between labels within the taxonomy. A recommendation is made for project maintainers to suggest certain tasks more often as *good first issue*s based on these results.

[13]The dataset is available at: `https://github.com/dalderliesten/Good-First-Issue`

Analysis of Developer Experience: Developer experience was investigated and considered to determine whether it made an impact on the type of contributions that would be made. Findings indicated a significant difference between *novice* developers with less than a year of demonstrated experience and other developers, the results of which are presented.

Developer Survey on Usefulness: A survey that was approved by *Delft University of Technology*'s human research ethics commission was sent out to sampled developers to obtain the developer perception of task recommendation and *good first issue*s. The findings and data obtained are presented in this report.

1.5 Structure

This thesis report begins by providing an overview of related work in Chapter 2. After this, the methodology for the research is outlined in Chapter 3. The results and their associated analysis are presented in Chapter 4, and an discussion of the research is provided in Chapter 5, after which conclusions and possibilities for future work are given in Chapter 6.

Chapter 2

Related Work

This section aims to provide an overview of existing literature and work related to *good first issue*s, developer onboarding, and other related topics within software engineering. This is done with the goal of providing the required background such that the state of the art and relevant knowledge can be understood and provide additional context regarding the contributions of this thesis report. Developer onboarding is discussed due to it being the focus of the creation of *good first issue*s, task recommendation is discussed due to its close relevance to the suggestion of *good first issue*s, and broader software engineering concepts are discussed based on their relevance to open source newcomer processes and procedures.

Due to the broad number of topics related to this thesis work and to *good first issue*s, each section in this chapter indicates the topic to which those related works fall under as to act as a classification. These topics include developer onboarding in Section 2.1, task recommendation for development practices and its effectiveness in Section 2.2, and broader software engineering topics related to open-source software development in Section 2.3.

2.1 Developer Onboarding

Developer onboarding refers to the process of actively finding and involving new developers within a project, with the goal of increasing the total number of developers working on an open (source) project. Multiple aspects related to developer onboarding have been studied, a few of which are presented in this section. Section 2.1.1 identifies social factors which affect developer onboarding, Section 2.1.2 discusses technical factors influencing developer onboarding. Additional *alternative* methods or other insights into developer onboarding are given in Section 2.1.3. An overview of all studied papers and works related to onboarding presented in this section are given in Table 2.1.

2.1.1 Social Factors affecting Onboarding

When considering how and whether developers join new projects, the social factors influencing these decisions play an important role as it can make up almost 50% of a newcomer's active time [2]. If a developer feels enticed to join a project and contribute, they may expect

Paper	Year	Aspect Studied	Onboarding Effect
Begel and Simon [2]	2008	Mentoring	Positive
Casalnuovo et al. [5]	2015	Prior Social Contacts in Project	Positive
Dabbish et al. [9]	2012	Repository Activity & Recency	Positive
Dabbish et al. [9]	2012	Signals of Appreciation from Developers to Newcomer	Positive
Fagerholm et al. [13]	2014	Mentoring of Newcomers	Positive
Hahn et al. [22]	2008	Personally Knowing Maintainer	Positive
Kosa and Yilmaz [29]	2016	Gamification	Uncertain
Labuschagne and Holmes [30]	2015	Onboarding Programs for Initial Contribution	Positive
Labuschagne and Holmes [30]	2015	Onboarding Programs on Short-Term	Positive
Labuschagne and Holmes [30]	2015	Onboarding Programs on Long-Term	Negative
Liu et al. [35]	2018	Recommendation of Repositories	Positive
Pham et al. [41]	2016	Automated Testing Requirements	Negative
Steinmacher et al. [47]	2015	Getting Slow Answers to Questions	Negative
Steinmacher et al. [47]	2015	Getting No Answer to Questions	Negative
Steinmacher et al. [47]	2015	Bad English Background	Negative
Steinmacher et al. [47]	2015	Impolite Communication	Negative
Steinmacher et al. [47]	2015	Newcomer Technical Background Expected	Negative
Viviani and Murphy [52]	2019	Code Reviews	Positive
Viviani and Murphy [52]	2019	Mentoring	Positive
Wang [54]	2012	Bug Search Tool	Positive

Table 2.1: An overview of related works to onboarding along with their aspect studied and the effect of that aspect upon newcomer onboarding.

that certain mentoring or social contacts exist and meet their expectations. This section aims to address social factors that may influence the onboarding process.

Casalnuovo et al. [5] studied developer onboarding on *Github* related to social aspects and overall developer productivity. They found that developers tend to join projects in which they have prior social contacts, such as developers they have collaborated with in past projects. Developers also exhibit a greater level of productivity when there is both technical and social familiarity with other contributors for an open-source project, yielding up to 54.3% increases in developer productivity. In contrast, if a developer is referred only on social connections and has no technical background in the project they contribute to due to social connections, overall productivity decreases by 9.6%. The authors suggest that engaging new developers in a meaningful manner is essential to increasing productivity, and that social connections are not sufficient on their own to ensure developer productivity. Their findings are complemented by a study from Hahn et al. [22], which found that developer onboarding was more likely to occur when they personally know a maintainer or the creator of the project.

Dabbish et al. [9] studied the effects a user's activity has on the perception from an open source development perspective. They found that four cues were commonly utilized to infer social information between users. When a contributor was active within an open source repository it is commonly inferred that the contributor is interested in the project and activity is seen by the community as a commitment. This cue is impacted by recency, meaning active users from the past are seen as less "*involved*" than current active users. The intentions of contributors were derived from their sequence of actions over time, meaning that contributor commits could be analyzed over time to derive their intentions and desired direction for a project. Project-wide cues were also identified. It was found that the number of people contributing to a project signified the importance of the project, in which contributors were more likely to see an open source project as "*important*" if a greater number of contributors and interested developers were involved in the project, either submitting bug and issue reports or actually contributing new code to the codebase. They utilized this information to identify a number of considerations open source projects should make when managing their repository. Transparency of the project in *Github* (such as issues and discussions) had a positive impact on contributor project perception, and when a limit of transparency was encountered (such as a lack of clarity in guidelines) communication arose within the project. These communication moments usually resulted in a desire to reach a mutual agreement or compromise regarding segments of the project that are not transparent or clear. Dabbish et al. [9] also identified that, within large open source repositories where transparency and interaction isn't always possible, users started showing "*signals of attention*" such as notifications and small comments of appreciation to indicate their interest in the work that a contributor was doing. This increased the sense and perception of combined social effort within an open source repository.

Steinmacher et al. [47] studied social barriers and challenging aspects faced by new contributors when making their first contribution(s) to open source software repositories and proposed for the creation of a *barriers model* to help alleviate these issues. This model was devised through a meta-analysis of other works. They found that new contributors faced issues related to reception within a new open source project such as not receiving an answer (questions on fora or question sites go unanswered) occurring and causing new contributors to not return. They found an identical result when finding many delayed answer, in which a slow pace of communication within an open source software project scares newcomers away. It was also found that impolite responses to new contributor questions or suggestions also scared away new contributors, partially attributed to new contributors being unsure how to respond to rude or snide remarks. The proposed solution to these types of problems is automated answer/feedback for newcomer contributions and first-time contributions. Additionally, shyness due to cultural differences (such as the perception of being unwilling to request assistance from those seen as *higher ranked*) or a lack of an English language background also decrease contributor onboarding [47]. These barriers reduce the likelihood that a contributor will join the open source community around a project repository.

Fagerholm et al. [13] studied the effect of mentoring within open source software projects and the impact this has on the onboarding of new contributors. They studied an instance in which mentors were assigned to new contributors at a hackathon that last multiple weeks such that the mentors provided recommendations for tasks that should be done, providing an overview of the given codebase's architecture, and guiding the participants in a manner such that they would perform the most simple tasks first. The mentors also assisted with the creation of (automated) test cases, assisting with minor bug fixes, and providing assistance when a participant got stuck. Relevant metrics that existed on *Github* were then utilized to identify the usefulness of the mentors. It was found that the activity and level of engagement with the codebase of the mentored developers was significantly higher than a sub-group of users in the hackathon that did not have mentoring. The suggested increase to onboarding is an activity level that is boosted for close to 300% when mentored versus those that are not mentored. The authors also found that although mentoring greatly increases the onboarding of new contributors, it does cause a reduction in productivity of mentoring developers, who need to dedicate their time to mentoring others. The usefulness of mentoring for new developers was also studied within closed-source projects for smaller sized companies by Viviani and Murphy [52], who found that doing pair programming with an experience *buddy* mentor would help newcomers onboard at a quicker rate and led to greater newcomer satisfaction with relation to the onboarding process. When evaluating more experienced developer that are new to a company, mentoring also has a positive effect on their onboarding as per research from Begel and Simon [2].

Labuschagne and Holmes [30] studied the effects of onboarding (task referral) programs within open-source software repositories to identify whether these programs resulted in additional contributions from newcomers. This was achieved by studying *Mozilla*'s[1] open-source onboarding programs, evaluating contribution attempts by developers within the projects. It was found that a newcomer's likelihood of having an initial contribution that is accepted by the project increases by between $8-13\%$ when an onboarding program exists. However, when investigating the rate of attrition of developers over time and their participation in an onboarding program, it was found that only 47,2% of coached newcomers make a secondary contribution, as opposed to 61,2% of developers that did not participate in such a program. The researchers also identified that dropout rates increase

[1]Additional information can be found at: `https://www.mozilla.org/`

when developer participate in an onboarding program. Their conclusions identify that although onboarding programs assist with short-term developer success, they often result in less long-term contributions.

As a result of these works, it can be stated that social factors which positively effect developer onboarding consist of having prior contacts within a project [5][22], ensuring that activity takes place within the project repository such that external perception is one of active development [9], answering questions that new developers ask within a development's issue or question base [47], and assigning experienced mentors to new developers within a project [2][13][52]. Negative social modifiers for developer onboarding include long delays for newcomer support queries [47], and shyness as a result of cultural differences or a lack of confidence in a developer's ability to communicate in English [47]. Although onboarding program are usful for short-term developer success, they have been found to lead to a long-term loss of developer activity [30].

2.1.2 Technical Factors affecting Onboarding

Technical factors influence the ability of a developer to contribute to a project, and identifying the impact and types of technical factors may help increase developer onboarding. A developer that has the required technical background or knowledge for a particular project is more likely to contribute to that project. Steinmacher et al. identified that newcomers were expected to have sufficient technical knowledge to contribute to the project from the viewpoint of the open source community [47]. This would sometimes manifest itself as unclear or non-meaningful messages and forms of communication occurring between new contributors and the community around a repository. Steinmacher et al.'s findings do not mean that developers are unwilling to seek out new challenges, however, with Hahn et al. [22] finding that developers aim to seek variety in the types of projects that they join regarding programming languages. The researchers state that developers are likely to join projects that rely on a language they have no experience with, but only during the more inexperienced stages of their developer career.

Pham et al. [41] identified that inexperienced developers view automated testing as a waste of time and are not likely to align with requirements for open-source projects and professional development efforts. Although not directly related to developer onboarding, the onboarding of inexperienced developers often requires an alignment to take place between developer contributions and project standards, and many projects maintain standards that require tests to be contributed alongside new feature or enhancements, such as in the *React*[2] and *Node.JS*[3] projects. This may result in a perception of difficulty to contribute to a project, resulting in a technical hurdle negatively affecting the chance a developer contributes to the project.

Viviani and Murphy [52] found that high-quality code reviews within organizations helped onboard developers and help disseminate technical knowledge regarding the project

[2]Additional information can be found at: `https://reactjs.org/docs/how-to-contribute.html#development-workflow`

[3]Additional information can be found at: `https://github.com/nodejs/node/blob/master/doc/guides/contributing/pull-requests.md#the-process-of-making-changes`

at a more effective rate than providing technical documentation to newcomers. Instead of overloading a newcomer with a lot of technical information, issues related to their contributions and an understanding of the codebase was provided in smaller segments when performing code reviews by making remarks about the contributed code. The authors also highlighted that newcomer pull requests in both open-source and private closed-source projects are evaluated differently, often with a greater focus on technical components, positively impacting the onboarding process.

Upon analysis of these works, it can be stated that technical factors that positively impact newcomer developer onboarding include having the required technical background and expertise needed for the project [47], a willingness to learn new technical requirements for a project from the developer standpoint [22], and the existence and dissemination of information through code reviews [52]. A negative factor that was identified was the requirement of contributing automated tests [41], as newcomer developers tend to perceive this as a waste of time and are thus unwilling to contribute.

2.1.3 Other Work Related to Onboarding

Additional work related to developer onboarding was identified during the literature review process. Due to their widespread nature, the works are presented in this section as a contained suggestion or finding.

Kosa and Yilmaz [29] studied the idea of employing gamification to onboard new developers to open source repositories. Gamification relies on the idea of utilizing game based functionality, such as awarding points and achievements, to developers when they perform certain actions. Kosa and Yilmaz propose the utilization of a framework that incorporates the six relevant domains of software development gamification (identifying business objectives, defining target behaviors, identifying the collaborators defined as "*players*," receiving activity identification and looping, adding a fun factor to motivate development, and having access to appropriate deployment tools) to encourage developers to remain engaged throughout a project during initial contribution phases. Despite providing this concept of onboarding, the authors provide no evidence that their framework or gamification method could yield additional engagement of initial contributions from new developers.

Wang identified that [54] a technical solution focusing on developer support tools can help positively impact developer onboarding. Wang identified that tools which supported the searching of bugs and issues that are similar to what a newcomer is currently experiencing, alongside a visual exploration of dependencies within the project's file network. Allowing newcomers to do bug search queries provides them with a simple point of entry for comparative analysis, perhaps providing them with a solution to their problem without having to directly confront members of the development or maintenance team. The package overview allows newcomers to see which dependencies may be necessary to develop or contribute within the project. Wang implemented these features into *Tessaract* [54] and found that developers that utilized bug querying were more likely to identify a related bug at a much faster rate than those without the ability to perform this search, thereby increasing the chance of making a contribution.

Liu et al. [35] developed a recommender system which employs a neural network for list-based ranking (*NNLR*). This recommender system aims to identity and suggest repositories that developers can contribute to based on their preferences. This is done by identifying nine project features, such as prior social connection based upon past contributions, company associations between new developer and company maintaining the project, and programming language matches between project and developer knowledge, and integrating an *NNLR* based upon those features. The authors found that by employing this recommender system a total of 2.044 onboarding occurrences took place in which developers contributed six or more commits, outperforming baseline expectations and indicating that recommender systems can help in newcomer onboarding.

2.2 Task Recommendation

The recommendation of tasks ensures that developers within a (open-source) project can provide contributions that a project requires, and can help prevent task duplication from occurring. This results in a greater level of efficiency and progression among the desired goals for a project. Alongside this, tasks can help onboard new developers by providing a manner of task suggestion, perhaps in the form of *good first issue*s, that can show them tasks suitable for their level. To gain insights into the effectiveness of task recommendation, multiple aspects and types of task recommendation must be studied and investigated. This section provides an overview of work related to studying the effectiveness of task recommendation. An overview of literature reviewed and presented regarding the topics of task recommendation is given in Table 2.2.

Gasparic and Janes [18] performed a literature review covering 46 papers that tackled the topic of task recommendation in 2016 to obtain an overview of suggestions and best practices. They aimed to identify four key factors, consisting of the performance expectations and its impact on task recommendation, whether effort had an impact upon suggested tasks, the social influences of recommendation, and other conditions that surround suggestions. An array of minor factors influencing task recommendation were found, such as publicly available source code examples making recommended tasks easier, a solid software review process, and the possession of sufficient knowledge for task completion. Among their most important findings, however, the major identified factors that impact task recommendation positively and were employed within task recommender tools were found to be product-related information, such as necessary source code modifications to ensure task completion and personalized source code component recommendations based upon personal developer interest (a developer that worked on the networking components before might enjoy additional networking tasks). The authors identified that additional research is required before conclusive evidence can be made regarding which information leads to the most optimal task recommendation schemes. They also mentioned that only a single tool focused on utilizing *testing* tasks for recommendations of tasks, meaning tool support for task recommendation should be created with a focus on testing. The authors also state that no clear factors for a broader context were identified, indicating that additional research is needed to identify what creates effective task recommendation within the software develop-

Paper	Year	Aspect Studied	Indicated Usefulness
Anvik and Murphy [1]	2011	Recommender Aspects	Developer availability positively influences recommendations
Anvik and Murphy [1]	2011	Recommender Aspects	Developer interest positively influences recommendations
Anvik and Murphy [1]	2011	Recommender Aspects	Codebase location positively influences recommendations
Gasparic and Janes [18]	2016	Broad Recommender Systems	Almost no tools rely on testing
Gasparic and Janes [18]	2016	Broad Recommender Systems	Additional research is needed to identify relevant aspects
Li et al. [32]	2016	Social Task Recommendation	Coupling social active/inactive developers improves activity
Mao et al. [37]	2015	Crowdsourcing Tasks	Increased Efficiency (50-71%), Increased Diversity (40-52%)
Zhou et al. [57]	2010	Task Centrality	Central Tasks have a Greater Completion Rate
Zhou et al. [57]	2010	Task Complexity	Simpler Tasks lead to greater Fluency

Table 2.2: An overview of related works to task recommendation along with their aspects studied and their indicated usefulness based upon the research provided.

ment context.

Mao et al. [37] studied the overall effectiveness of providing tasks for *crowdsourcing* to developers to prevent an overload of information from occurring based upon the perspective of the developer. The authors argued that two focuses for task recommendation should include ensuring that the correct developers are assigned to the correct tasks, and that the developer assigned is reliable such that the project can assume the task will be completed. The research was completed through study of the *Topcoder*[4] platform and extraction of features relevant for a contributor, such payment for completion of task, task description, and date provided. By providing a recommender system that took these categories into consideration, they aimed to get a greater diversity of tasks shown per developer, and to increase the overall completion rate of tasks. By implementing such a recommender system versus a control group which did not have such a system, it was found that that the completion rate of tasks (stated as efficiency) was increased compared to the control baseline with values between 50-71%, and increased the diversity of tasks suggested to developers with values of 40-52%. This indicates that task recommendation improves completion rate of tasks when considering a possibility of paid tasks and when compared to no recommendations being made.

Li et al. [32] performed an analysis over Chinese development platform *JointForce*[5] with the goal of identifying whether socially-driven recommender systems would yield good results. Their study employed a constructed graph network that would recommend software projects based on the degrees of social contacts they have within each project, and then recommended tasks be completed based upon projects with the greatest degree found. They also augmented this method by recommending developers that have stopped actively contributing to the project to be coupled with their social contacts that had taken up tasks, as to entice inactive developers to participate again. Their findings indicated that these methods led to an decrease of 16,7% of inactive developers within the projects they sampled upon.

Zhou et al. [57] studied developer fluency and qualifications in the context of tenure at a project. Although not directly relevant to task recommendation, their findings include a number of elements relevant to possible task recommendation. Two aspects, namely task difficulty and task centrality, were found to be important for effective task distribution and recommendation. Task difficulty revolves around the technology required to complete it (such as frameworks or programming languages), the domain for which the task must be done, the number of social connections the developer is familiar with, and the exposure that a developer has to other stakeholders. Task centrality refers to the system-wide impact of the contribution and the future impact, measuring how much a task's completion will impact the magnitude of the overall system. Their findings indicate that tasks which have a high factor of system centrality, meaning tasks that have a high impact on the system, lead to faster completion of tasks and greater developer fluency. They also identified that newer developers with a smaller degree of fluency need to begin with simpler tasks, and slowly grow into complex tasks. As a result, task recommendation should focus on providing important tasks to a repository that are simple.

[4]Additional information can be found at: `https://www.topcoder.com/`

[5]Additional information can be found at: `http://www.chinasofti.com/en/internet-it-service/jointforce.shtml`

Anvik and Murphy [1] studied the usefulness of a bug report repository within a project. By studying recommenders within *Eclipse*, *Firefox*, *gcc*, *Mylyn*, and *Bugzilla* they identified components related to the assistance of good recommendations, which they validated through the use of a survey that was sent to developers. The authors identified that by utilizing a developer recommender that recommends a developer that could fix a bug report, a component recommender which suggests which codebase or product components are related to a report, and an interest recommender that identifies developers that may enjoy fixing issues related to the report, triage times for reports are decreased, with an accuracy of 75% tested upon a sample set. This suggests that a combination of developer interest, availability, and component identification can help recommend tasks for developers.

Based upon these findings, it can be seen that social factors, such as the number of known developers within a project, has a significant impact on effective task recommendation [37][32][57]. However, additional research is needed before a conclusive end-all techniques can be identified to ensure efficient task recommendation [18].

2.3 Broader Software Engineering Topics

When considering the onboarding of developer, broader topics within software engineering can be discussed to both provide a background in efficient strategies for developer onboarding and to identify best practices. This section aims to provide some literature related to these topics and to identify whether (new) developers should be aware of these techniques. Papers related to continuous integration are discussed in Section 2.3.1, the pull-request development model and its related works are given in Section 2.3.2, and works related to developer turnover within software projects is presented in Section 2.3.3. An overview of all papers in this section is provided in Table 2.3.

2.3.1 Continuous Integration

Continuous integration (CI) refers to the practice of committing smaller pieces of code to a repository at multiple instances by multiple developers, along with additional tools such as automated testing and static analysis of the codebase to verify functionality and best practices. A number of papers investigating the effects of continuous integration upon (open-source) software development either through direct mining or through tools such as *TravisTorrentTravisTorrent*[6] are provided in this section.

Vasilescu et al. [50] studied the rate of passing builds within *Travis*[7] continuous integration in open source *Java*, *Python*, and *Ruby* repositories that existed on the *Github* platform. They found that 92,3% of the projects they sampled ($n = 223$) employed *Travis-CI*, although only half of the sampled projects had any associated builds. They also identified that contributions that are given to an open source repository through a pull request are more likely to fail integration tests than contributions made through direct commits, suggesting that

[6]Website can be found at: `https://travistorrent.testroots.org/`

[7]Read more at: `travis-ci.com`

the more rigorous pull request review process is likely to result in greater discussion and dissection of the proposed changes.

Hilton et al. [25] studied the usefulness and overall effectiveness of utilizing continuous integration. They found, based on an analysis of 34.544 open-source repositories on *Github* and a survey of 442 developers, that about 40% of the analyzed repositories utilized continuous integration, with *TravisCI* being the largest utilized continuous integration provider. It was also found that projects that are more popular (as gauged by *Github* star figures) are more likely to use continuous integration, and that the popularity of a programming language or language itself does not impact likelihood to utilize continuous integration. In projects that did not use continuous integration, it was found that this was mostly due to developer unfamiliarity with continuous integration and potential costs associated with training developer to utilize it. Regarding the benefits of continuous integration, projects utilize continuous integration to identify bugs early and reduce concerns about potential build-breaking issues that could lead to deployment issues. It was also found that continuous integration is not perceived as useful for debugging or identifying issues within the codebase. Regarding the pull-based development model, it was found that using continuous integration fastens the medium pull request acceptance rate by 1,6 hours, and helps avoid breaking the build by not merging problematic pull requests into a project. Yu et al. [55] found a cost for this improvements, however, and identified that continuous integration increases pull request response time by 16%, since integrators are likely to wait for continuous integration to complete before evaluating any component of a pull request.

Another paper by Vasilescu et al. [51] studied the productivity impact of continuous integration on developer productivity and software quality. This was done through a numerical analysis of numerous repositories that utilized continuous integration in at least 25% of their pull requests. They compared these results to a quality measurement consisting of the number of bugs per unit time. It was found that each integrator added to a project allows for 23,6% additional pull requests to be merged per unit of time, while also increasing the number of pull requests that are rejected. They also found that teams utilizing continuous integration are more effective at merging pull requests submitted by core developers, and that the availability of continuous integration is associated with external contributors having fewer rejected pull requests. Continuous integration also positively impacts bug reports, with repositories relying on it seeing a 48% increase in bug reports during development that lead to eventual fixes. Overall, bug detection and reporting is also increased when utilizing continuous integration, but it was found this came without a penalty or cost to external software quality, since non-integrator contributors do not experience an increasing number of defects related to compatibility or from local development.

Fitzgerald et al. [14] studied the trends within continuous integration in 2014. The authors identified that, at a business logic level, continuous integration fulfills the role of development and deployment, but it does not adequately do so. The authors claim that, despite its usefulness, that continuous integration does not adequately cover changing factors within the development process. For example, initial tests and development my be heavily focused upon fulfilling technical requirements, but not necessarily on filling end-user or business requirements. The authors state a better alternative must be found to continuous integration and other *DevOps* processes that better combines the business requirements, along with a more precise definition of what should be tested for each feature or branch.

Elazhary et al. [12] studied the effect of having continuous integration validate that non-functional requirements related to contribution guidelines are met within a commit. By analyzing 53 projects mined through the *GHTorrent* [20] tool, they aimed to identify if projects that use continuous integration would include checks for these contribution guidelines, and whether these guidelines matched the rules loaded into the continuous integration. The authors found that although 72% of projects employing continuous integration tend to have rules and checks for aspects related to code style and technical information, many projects do not check or validate contribution procedures, with only 31% mentioning continuous integration in their guidelines. Examples mentioned 51% of projects that would allow the re-opening of existing issues and 68% of projects that allowed the re-opening of pull requests, without mentioning this in their contributing guidelines. They also found that contributing documentation often did not explain the workflow or integration of continuous integration within a project's processes. Their overall concluding remarks state that contributing guidelines appear to be written more to suit existing project maintainers as opposed to newcomer developers.

Beller et al. [3] studied failure causes of *Travis CI* build processes in 2.640.825 *Java* and *Ruby* projects on *Github* and how testing-related tasks are performed within the context of continuous integration. They achieved this by extracting *Travis* projects from *Github* projects, and employing the SHA1 hash of a *git* commit to obtain relevant *Travis* information. Their findings indicate that, with a utilization rate of *Travis CI* among *Github* repositories of 31,1%, failing tests are the single dominant reason for builds failing within continuous integration. They also found that many projects ignore an occasional failing build, but do not tolerate successive failing builds over time.

The given literature indicates that continuous integration leads to additional pull request discussions and dissection [50], and increases overall code quality. However, it was also found that this comes at a cost to developer efficiency [51], slows down pull request response times [55], may not be written sufficiently outlined in newcomer documentation [12], and may not be sufficient to fulfill all business requirements related to testing and static analysis within an organization [14]. It can also be stated that testing is the main cause for failure within continuous integration [3].

2.3.2 Pull-Based Development Model

The pull-based development model requires developers within projects to make the required changes, and then allow them to be reviewed in what is known as a *pull request* (PR). Literature related to both the efficiency and some best practices are given within this section.

Gousious et al. studied the integrator perspective of the pull-based development model [21]. Integrators are responsible for managing and integrating contributions from non-integrator participants. Their investigations aimed to identify how integrators utilize code reviews to merge contributions, specifically aiming to identify the criteria integrators utilize to evaluate the quality and acceptance decision of a contribution. Multiple findings were found, including that integrators prefer having metadata about merged pull requests, and 75% of the surveyed integrators ($n = 749$) indicate they review all pull requests. They also found that integrators are most likely to merge pull request contributions if the code quality is high and conforms to the agreed project style, include tests, and adhere to the project architecture. Contributions are prioritized by cruciality (meaning bug fixes are more important than new features), urgency, and size. Integrators do state, however, that challenges include maintaining the quality due to the number of contributions, and motivating contributors to keep contributing to the project. Integrators also state issues reaching consensus about decisions made in pull requests and communicating these decisions to contributors.

Gousious et al. also studied the contributor's perspective of the pull-based development model [19]. Contributors are users that contribute new code to the codebase of an open source repository. The research aimed to study *top*-contributors and determine what motivated their contributions to pull-based open source repositories, how these contributors prepared for a contribution, and what they perceived as challenges related to the pull-based software development model. Many contributors stated that they contribute to projects due personal interest or them utilizing said project, but also due to greater employ-ability through the enrichment of their curriculum vitae (CV). The latter motivation has become known as *career concern* [19]. Contributors stated they evaluate their contributions based on compliance (to what degree does a contribution meet the requirements of the project defined in, for example, a contribution guidelines document), code and commit quality, and how well tested their contributions are. Top contributors noted that many challenges they face from the pull-based development model include difficulty complying with the many different requirements between projects, and not being able to keep up with the social requirement for a project (such as communication form and what to communicate about). They also state many contributors struggle to understand the code base, and that new contributors are appreciative of well-defined guidelines regarding contribution and appreciate responsive projects, ensuring that their submissions are active.

Yu et al. [55] studied the factors which affect pull request response time and latency in the completion of the handling of pull requests in pull-based development models. This research analyzed 103.284 pull requests for this purpose. They identified that factors causing the largest direct delay in pull request responsiveness include the amount of discussion related to the pull request (with more discussion causing a greater delay in the handling of the pull request), a greater number of total commits in a pull request having a negative impact on response time, and the total lines of code impacting the time taken to evaluate a pull request. It was also found that pull requests making changes to tests or core components of a repository yield higher response latency. Yu et al. [55] also identified numerous social and non-technical factors affecting pull request response time, including the social connections that a contributor has to the open source project. If a contributor engages in social activity with maintainers or integrators of a project, their pull request is likely to have a smaller response time and be dealt with in a more rapid fashion. They also identified that integrators with a larger workload (both related to the repository itself and their own personal responsibilities as an integrator) negatively impacts pull request response time.

Saito et al. [44] tried to identify what developer experiences were with the *Git*-tools related to pull-based software development. The authors found that, within a survey of 1552 developers, the majority found the pull-based development model to be functional and felt like they had a good grasp of the *Git* commands, but many stated they struggle with the act of *rebasing*[8], possibly pointing to additional resources that could be created to allow developers to practice *rebasing* and to obtain a better grasp of it.

Based upon this literature, it can be stated that the pull-based development model manages to derive contributions based upon personal interest [19] and to enrich their curriculum vitae as to appear more attractive to potential partners and employers [19], but induces challenges for contributors related the differing requirements per project for contributions [19] and being unable to keep-up with changing social requirements [19]. Integrators indicate that 75% of them look at all submitted pull requests to a project [21] and that high code quality resulted in faster acceptances and merging of pull requests [21], whereas discussion surrounding a pull request or a greater number of commits within a pull request has a negative impact on response time [55]. It was also identified that developers tend to have a good overview of *Git* functionality, but struggle with the *rebase* command [44].

2.3.3 Developer Turnover

Developer turnover refers to the occurrence in which existing developers within a project decide to leave the project or become inactive. The knowledge and experience they leave behind may be of importance to the project, and methods of ensuring no single developer takes too much knowledge with them are important. This section provides literature on developer turnover and its consequences.

Lin et al. [33] studied the impact of developer turnover within large open source software projects and the reason as to why this occurred. The authors made an analysis of five large open-source software repositories and projects, analyzing over 1.5 million commits and, after filtering, more than 8.000 contributors. They found that developers that are relatively new to the projects are less likely or willing to continue contributing to the project as opposed to earlier developers. They also found that developers who contribute to both their own files and files created by others are more likely to stay within a project as developers who focus exclusively on either their own contributions and files or other's files. Other factors positively influencing developer continuation and onboarding include maintaining files as opposed to only creating new files, and that developers who mainly contribute code (as opposed to documentation or testing) also have higher *survival rates*. The authors suggest balancing collaboration with individualism, ensuring new developers also perform maintenance tasks on existing parts of the codebase, and that some coding tasks should be assigned to contributors that focus mostly on documentation.

Foucault et al. [16] studied the turnover in open-source software and the impact of this upon the project. By employing metrics related to developer turnover, such as external/internal turnover & contributors that stayed between multiple measurements, the authors constructed a dataset and identified numerous aspects of turnover at both a developer and project level. Findings included that contributors are active and remain active within a project are developers that are paid by a company to use the project or develop that project in most cases or act as consultants, with some outlying developers contributing without further financial motives or reasoning. The impact of developer turnover upon software

[8]Additional information can be found in the *Git* documentation: `https://git-scm.com/docs/git-rebase`

quality was found to exist, in which developer turnover negatively impacted the density of bug fixed. Projects in which existing contributors left the project would see slower and less bug fixes.

Cortazar et al. [26] studied the knowledge loss related to software development that occurs during developer turnover. When a (senior) developer leaves a software project a possible loss of knowledge can occur that can negatively impact the project. By selecting four open-source software projects and their associated repositories (*Evolution*, *GIMP*, *Evince*, and *Nautilus*), the authors studied the *orphaning* of lines of code within existing repositories. The authors defined orphaning as lines of code whose original contributor or developer had since left or stopped actively contributing to the project. It was found that, when a lead developer leaves, significant parts of the codebase become orphaned, thus negatively impacting development as the knowledge about these lines of code is gone. They also found that in the case of *Evince*, an effort was undertaken to un-*orphan* major lines of code, which worked, until another developer left, once again orphaning large parts of the code. The overall conclusion states that code orphaning can lead to productivity losses among developers, with possible solutions lying in a maintenance team which specifically aims to address and simplify difficult or orphaned code.

Based upon these findings, it can be stated that developer turnover occurs due to new developers being unwilling to integrate within the project [33], and due to certain developer making mostly non-code contributions and not being as heavily invested in the project as those that do contribute source code [33]. It was also found that paid developers from companies tend to remain active in open-source repositories longer than volunteers [16]. When considering the effects of developer turnover, it was identified in the literature given that developer turnover impacts the rate at which bugs are fixed [16], lowers software quality [16], and slows overall development down due to a loss of knowledge [26] and code that becomes orphaned.

Paper	Year	Aspect Studied	Findings
Beller et al. [3]	2017	Continuous Integration	Tests are the main cause for build failure.
Beller et al. [3]	2017	Continuous Integration	Most projects don't mind one-time build failure.
Elazhary et al. [12]	2019	Continuous Integration	72% of projects using CI check static analysis
Elazhary et al. [12]	2019	Continuous Integration	Not clear from documentation how CI fits into project processes
Fitzgerald et al. [14]	2014	Continuous Integration	Tests do not match business requirements over time
Gousious et al. [21]	2014	Pull-Based Development	High code quality increases PR merge likelihood
Gousious et al. [21]	2014	Pull-Based Development	75% of integrators view all PRs
Gousious et al. [21]	2014	Pull-Based Development	Integrators prioritize bug fixes over new features for PRs
Gousious et al. [19]	2016	Pull-Based Development	CV enrichment driving factor for contributors
Gousious et al. [19]	2016	Pull-Based Development	Contributors evaluate compliance, code quality, tests
Hilton et al. [25]	2016	Continuous Integration	40% of repositories used continuous integration
Hilton et al. [25]	2016	Continuous Integration	Number of *Stars* increases CI adoption
Hilton et al. [25]	2016	Continuous Integration	Developer training/costs largest reason to avoid CI
Saito et al. [44]	2016	Pull-Based Development	Developers struggle with *rebasing*
Vasilescu et al. [50]	2014	Continuous Integration	92,3% of projects use *Travis-CI*
Vasilescu et al. [50]	2014	Continuous Integration	Pull-Request commits more likely to fail integration tests
Vasilescu et al. [51]	2015	Continuous Integration	23,6% pull request increase per integrator
Vasilescu et al. [51]	2015	Continuous Integration	Positive impact on number of bug reports
Yu et al. [55]	2015	Continuous Integration	CI increases pull request waiting time by 16%
Yu et al. [55]	2015	Pull-Based Development	More discussion / commits / lines of code slow response time

Table 2.3: An overview of related works to broader software engineering topics, the year in which they were published, and additional information per paper.

Chapter 3

Methodology

The methodology section aims to provide an overview of the procedures utilized to perform the research and obtain the results as given in Chapter 4. By outlining the decision made for the research process, and by providing an outline of the tools developed and utilized, the data obtained can be understood and the analysis can be justified. An overview of the procedures and methodology employed for data collection is given in Section 3.1, an overview of the analysis' procedures and methods is given in Section 3.2, and an explanation of the questionnaires that were given to developers is given in Section 3.3. A graphical representation of the procedure outlined in this chapter is given in Section 3.4.

3.1 Data Collection

The collection of data for this research depends significantly on the ability to obtain data from *Github*'s V3 application programming interface (API)[1]. Since the focus of this research lies upon identifying the usefulness of labelling *good first issue*s upon *Github*, a procedure was designed implementing components provided by the API such that data regarding *Github* issues and repositories could be obtained. Naturally, a selection of relevant repositories had to be made due to the inability to access all existing repositories on *Github*.

To guide the data collection process, the selection of repositories is outlined in Section 3.1.1, the collection of data related to the first commits of users within repositories is outlined in Section 3.1.2, the collection of issues labelled as *good first issue*s within a repository is outlined in Section 3.1.3, and an outline of steps taken to streamline and simplify the data collection process is given in Section 3.1.4.

3.1.1 Selection of Repositories

Due to the limitations caused by the *Github* V3 API, such as a rate limit preventing a large number of queries[2] and the inability to access repositories that are not publicly listed or

[1]More information can be found at: `https://developer.github.com/v3/`

[2]Additional information can be found at: `https://developer.github.com/v3/#rate-limiting`

accessible by the investigating entity[3], a selection of repositories for data collection and manual analysis had to be made. The goal of the selection of these repositories was to get a wide and diverse set of project disciplines covered to account for possible differences within domains and to increase cross-dimensional validity. The sample size also had to be representative and feature projects of multiple sizes (small projects with $<$1.000 commits and larger projects with $\geq$ 1.000 commits), differing repository domains (such as graphics, web development, or machine learning), and projects that are maintained both by a private community (meaning the development is not steered by a large company, such as *Microsoft* or *Facebook*) and a company-steered project.

To accommodate the large range of factors influencing the selection of repositories, it was decided to utilize a combination of *accidental sampling* based on suggestions given by *Github*'s trending repositories category, and *critical case sampling* by utilizing *Stack-Overflow*'s[4] developer surveys. *Github*'s trending repositories page, as shown in Figure 3.1, provides an overview each day with repositories that see increased activity from the community. The method utilized is not public[5], but this method provides a somewhat random sampling of active repositories that ensures a diversification of domains covered by the research due to its random nature. The data from *StackOverflow* provides a survey of projects that developers actively use. Due to the high engagement rate of *StackOverflow* and the broad diversity within its visitors [28], *StackOverflow*'s annual developer survey provides a wide overview of projects to sample from, ensuring a greater level of representation for different domains and project types. Additional repositories were sampled from the latest annual survey available at the time of performing this research, which was the 2019 edition[6]. These projects were added as an addition to those sampled from *Github*'s trending page.

As a result of the selection of repositories, a total of 105 repositories were identified for analysis[7], of which 46 repositories were found to utilize *good first issue*s in some manner or form, including tags that were not called "*good first issue*" but had a description that matched that of a task being labelled as good for newcomers or suitable for inexperienced developers[8]. These 46 repositories were then utilized for data mining as outlined in Section 3.1.2 and Section 3.1.3.

[3] Additional information can be found at: `https://developer.github.com/v3/repos/#list-repositories-for-the-authenticated-user`

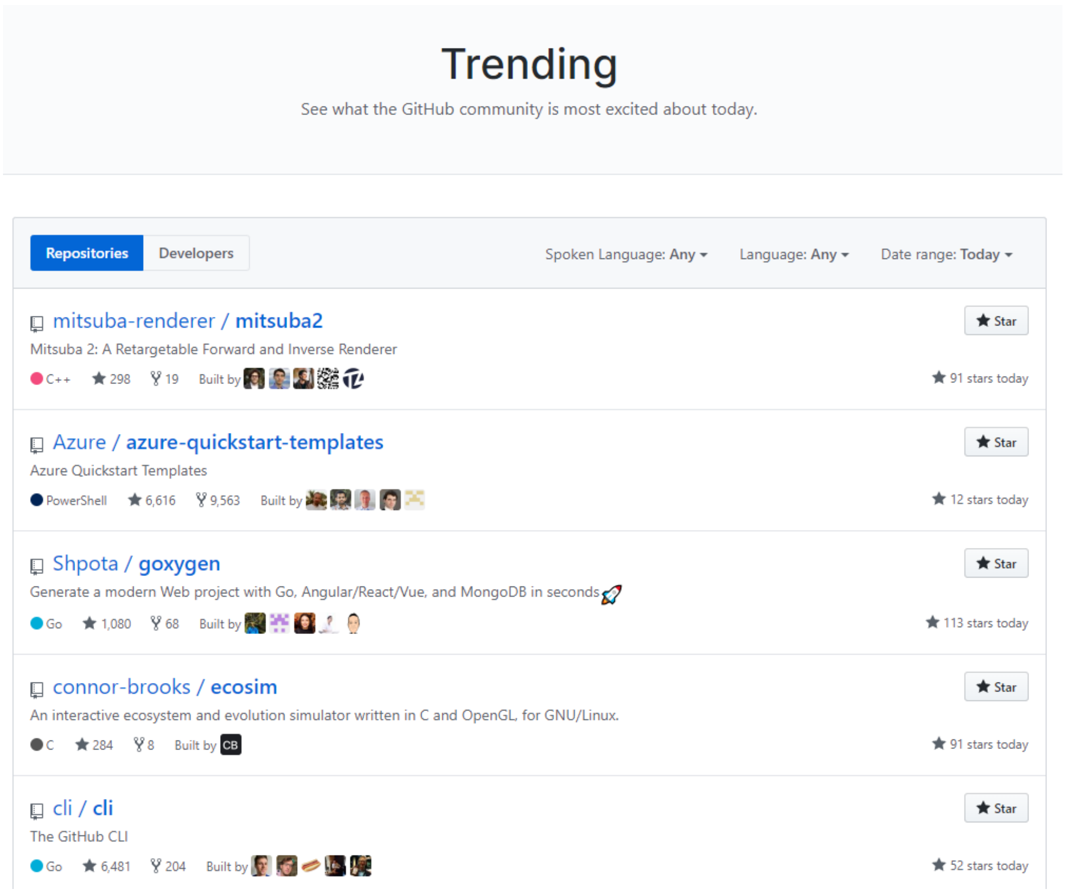
[4] Website can be found at: `https://stackoverflow.com/`

[5] Confirmation can be found at: `https://github.community/t5/How-to-use-Git-and-GitHub/How-github-detect-trending-repositories/m-p/26464/highlight/true#M7517`

[6] Survey can be found at: `https://insights.stackoverflow.com/survey/2019`

[7] A list of all sampled repositories can be found at: `https://github.com/dalderliesten/Good-First-Issue/blob/master/Sample%20Set/Sampled-Repositories.csv`

[8] These alternative labels are listed in the analysis list, which can be found at: `https://github.com/dalderliesten/Good-First-Issue/blob/master/Sample%20Set/Repositories-with-Good-First-Issues.csv`


Trending

See what the GitHub community is most excited about today.

mitsuba-renderer / mitsuba2
Mitsuba 2: A Retargetable Forward and Inverse Renderer
C++ ★ 298 19 Built by ★ 91 stars today

Azure / azure-quickstart-templates
Azure Quickstart Templates
PowerShell ★ 6,616 9,563 Built by ★ 12 stars today

Shpota / goxygen
Generate a modern Web project with Go, Angular/React/Vue, and MongoDB in seconds
Go ★ 1,080 68 Built by ★ 113 stars today

connor-brooks / ecosim
An interactive ecosystem and evolution simulator written in C and OpenGL, for GNU/Linux.
C ★ 284 8 Built by CB ★ 91 stars today

cli / cli
The GitHub CLI
Go ★ 6,481 204 Built by ★ 52 stars today

Figure 3.1: An example of the trending repositories page on *Github*.

3.1.2 Collecting First Commit Data

To collect data related to the first commit of each user, an automated script was written which employs the *Github* application programming interface (API) to obtain all commits for the indicated repository. First, the given repositories' *Git* location must be provided, which can be done within a central method in the application. Once this location has been provided, all commits from the repository are queried and obtained from the API. The order in which the commits are obtained is chronological, meaning the first element in the commit list is the earliest commit, and the last element in the commit list is the latest or final commit in the repository. These commits are then stored within a list, after which an iterator is created such that each commit can be inspected manually.

A separate list is then created for the tracking of the unique users within the repository, which is initially empty. For each of the commits, the author name is inspected and, if the author name does not yet exist within the aforementioned list, the name is stored within the unique user list and the commit found is put within a separate "*first commit*" list structure. Once all commits have been iterated upon, the list with first commits per user is written to a comma separated value (CSV) file for persistent storage, and the list of found commits is returned to the application's caller.

A pseudocode representation of the procedure for the collection of first commit data is provided in Algorithm 1.

Algorithm 1 Obtaining First Commits per User given a Repository *given_repository*

```
$first\_commits \leftarrow \emptyset$
$users\_found \leftarrow \emptyset$

$all\_commits \leftarrow given\_repository.get\_commits()$

for all commit in all_commits do
    if commit.author not in users_found then
        $users\_found \leftarrow commit.author$
        $first\_commits \leftarrow commit$
    end if
end for

return first_commits
```

3.1.3 Collecting *Good First Issue* Tasks

To obtain the tasks that had been labelled as a *good first issue* within a repository, another automated script was written that very closely resembles the procedure for the first commit data outlined in Section 3.1.2. After the repository *Git* location has been given, a tag may be given for a *good first issue*. This is done to allow for flexibility in the type of tag which must be obtained, since not all repositories sampled utilize *good first issue* labels but utilize another label while intending to indicate the same type of task. An example of such a label could be *ASP.NET*'s "*good first contribution*" label.

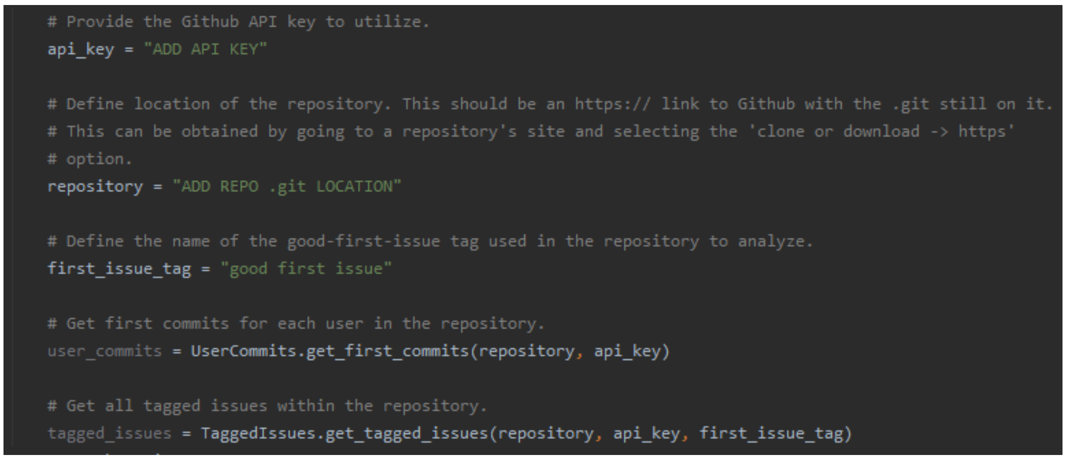
Once these parameters have been given to the script, a call is made to the *Github* API to provide all issues for a repository, and then utilizes the API's *labels* parameter[9] to obtain only the issues that are labelled with the user given label. Since labels within the *Github* API are their own objects, a conversion takes place which validates the existence of the label within the repository, and if so, converts it to the appropriate object. Then, using this label object, the call is made and the list of issues is returned. This list is then stored in a CSV file for persistent storage, and the list of found issues is returned to the caller.

A pseudocode representation of the procedure for the collection of first commit data is provided in Algorithm 2.

[9]Additional information can be found at: `developer.github.com/v3/issues/#list-repository-issues`

Algorithm 2 Obtaining Good First Issues for the Repository *given_repository* and the desired issue label *given_label*

```
$querying\_label \leftarrow given\_label$ string converted to a $label$ object

$found\_issues \leftarrow given\_repository.get\_issues()$

for all $issue$ in $found\_issues$ do
    if $querying\_label$ not in $issue.labels$ then
        $found\_issues \leftarrow found\_issues - issue$
    end if
end for

return $found\_issues$
```

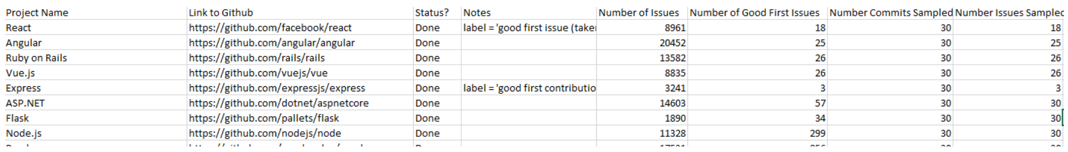


```
# Provide the Github API key to utilize.
api_key = "ADD API KEY"

# Define location of the repository. This should be an https:// link to Github with the .git still on it.
# This can be obtained by going to a repository's site and selecting the 'clone or download -> https'
# option.
repository = "ADD REPO .git LOCATION"

# Define the name of the good-first-issue tag used in the repository to analyze.
first_issue_tag = "good first issue"

# Get first commits for each user in the repository.
user_commits = UserCommits.get_first_commits(repository, api_key)

# Get all tagged issues within the repository.
tagged_issues = TaggedIssues.get_tagged_issues(repository, api_key, first_issue_tag)
```

Figure 3.2: The central entry point, showing the API key, repository location, and desired label to filter upon.

3.1.4 Application Entry Point

To streamline the data collection procedure, and to provide additional flexibility regarding the labels as outlined in Section 3.1.3, a central entry point was made that brings all scripts together as shown in Figure 3.2. This entry point allows for the definition of an API key, the desired good first issue label, and makes all calls. This prevents multiple scripts from having to be executed to obtain the data and for the code to have to be edited internally to support additional labels. The lists are also returned to allow for possible extension in future work.



Project Name	Link to Github	Status?	Notes	Number of Issues	Number of Good First Issues	Number Commits Sampled	Number Issues Samplec
React	https://github.com/facebook/react	Done	label = 'good first issue (taker	8961	18	30	18
Angular	https://github.com/angular/angular	Done		20452	25	30	25
Ruby on Rails	https://github.com/rails/rails	Done		13582	26	30	26
Vue.js	https://github.com/vuejs/vue	Done		8835	26	30	26
Express	https://github.com/expressjs/express	Done	label = 'good first contributio	3241	3	30	3
ASP.NET	https://github.com/dotnet/aspnetcore	Done		14603	57	30	30
Flask	https://github.com/pallets/flask	Done		1890	34	30	30
Node.js	https://github.com/nodejs/node	Done		11328	299	30	30

Figure 3.3: The CSV file containing numerical data, sample status, and the number of issues, number of *good first issue*s, number of sampled issues, and number of sampled commits.

3.2 Analysis of Data

To analyze the obtained data, procedures and analysis templates had to be designed such that an identical analysis process was carried out across all repositories and projects found. The analysis component focused on identifying aspects related to issues and commits that could help differentiate between the type of work being done and the overall usefulness of these labels in practice. To help guide the analysis and to explain the procedure as to allow for reproducability, the procedure for obtaining numerical data for each repository is outlined in Section 3.2.1, the analysis for good first issues obtained through data collection is given in Section 3.2.2, and the procedure for the analysis of the first commits of user in a repository is provided in Section 3.2.3.

3.2.1 Obtaining Numerical Repository Data

Despite having automatically obtained the first commits per user and *good first issue*s from repositories as outlined in Section 3.1, numerical data related to research questions 1 and 2 had to be analyzed manually. Verification of the CSV files had to be done for both commits and issues, since some repositories contained characters or descriptions that would add spaces and adding blank lines, possibly bloating the values of issues and commits. Once these files were verified, the total number of *good first issue*s were obtained by inspecting the associated CSV file, whereas the total number of issues were obtained by navigating to the *Github* repository and inspecting the total number of issues. All this information was stored in a separate CSV file which served as a master list for the sampled repositories, as shown in Figure 3.3. Additionally, the number of sampled issues and sampled commits were noted.

Since the analysis work was manual, and due to the fact that some repositories have tens of thousands of issues and potentially more commits, the decision was made to sample at most 30 issues and 30 commits per sampled repository. The value of 30 was chosen as an arbitrary value, but mostly due to it being considered manageable to manually inspect and analyze that number of issues and commits per project. If a certain project had less than or exactly 30 issues or commits, this was noted in the file as to provide an overview of the spread and size of a particular project. Sampling of issues and commits was done by generating 30 random values within a random number generator, and analyzing issues or commits on the indicated rows.

Column Name	Description
Issue	Gives the name or title of the issue.
URL	Contains the hyperlink to the issue.
Associated PR	Contains a hyperlink to the associated PR or commit for this issue.
Taken by New Developer?	Value of 1 if taken by a new developer, 0 otherwise.
Bug Fix	The issue is related to a mistake or error that needs to be fixed.
Enhancing an Existing Feature	The issue wants an existing feature to be extended with new functionality.
New Feature	The issue requires an entirely new functionality to be created.
Documentation	The issue is related to a change in the documentation.
Testing	The issue is related to the creation of changing of tests.
Refactoring	This issue requires changes to be made that optimize or increase readability.
Other Information	Stores additional tags or relevant information if required.

Table 3.1: An overview of the aspects analyzed for each issue as presented in the CSV template for issue analysis.

3.2.2 Analyzing Good First Issues

To analyze the *good first issue*s that existed within a repository, a standardized analysis template was created which tracked a number of aspects related to the issue's task regarding software development and utilization within the project. This template, taking the form of a CSV formatted file, allowed for manual analysis on at most 30 *good first issue*s per project as outlined in Section 3.2.1. The template tracked 11 aspects of each issue which were utilized to provide an overview of the issue and its intentions. These aspects are given and described in Table 3.1.

Each issue was manually inspected and the issue name and hyperlink (URL) to the *Github* location of that issue were stored as to provide context to the description of the issue and to make clearer what the issue intended. The hyperlink can be utilized to verify or validate the claims made for each issue as to increase reproducability. If the issue was implemented with an associated pull request (PR), a link was given to the pull request to check whether the implementation provided matched the issue's description. Additionally, by inspecting the pull request associated with a *good first issue* it could be validated whether the issue was implemented and taken by a new developer.

To classify each issue, a taxonomy was created which could be utilized to identify the types of tasks that were being suggested as *good first issue*s. This taxonomy focused on the type of contribution that was expected of a newcomer as opposed to the domain in which the contribution was requested. The tracked aspects within the taxonomy consisted of bug fixes, enhancements to existing features, new features, documentation related changes, test related contributions, and refactoring. Each classification is described in greater detail below.

Bug Fix: A bug fix was defined as being an issue which requested an error or unexpected behavior to be solved by a newcomer. A bug fix can be identified by seeing an issue or commit focusing on unintended behavior, such as results not matching the expected outcome or a program crashing during execution.

Enhancement: Enhancements to existing features are seen as (small) extensions to existing behavior that would allow a project to perform additional tasks within existing functionalities, such as allowing existing export functionality to support a new file type or adding additional user interface elements to an already existing user interface.

New Feature: A new feature is defined as an issue requesting a newcomer to add new functionality, such as adding an ability to perform exports or creating entirely new functionality as opposed to merely extending it. Issues requesting and commits adding a new features distinguish themselves by focusing on the new behavior added or by stating that the created functionality did not exist in any form before the contribution.

Documentation: A documentation related change requires a newcomer to make adjustments or enhancements to documentation, such as project documentation or comments within the codebase. Documentation related changes can include type-error fixes, comment changes for clarification or to reflect new behavior, or textual contributions without new content in the codebase.

Testing: A test or testing related change required new tests to be created by a developer or modifying existing tests to ensure they are testing correct behavior. Issues and commits related to testing can usually be identified by identifying which location the modifications have been made in the codebase. The majority of testing related contributions will be put within a separate *test* directory or have the word *testing* within the file name.

Refactoring: A refactoring related issue or commit requires a new developer to change an existing part of the codebase with a focus on making it simpler or easier to utilize, but without extending its existing features or changing behavior. An example of this could be a class which performs sorting in multiple methods. Such a class could be modified to have a single *sort* method which is called in each other method in the class. The behavior or features provided is not changed, but a modification to make it easier to maintain is contributed.

In addition to the taxonomy given above, a field for additional information was given per issue, allowing notes to be made that might be relevant to an issue. Some issue might not have an associated pull request due to the rejection of what is requested in an issue by project maintainers. This type of information could then be tracked within this field. Based on this template, each sampled issue as outlined in Section 3.2.1 was manually inspected and the template was filled in for each issue.

3.2.3 Analyzing First Commits

To analyze the type of tasks new contributors made within their first commits for a project, another CSV template was made which features additional relevant information related to a commit. The decision was made to focus on the pull request associated with the first commit as opposed to only the first commit such that additional context and relevant information of a first contribution could be understood. If a commit had no pull request, then the commit information was used for analysis. The template employed the taxonomy introduced in Section 3.2.2 and contains additional fields as shown in Table 3.2.

Fields for the title of the first commit pull request and *Github* hyperlink such that a human readable description and location could reveal the contribution of pull request and first commit. Additionally, an inspection was done to validate whether or not the pull request and associated first commit were related to a *good first issue*. Although *good first issue*s were tracked within the issue analysis stage, due to sampling it is possible that a *good first issue* is not analyzed but a commit associated to that issue is. Hence, this field inspected to account for possible sampling related mismatches.

The experience of the developer at the time of first contribution was also analyzed. Perhaps it could be seen that a developer's experience had an impact upon the type of first contribution they would make. To categorize the developers analyzed, three categories were created for which each developer was put into. These categories are presented in Table 3.3. First, active experience was defined as a developer having made at least 10 commits within a year. With that definitions in mind, *Novice* developers were developers who had less than one year of active experience at the time of their first contribution. Developers with between one and two years of active experience were labelled as *intermediate*[10]. Developers with more active experience were marked as *experienced* developers.

Beyond these fields, the taxonomy employed matches exactly with the taxonomy introduced in Section 3.2.2. Each commit and developer profile was manually inspected accordingly.

3.3 Questioning of Individuals

In addition to the obtaining of repository data and its analysis, and with the goal of answering research question 6 as outlined in Section 1.3, a questionnaire was created and sent out to developers that had been found to contribute with the aim of identifying non-functional

[10]During data mining and analysis, intermediate developers were classified as *medior*s. This was changed later for grammar related purposes, but the data was left unedited.

Column Name	Description
First Commit Pull Request	Name of the pull request or commit.
Related to a Good-First-Issue?	If the commit/PR is related to a good first issue, a value of 1 is placed here. Otherwise, 0.
Developer Experience	Contains the experience level of the commit's developer.
Link to PR	Contains a hyperlink to the associated PR or commit for this issue.
Bug Fix	The issue is related to a mistake or error that needs to be fixed.
Enhancing an Existing Feature	The issue wants an existing feature to be extended with new functionality.
New Feature	The issue requires an entirely new functionality to be created.
Documentation	The issue is related to a change in the documentation.
Testing	The issue is related to the creation of changing of tests.
Refactoring	This issue requires changes to be made that optimize or increase readability.
Other Information	Stores additional tags or relevant information if required.

Table 3.2: An overview of the aspects analyzed for each commit as presented in the CSV template for commit analysis.

Category	Level of Experience
Novice	$<$1 year of active experience
Intermediate	1 - 2 years of active experience
Experienced	$>$2 years of experience

Table 3.3: An overview of the labels utilized to label developer experience and the requirements for each category.

and non-numerical aspects related to *good first issue*s. The method in which the sample population was selected is outlined in Section 3.3.1, and details related to the given questionnaire is provided in Section 3.3.2. A note about the ethical validity of the questionnaire is given in Section 3.3.3.

3.3.1 Sample Selection

To be selected as a possible participant to answer the questionnaire, a developer had to be part of the sampling that was done as outlined in Section 3.2.1. If a developer was part of the 30 sampled commits (since issues themselves do not always have assigned developers), their *Github* profile would be inspected. If the profile contained a publicly accessible and presented e-mail address as to respect their privacy and demands, the developer was included in the sample set and received an invitation to participate in the survey. No further contact took place, nor was any additional information utilized to identify developers. Once a developer was selected to partake in the survey, they were categorized into positive and negative groups. Developers that had made a first commit in a repository that was related to a *good first issue* were put into the positive group, and developers that had submitted a contribution unrelated to a *good first issue* were placed within the negative group.

3.3.2 Questionnaire

The questionnaire that was sent to sampled developers aimed to identify developer thoughts regarding *good first issue*s and the newcomer developer experience. Since developers were split into two groups as outlined in Section 3.3.1, two different versions of the survey were created. The differences between the questions only existed in question 1, which had slightly different wording and an additional sub-question within the negative variant. The questionnaire was hosted upon the *SurveyMonkey* platform[11].

Question 1 (Positive Variant)
You were identified to have made at least one first contribution to an (open source) project that was related to a task/issue labelled as a *good first issue*. Did you make this contribution due to this label, or in part as a reference from this label? Can you describe how you identify your first possible contribution when joining a project?

Question 1a (Negative Variant)
Your first contribution to the indicated (open source) project was not related to a task/issue labelled as a good first issue. Why did you not employ a task/issue assigned as a *good first issue* for your first contribution? Can you describe how you identify your first possible contribution when joining a project?

Question 1b (Negative Variant)
Do you prefer or try to find issues/tasks labelled as *good first issue*s when you

[11]Additional information can be found at: `surveymonkey.com`

want to make an initial contribution to a project/repository?
Answer Options: **Yes**, **No**

The first question in both variants aims to identify whether the developer felt that the *good first issue* label assisted them in making their contribution, and whether they prefer having such indicators when contributing to a project or repository. For the negative variant, the focus was shifted to understanding whether a developer would have preferred to have completed a *good first issue* and whether they would have taken a *good first issue* if it was available.

Question 2a (Both Variants)
You were identified to have made at least one first contribution to an (open source) project that was related to a task/issue labelled as a *good first issue*. Did you make this contribution due to this label, or in part as a reference from this label? Can you describe how you identify your first possible contribution when joining a project?
Answer Options: **Not Useful**, **Somewhat Useful**, **Very Useful**

Question 2b (Both Variants)
Why do you believe that labelling tasks as suitable for newcomers is as you indicated above?

The second question aims to identify how developers feel towards the idea of labelling tasks as *good first issue*s. By obtaining this information, it can be seen whether or not investing time as a project or repository into labelling *good first issue*s is worth it from a developer standpoint, and perhaps identify weaknesses in employing a technique of directing newcomers to such tasks.

Question 3 (Both Variants)
What types of tasks do you prefer doing when making a first contribution to a project or software repository? Why do you prefer these types of tasks?

Although numerical data regarding developer practices and habits are analyzed as outlined in Section 3.2.3, it can also be considered important to investigate the developer standpoint. It may be that developers indicate that tasks that they find useful when making an initial contribution to a project do not align with the suggested tasks given by repository maintainers. This question aims to identify such possible misalignment.

Question 4 (Both Variants)
How would you inform developers about tasks/issues that are good for newcomers if you were a maintainer of a project?

Perhaps directing newcomers to issues on *Github* is not a preferred method, and perhaps this is leading to a loss in effectiveness of the labelling of *good first issue*s. To identify if this is the case, and to investigate what the preferred method of directed onboarding for newcomers would be, this question aims to allow a developer to indicate what they would do.

Question 5 (Both Variants) ______________________

If you wish to be informed of the completion of the thesis report and access it, you may leave your e-mail address here. We will send you a notification once research is completed. Please note that this is optional and you may skip this question.

To provide a developer with the opportunity to be informed of the outcome of the survey and research, a respondent may voluntarily leave their e-mail address. They will be informed after termination of the thesis work and upon publication of the thesis. All data related to this question was destroyed upon completion of the research.

3.3.3 Guaranteeing Ethical Practices

To ensure ethical procedures and standards were followed and adhered to, Delft University of Technology's human research committee[12] was informed of the intention to perform this survey and approved the questionnaire under request 1119. The approval was given on 1 May 2020. The committee aimed to ensure that participants in the survey would not violate essential rights of participants as dictated by Delft University of Technology's regulations on human trials [39].

3.4 Overview of Procedures

To provide additional clarity, a visual representation of the procedure as outlined in Sections 3.1, 3.2, and 3.3 is provided in Figure 3.4.

[12]More information can be found at: `tudelft.nl/en/about-tu-delft/strategy/integrity-policy/human-research-ethics/`

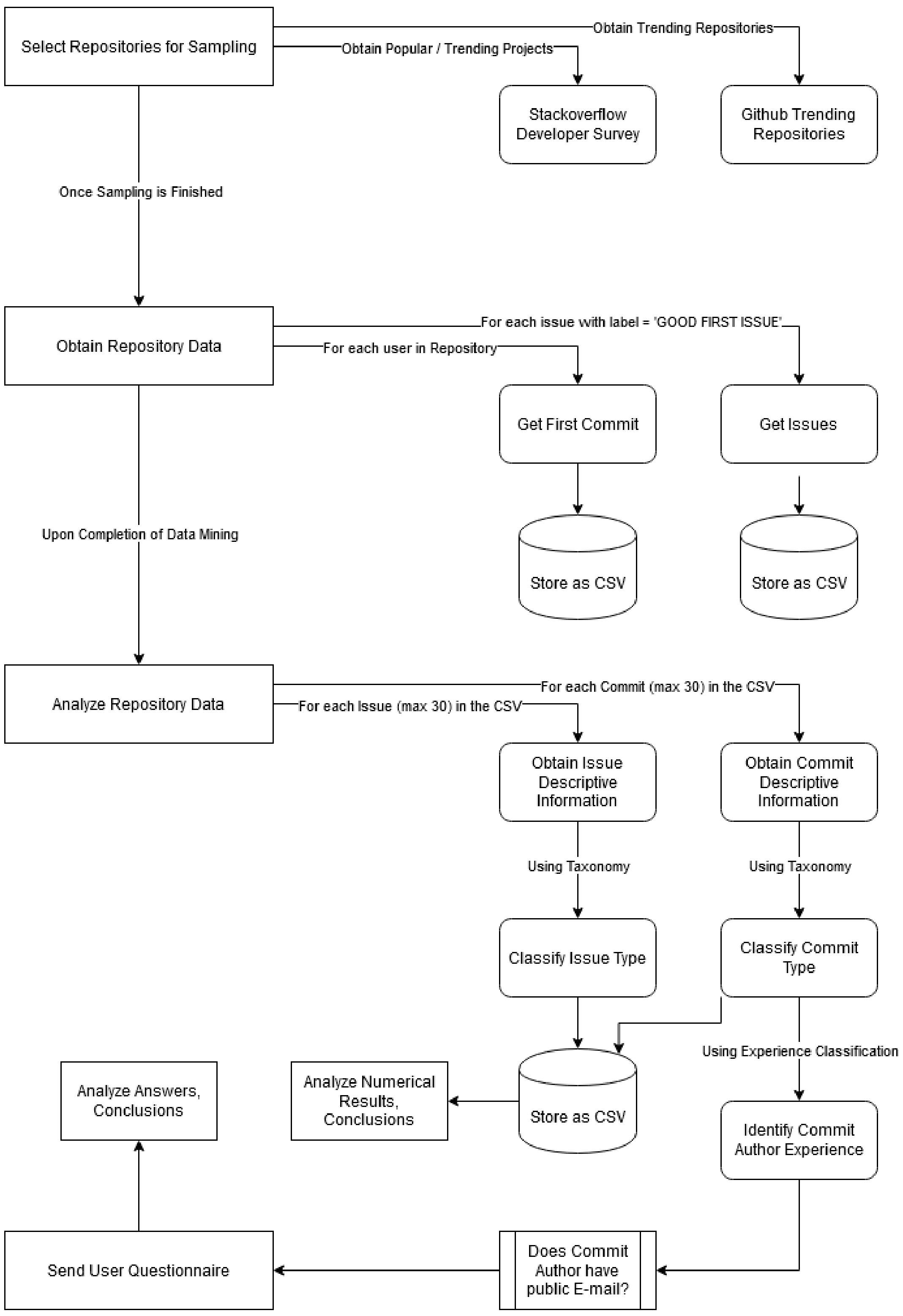

Select Repositories for Sampling
Obtain Trending Repositories
Obtain Popular / Trending Projects
Stackoverflow Developer Survey
Github Trending Repositories
Once Sampling is Finished
Obtain Repository Data
For each issue with label = 'GOOD FIRST ISSUE'
For each user in Repository
Get First Commit
Get Issues
Store as CSV
Store as CSV
Upon Completion of Data Mining
Analyze Repository Data
For each Commit (max 30) in the CSV
For each Issue (max 30) in the CSV
Obtain Issue Descriptive Information
Obtain Commit Descriptive Information
Using Taxonomy
Using Taxonomy
Classify Issue Type
Classify Commit Type
Using Experience Classification
Analyze Answers, Conclusions
Analyze Numerical Results, Conclusions
Store as CSV
Identify Commit Author Experience
Send User Questionnaire
Does Commit Author have public E-mail?

Figure 3.4: A visual flowchart overview of the procedure as outlined in the methodology..

Chapter 4

Results and Analysis

This chapter provides an overview of the results that were found after both numerical and manual analysis of the data and an analysis of these findings. The results section also aims to provide results based upon the classifications utilized and defined in Chapter 3. The numerical results of the data mining process are provided in Section 4.1, numerical aspects of the analysis are presented in Section 4.2, a comparison between the classification differences is provided in Section 4.3 and classification combinations and their associated trends are discussed in Section 4.4. Additionally, results based on the effects of developer experience are presented in Section 4.5, and the results and analysis of the survey are presented in Section 4.6. Finally, an analysis of the overall effectiveness of *good first issue* labelling is discussed in Section 4.7.

4.1 Numerical Results of Data Mining & Sampling

Based upon the sampled repositories outlined in Section 3.1, a total of 46 repositories contained issues labelled as *good first issues*[1] out of a total of 105 repositories sampled[2], therefore suggesting that 43% of repositories utilize *good first issue*s in some form. From these repositories, a total of 301.380 issues were available for sampling, of which 4.792 were found to have the label *good first issue* or an equivalent label[3], meaning the representation of *good first issue*s within the total body of issues is 1,5%. An overview of this data is provided in Table 4.1.

4.2 Numerical Results of Analysis

This section provides an overview and insights into the findings of the analysis from a numerical perspective. The findings of the sampled issues are given in Section 4.2.1, and

[1]A list of repositories with *good first issue*s can be found at: `https://github.com/dalderliesten/Good-First-Issue/blob/master/Sample%20Set/Repositories-with-Good-First-Issues.csv`

[2]A list of the sampled repositories can be found at: `https://github.com/dalderliesten/Good-First-Issue/blob/master/Sample%20Set/Sampled-Repositories.csv`.

[3]All issues and commits that were sampled are listed at: `https://github.com/dalderliesten/Good-First-Issue/tree/master/Data`

Feature	Value	Percentage
Repositories Sampled	105	100%
Repositories with *Good First Issue*s	46	43,8%
Number of Issues	301.380	100%
Number of *Good First Issue*s	4.792	1,5%

Table 4.1: Numerical results of the data mining and sampling procedure.

Feature	Value	Percentage
*Good First Issue*s taken by Newcomers	279	32,5%
*Good First Issue*s taken by Existing Developers	340	39,6%
Deprecated & Incomplete *Good First Issue*s	239	27,8%
***Good First Issue*s Selected for Analysis**	858	100% (14,3% of obtained issues)

Table 4.2: Numerical results of the sampled issues and their completion status.

the sampled commits and their relevant aspects are given in Section 4.2.2.

4.2.1 Sampled Issues

From the 4.792 *good first issue*s that were sampled as described in Section 4.1, 858 individual issues were selected for manual analysis through the methods outlined in Section 3.2.2[4]. Of these 858 issues, 279 were implemented by a new developer or contributor to the repository, whereas 340 *good first issue*s were implemented by developers that had already made a contribution to that particular project. 239 *good first issue*s were found to not have been implemented or featured ongoing development work and therefore had no associated pull request. These data are also given in Table 4.2.

In addition to the sampling of issues, the categories in which issues were placed based upon the taxonomy as outlined in Section 3.2.2 are given in Table 4.3. Of the issues sampled, 251 were related to the fixing of a bug or solving of unintended behavior. A total of 159 *good first issue*s sampled were related to comment or documentation changes. When it comes to features, 208 *good first issue*s requested enhancements to existing features, whereas 109 *good first issue*s wanted an entirely new feature to be implemented by a newcomer. 171 *good first issue*s demanded a newcomer refactor an existing component of the codebase as to make it easier to read or optimize its components, and 75 *good first issue*s were related to the creation or updating of tests within the codebase. Note that because some issues were

[4]A list of these issues can be found at: `https://github.com/dalderliesten/Good-First-Issue/blob/master/Analysis/_ALL-ISSUES.csv`

Classification	Number of Issues Marked	Percentage
Bug Fix	251	29,2%
Documentation	159	18,5%
Enhancing a Feature	208	24,2%
New Feature	109	12,7%
Refactoring	171	19,9%
Testing	75	8,7%
Total	973	113,2%

Table 4.3: Overview of the taxonomy classification for issues that were sampled.

tagged with multiple classifications of the taxonomy the total number of markers is greater than the number of issues, which is intentional.

4.2.2 Sampled Commits

A total of 1.272 first commits by a developer that was new to the project were sampled for manual analysis as outlined in Section 3.2.3[5]. A total of 49 commits were related to a good first issue, equating to a 3,8% newcomer commit to *good first issue* ratio. Of the commits sampled, 272 were found to be a bug fixing contribution, 230 commits aimed to enhance an existing feature as to provide additional functionality, and 138 developers contributed an entirely new feature within the codebase. 447 first commits were focused on enhancing or editing documentation, 68 were related to improving or fixing tests, and 310 were related to the refactoring of existing components of the codebase to increase readability or make it more efficient. Note that the total number of markers is larger than the total number of sampled commits due to certain commits having multiple markers as outlined in Section 3.2.2. This information is also show in Table 4.4.

4.3 Comparing Issue and Commit Classifications

To evaluate whether correct types of tasks are being labelled as to investigate research questions 3 and 5, a comparison can be made between the labels assigned to *good first issue*s and the labels assigned to initial contributions from developers as sampled. This allows an evaluation to be done regarding the effectiveness of how issues are currently being labelled and whether any steering can improve the labelling practice. The numerical differences are presented in Section 4.3.1, whereas an analysis of these differences and their implications is given in Section 4.3.2. During analysis, certain trends and possible additional labels were also identified, which are provided and discussed in Section 4.3.3.

[5]The list of commits can be found at: `https://github.com/dalderliesten/Good-First-Issue/blob/master/Analysis/_ALL-COMMITS.csv`

Classification	Number of Commits (Marked)	Percentage
Bug Fix	272	21,3%
Documentation	447	35,1%
Enhancing a Feature	230	17,1%
New Feature	138	10,8%
Refactoring	310	24,3%
Testing	68	5,3%
Total	1.465	113,9%

Table 4.4: Overview of the taxonomy classification for commits that were sampled.

4.3.1 Numerical Comparison of Classifications

To determine whether initial contributions align with issues containing the *good first issue* indicator, a comparison should be made between the issue types labelled by project maintainers and the first commits made by a developer within a project and analyze any discrepancies between the two. Based upon the results obtained as shown in Tables 4.3 and 4.4, a difference can be seen between all taxonomy categories. Measuring the Δ difference indicates the percentile change between recommended *good first issue*s and actual first-time commits.

These differences, as shown in Table 4.5, indicate that the most significant difference exists in the number of documentation *good first issue*s and the number of first contributions that consider documentation. This difference of 16,6% signifies a significant increase in first-time contributors performing the action of providing documentation versus the suggested issue count. Other significant differences are given by bug fixes, which have a Δ difference of -7,9% when comparing the suggested *good first issue*s with bug fixing requirements to actual first contributions, and feature enhancements, which has a Δ difference of -7,1%. Refactoring labels indicate a Δ difference of 4,4%, whereas testing has a Δ difference of -3,4%. The smallest gap between issue and first contribution labelling exists within the feature marker, which has a Δ difference of -1,9%.

A visual comparison of all the Δ differences found for all taxonomy labels between the sampled commits and sampled issues is given in Figure 4.1.

4.3.2 Analysis of Differences

To account for limitations in sampling and differences in sample size, the Δ difference was calculated for each difference found between labels. An application of the Mann-Whitney significance test was employed to identify whether the differences were statistically significant. These test results are given in Table 4.6, indicating that a significant difference exists for the *bug fix*, *documentation*, *feature enhancement*, and *testing* categories when considering their p-values. Additionally, the Δ difference and effect sizes were considered, as shown

Taxonomy Label	Issues (%)	Commits (%)	Δ Difference (%)
Bug Fix	29,2%	21,3%	-7,9%
Documentation	18,5%	35,1%	16,6%
Enhancing a Feature	24,2%	17,1%	-7,1%
New Feature	12,7%	10,8%	-1,9%
Refactoring	19,9%	24,3%	4,4%
Testing	8,7%	5,3%	-3,4%

Table 4.5: Tabular representation of the percentile difference between labelling in sampled issues and sampled commits with respect to the *good first issue* labels.

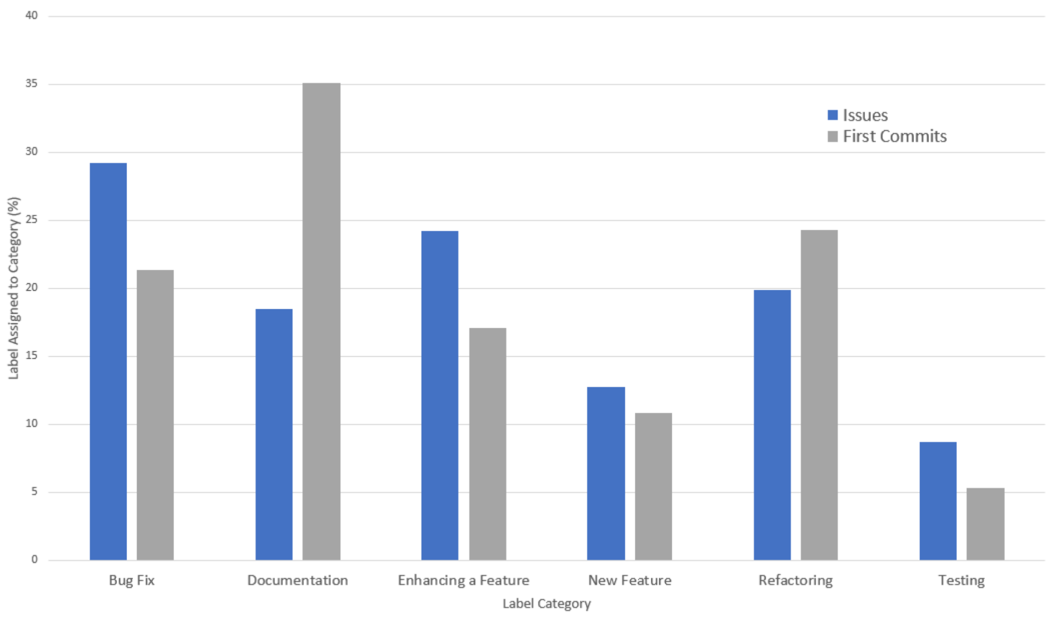


Figure 4.1: Visual overview of the percentiles assigned to each taxonomy label for both issues and commits sampled, in which the blue components represent the issue percentiles and the grey bars represent the first commit percentiles.

in Table 4.5. An effect size indicates the level of magnitude for findings, and can be utilized to identify how significant a correlation is. An effect size greater than 0,5 indicates a large effect, between 0,3 and 0,5 indicates a medium effect, and a value between 0,1 and 0,3 indicates a small effect. Note that despite multiple labels possessing a significant difference, the effect size is limited to *small* for *documentation* and *bug fix* labels, and none for all other labels. The results should, as a result, be interpreted within that context. A Δ difference of 5% was considered a significant difference, which existed for three classifications, namely the bug fixing classification with a $|\Delta|$ difference of 7,9%, the documentation classification with a $|\Delta|$ difference of 16,6%, and the feature enhancement classification with a $|\Delta|$ difference of 7,1%. The Δ differences for both bug fixing and feature enhancement categories are negative, whereas the documentation Δ difference is positive. When considering the values, the -15% offset for the aforementioned negative categories appears to be accounted for by the 16,6% positive differences for the documentation category.

Bug Fix When considering the bug fixing classification, the Δ difference is negative with a small effect size, meaning *good first issue*s recommend newcomer developers to perform bug fixes 7,9% more often than first contributors actually do. When considering the data presented, it appears that many projects and repositories suggest performing minor bug fixes for first time contributors, perhaps considering that newcomers will then be exposed to the codebase and look around some components to gain a better understanding of the project. However, developers require some understanding of the codebase before they can fix issues, and an understanding of the codebase and decisions made has been found to be attained by reading through documentation [15]. It appears that bug fix tasks are suggested at a higher rate than their likelihood of being implemented to newcomer developers and tend to be diverted by newcomer developers in favor of a documentation related contribution.

Enhancing a Feature When considering the feature enhancement classification label, the Δ difference is also negative but there is no magnitude given by effect size, for which 7,1% more *good first issue*s with the feature enhancement label are given compared to commits in which a newcomer developer enhances an existing feature. The enhancing of a feature requires that a developer either possesses knowledge of the codebase (as they must understand where components are found) and possibly even knowledge of the programming language and development tools that are utilized within the project. In order to gain this understanding, developers may have to consult documentation related to the project before they begin. When consulting this documentation, a newcomer developer may identify lacking components of the documentation or segments that can be clarified, leading to a first contribution consisting of a documentation change as opposed to an enhancement of a new feature.

Documentation The documentation label shows a positive Δ difference of 16,6% and a small effect size, indicating that less documentation-based *good first issue*s are suggested when compared to the number of first commits made in which a documentation change or addition is the contribution. The analysis results indicated in Table 4.6 also indicate that a significant z-value of -8,331334 was found, indicating that the *documentation*-related labels

Taxonomy Label	*p*-Value	*z*-Value	Effect Size	Significant Difference
Bug Fix	0,0000350067	4,138183	0,108 (Small)	Yes
Documentation	$1{,}11022 \cdot e^{-16}$	-8,331334	0,218 (Small)	Yes
Enhancing a Feature	0,000561582	3,449521	0,090 (None)	Yes
New Feature	0,186586	1,320745	0,035 (None)	No
Refactoring	0,0162341	-2,403609	0,063 (None)	No
Testing	0,00213845	3,070298	0,080 (None)	Yes

Table 4.6: Results of the two-tailed *Mann-Whitney* significance tests with a significance level $\alpha = 0,01$ between the differences in taxonomy labels, including their associated effect sizes.

feature a significant discrepancy. It is known that documentation serves as a primary source to obtain information related to development [17], and that more in-depth documentation leads to better development turnaround and fewer errors during development [42], and the results appear to indicate that newcomer developers within the sampled projects read documentation before making a contribution. When analyzing the sampled commits, many documentation labelled contributions make minor changes, such as fixing spelling errors or clarifying the phrasing of a comment. These trends seem to suggest that many newcomer developers first utilize the documentation to obtain the necessary knowledge for other contributions, but instead fix issues existing within the documentation. As a result, repositories aiming to onboard new developer should aim to provide additional documentation-related *good first issue*s to reflect this tendency.

To investigate this finding further, 30 additional second contributions were identified from developers that had provided an initial contribution to a project that was assigned a *documentation* label. From these 30 sampled first contributions by newcomer developers, 10 secondary contributions were made, indicating at 33,3% rate of documentation labels leading to a second contribution. Of these 10 secondary contributions, one secondary contribution was related to a *new feature*, one was related to the *enhancement of a feature*, one to *testing*, and one to *refactoring*. The majority of secondary contributions, however, consisted of additional *documentation* based changes, adding six. This appears to suggest that, although *documentation* related changes are a good *good first issue* for newcomers, they do not directly result in more complex contributions being made after the initial commit.

Testing Despite having a small Δ difference of -3.4%, the Mann-Whitney significance test indicated that a significant difference existed within the testing category although there was not effect size indicated. Upon analysis of the data, it was found that the difference between the number of issues suggesting a *testing* related *good first issue* and actual first contributions might be explained by a difference in approach. Projects tend to create one (*good first*) issue per test that must be created, usually related to an entity or class. However, commits tend to favor clustering many entities and classes into a single pull request. When considering (tightly) coupled entities and classes, it makes sense to developer system-wide tests and smoke tests in context of each other as opposed to developing them in isolation, as the *good first issue*s appear to suggest. This may help suggest that repositories might prefer to mark related tests that need to be developed into a single *good first issue* as opposed to multiple issues.

4.3.3 Additional Labels and Trends

During classification of both issues and commits, an additional type of contribution was identified that could warrant its own label, namely that of *organizational* tasks. Organizational tasks are tasks related to the organization of the project or repository, such as adding or removing dependencies, moving files between locations during a re-organization, or copying an existing repository over to a new git storage platform. The organizational label would have accounted for 58 contributions, or 4,5%, from commits within the sample population.

4.4 Comparing Classification Combinations

After having analyzed the classification labels and their differences in Section 4.3, and considering the nature of the taxonomy allowing for multiple labels to be applied to a single commit or issue, an analysis was made of the existing combination of labels to identify possible trends that can help improve labelling practices. An overview of the numerical results of the classification combinations is given in Section 4.4.1, and an analysis of this data is provided in Section 4.4.2.

4.4.1 Numerical Results of Combinations

The labels that are co-associated with each other within the sample population of commits is given in Table 4.7, whereas the co-associated classifications found within the sampled issue population is given in Table 4.8. Due to the nature of the data and the large number of possible combinations, not every individual category combination is listed here. Additionally, some association data mining was done to identify whether any inferences existed between label combinations, the results of which are presented in Table 4.9 for the commits and Table 4.10 for the issues sampled.

4.4.2 Analysis of Combinations

Based upon the findings of the association mining and statistical analysis as shown in Table 4.9 and Table 4.10, the combinations identified are not statistically significant. To be statistically significant, one of the combinations would need to have a lift value of ≥ 1 to determine some form of association, which none of the combinations are close to having. To further analyze the combination results in context of the significance, a decision was made to utilize the raw comparison numbers to analyze the data.

For both issues and commits, the same label combinations were found to be significant and a common occurrence when utilizing raw values for analysis. For issues, the significant combinations consist of the *Refactoring / Enhancing a Feature* label combination with 27 instances and the *Refactoring / Bug Fixing* label combination with 23 occurrences. When considering commits, the significant combinations are identical, with *Refactoring / Enhancing a Feature* yielding 45 instances of the combination and *Refactoring / Bug Fixing* yielding 37 occurrences. As a result, it is interesting to consider why the refactoring and bug fixing or feature enhancement combinations seem to occur more often than their counterparts.

From the perspective of programming, refactoring and bug fixing form a natural association. Previous work has shown that refactoring certain types of code smells within a codebase has a tendency to fix bugs within the system at the same time [36][40], thereby helping to explain the combination's degree of coupling. Identifying the link between refactoring and the enhancement of features is a bit more difficult, as no previous research provides a clear indication of what causes this. Based upon observations within the scope of the analysis, many enhancements contributed by newcomers focused on extending file format capability or featured re-writes to make existing features more extendable, thereby usually

Category A / Category B	Bug Fix	Enhancing a Feature	New Feature	Documentation	Testing	Refactoring
Bug Fix	▨	14	1	24	7	37
Enhancing a Feature	14	▨	9	6	7	45
New Feature	1	9	▨	10	13	13
Documentation	24	6	10	▨	7	12
Testing	7	7	13	7	▨	13
Refactoring	37	45	13	12	13	▨

Table 4.7: Tabular representation of the number of commits per label in the row that have a combination with the label in the column.

Category A / Category B	Bug Fix	Enhancing a Feature	New Feature	Documentation	Testing	Refactoring
Bug Fix	▨	10	2	4	17	23
Enhancing a Feature	10	▨	4	1	7	27
New Feature	2	4	▨	2	8	6
Documentation	4	1	2	▨	3	3
Testing	17	7	8	3	▨	8
Refactoring	23	27	6	3	8	▨

Table 4.8: Tabular representation of the number of issues per label in the row that have a combination with another issues labelled in the column.

Category A / Category B	Bug Fix	Enhancing a Feature	New Feature	Documentation	Testing	Refactoring
Bug Fix		(0,011, 0,0610, 0,0002)	(0,0007, 0,0072, 0,00002)	(0,0188, 0,0537, 0,0002)	(0,0055, 0,1029, 0,0004)	(0,0291, 0,1193, 0,0004)
Enhancing a Feature	(0,0110, 0,0511, 0,0002)		(0,007, 0,0652, 0,0003)	(0,0047, 0,0134, 0,00005)	(0,0055, 0,1029, 0,00044)	(0,0353, 0,1451, 0,0006)
New Feature	(0,0008, 0,0036, 0,00003)	(0,007, 0,0391, 0,0003)		(0,0078, 0,0224, 0,00016)	(0,0102, 0,1911, 0,0013)	(0,0102, 0,0419, 0,0003)
Documentation	(0,0188, 0,0882, 0,0002)	(0,0047, 0,026, 0,00006)	(0,0078, 0,0724, 0,00016)		(0,0055, 0,1029, 0,0002)	(0,0094, 0,0387, 0,00008)
Testing	(0,0055, 0,0257, 0,0003)	(0,0055, 0,0304, 0,00044)	(0,0102, 0,0942, 0,0014)	(0,0055, 0,0156, 0,00023)		(0,0102, 0,0419, 0,0006)
Refactoring	(0,0290, 0,1360, 0,0004)	(0,0354, 0,1956, 0,00063)	(0,0102, 0,0942, 0,0003)	(0,0094, 0,0268, 0,00008)	(0,0102, 0,1911, 0,0006)	

Table 4.9: Tabular representation of the results of the association rule mining, displayed as tuples consisting of *(support, confidence, lift)*, for the combinations found in the sampled commit population.

Category A / Category B	Bug Fix	Enhancing a Feature	New Feature	Documentation	Testing	Refactoring
Bug Fix	▨	(0,0116, 0,04807, 0,00019)	(0,0023, 0,01834, 0,00007)	(0,0046, 0,02515, 0,00010)	(0,0198, 0,22666, 0,00090)	(0,0268, 0,13450, 0,00053)
Enhancing a Feature	(0,0116, 0,0398, 0,00019)	▨	(0,0047, 0,03669, 0,00017)	(0,0012, 0,00628, 0,00003)	(0,0082, 0,09333, 0,00044)	(0,0315, 0,15789, 0,00075)
New Feature	(0,0023, 0,0079, 0,00007)	(0,0046, 0,01923, 0,00017)	▨	(0,0023, 0,01257, 0,00011)	(0,0093, 0,10666, 0,00097)	(0,0069, 0,03508, 0,00032)
Documentation	(0,0046, 0,0159, 0,00010)	(0,0012, 0,0048, 0,00003)	(0,0023, 0,01834, 0,00011)	▨	(0,0035, 0,04000, 0,00025)	(0,0035, 0,01754, 0,00011)
Testing	(0,0198, 0,0677, 0,00090)	(0,0082, 0,0337, 0,00021)	(0,0093, 0,07339, 0,00097)	(0,0035, 0,01886, 0,00025)	▨	(0,0093, 0,04678, 0,00062)
Refactoring	(0,0268, 0,0916, 0,00053)	(0,0315, 0,1298, 0,00075)	(0,0070, 0,05504, 0,00032)	(0,0035, 0,01886, 0,00011)	(0,0093, 0,10666, 0,00062)	▨

Table 4.10: Tabular representation of the results of the association rule mining, displayed as tuples consisting of *(support, confidence, lift)*, for the combinations found in the sampled issue population.

retouching existing components to make them easier to extend or read. This follows a textbook definition of refactoring, and enhancing features has been identified as a good moment to consider changing parts of the codebase to meet the extension's needs.

When considering *good first issue* labels, and keeping in mind the rate of significance of the data, the combinations themselves do not appear to yield a clear indication that certain combinations are preferred by new developers or that a certain label tends to promote another label taking place. There is also no clear indication that developers exhibit a preference to certain labels being combined under both raw and statistical considerations.

When considering one-way associations as represented in the *confidence* value of the tuples shown in Tables 4.9 and 4.10, a number of one-way associations can be found related to the *testing* classification. Within the commit associations, when considering the association *bug fixing* to *testing* a confidence value of 10,29% whereas the association *testing* to *bug fixing* is merely 2,57% is found, implying that first contributions that fix bugs are likely to involve testing, but not the other way around. When attempting to fix bugs in a codebase, it is likely seen as a good practice to contribute tests that help prevent those bugs from arising again, possibly explaining the combination. It has also been found that tests are commonly written to help replicate bugs and prevent them from occurring again [56]. It seems more probable to write tests after fixing bugs, than fixing bugs after having made tests, as those tests would have likely indicated bugs that could have been contributed before the bugged code enter the codebase. An identical link was found when considering *feature enhancement* and *testing*, which has a confidence value of 10,29%, whereas the *testing* to *feature enhancement* association has a confidence value of 3,04%. This can also be explained by enhanced features requiring testing, and testing itself likely not leading to the enhancement of features. Another related association is the *new feature* to *testing* association with a confidence of 19,11%, whereas the counterpart *testing* to *new feature* association is merely 9,42%. This suggests that new features tend to come bundled with tests to verify this new functionality, whereas tests themselves are not a stable indicator of a new feature being added. Across all facets, labels tend to invoke new testing behavior, even when considering *documentation* changes or the *refactoring* to *testing* confidence of 19,11%, whereas tests themselves do not guarantee another contribution being made directly.

One-way associations for issues reveal similar trends as the commit counterparts outlined above. The *bug fixing* to *testing* confidence tends to 22,66%, whereas the reverse confidence tends to a mere 6,77%. This trend continues for all label combinations with *testing*, in which the label to *testing* association reveals a high confidence, whereas the reverse is not the case.

4.5 Developer Experience

Developer experience was tracked for commits that were implemented as outlined in Section 3.2.3 and employing the three-step categorization given in Table 3.3. To determine whether developer experience caused a shift in the type of first contributions that were made, these data ware tracked and analyzed to investigate whether a correlation between experience and types of tasks existed. The numerical data related to developer experience are presented in

Experience Classification	Number of Commits	Total %
Novice	154	12,1%
Intermediate	251	19,7%
Experienced	849	66,7%
Bot	18	1,4%
Total	1.272	100%

Table 4.11: Tabular overview of the developer experience distribution of sampled commits.

Classification	Value	Percentage
Bug Fix	35	22,7%
Documentation	77	50,0%
Enhancing a Feature	27	17,5%
New Feature	9	5,8%
Refactoring	4	2,5%
Testing	25	16,2%
Total	177	114,7%

Table 4.12: Tabular overview of the sampled commits distribution for the *novice* category of developer experience.

Section 4.5.1, and an analysis of the data is performed in Section 4.5.2.

4.5.1 Numerical Developer Experience Data

The developer experience categorization was performed upon all sampled commits as defined in Section 4.2.2. As shown in Table 4.11, a total of 154 first commits were found to be contributed by a *novice* developer with less than a year of active experience, 251 contributions were made by a *intermediate* developer with between one and two years of experience, and 846 commits were contributed by an *experienced* developer with more than two years of active experience. Additionally, 18 contributions sampled were found to come from automated processes (such as updating a dependency with a security vulnerability or removing a deprecated file), which were labelled as *bot* contributions.

Additionally, the taxonomy for labels was also investigated based upon the developer experience level they were associated with. A visual overview of the distributions per developer experience category is given in Figure 4.2.

For the *novice* category of developers, it was found that 35 of the commits were bug fixes, 27 commits enhanced a new feature, 9 commits added new features to the codebase, and 77 commits focused on enhancing or editing documentation. Additionally, 4 commits added tests, and 25 commits were related to refactoring components within the codebase. These data are provided in tabular form in Table 4.12.

Classification	Value	Percentage
Bug Fix	53	21,1%
Documentation	88	35,0%
Enhancing a Feature	44	17,5%
New Feature	24	9,5%
Refactoring	62	24,7%
Testing	15	5,9%
Total	286	113,7%

Table 4.13: Tabular overview of the sampled commits distribution for the *intermediate* category of developer experience.

Classification	Value	Percentage
Bug Fix	183	21,6%
Documentation	272	32,1%
Enhancing a Feature	159	18,7%
New Feature	105	12,4%
Refactoring	214	25,2%
Testing	48	5,6%
Total	981	115,6%

Table 4.14: Tabular overview of the sampled commits distribution for the *experienced* category of developer experience.

When considering the *intermediate* category of developer experience as shown in Table 4.13, a total of 251 commits were identified of which 53 were bug fixing contributions, 88 were related to documentation, 62 were related to refactoring, and 15 were related to the creation of tests. Regarding features, 44 commits enhanced existing features and functionality whereas 24 contributions added a new feature to the repository.

Finally, for the *experienced* developer category it was found that of the total 846 commits, 183 were related to bug fixes and 214 were related to testing. Documentation related changes consisted of 272 commits. For features, 159 commits enhanced an existing feature in the codebase, whereas 105 commits provided a new feature to the codebase. These data are provided in tabular form in Table 4.14.

4.5.2 Analysis of Developer Experience Differences

When considering the percentage of given labels across developer experience levels, it can be stated that the *bug fix* and *feature enhancement* labels appear to be relatively leveled regardless of developer experience level. This indicates that the fixing of bugs and the enhancement of existing features is not influenced by developer experience level, which is a

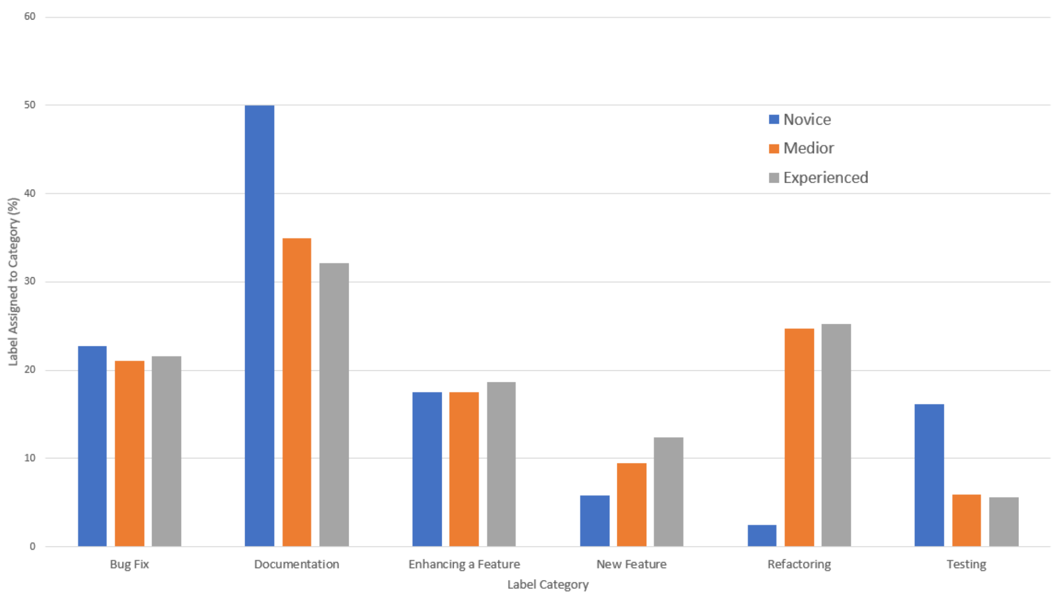


Figure 4.2: Visual overview of the percentiles assigned to each taxonomy label for all developer experience categories, in which the blue components represent the *novice* developer experience category, orange represents the *intermediate* category, and the *experienced* category is represented by the grey color.

sensible conclusion, as the resolving of bugs needs to be done regardless of developer experience. The same can be said for feature enhancement, in which features can be extended in a wide range of manners. A *novice* might add the ability to add a command line option to a feature, whereas an *experienced* developer may extend the feature to perform entirely new and complex functions.

To analyze the differences between developer experience levels and the category labels, a Mann-Whitney significance analysis was performed over the data as shown in Table 4.15 and the associated effect size values were calculated. Based on the given results, it can be stated that almost all pairs did not have a sufficient effect size to deduce any form of strong correlation. This analysis shows that only the *documentation* label has a statistically significant difference, and this should be considered when attempting to derive conclusions from the data. However, in an attempt to account for the difference in *documentation*-related contributions between the *novice* category and *intermediate* and *experienced* categories, the *testing* and *refactoring* labels are also considered as they are the most significant differences after *documentation*. These labels show significant differences between developer experience levels, and each label deserves its own explanation due to the range of tasks associated with each label.

Documentation The documentation label indicates that *novice* developers are more likely to make a first contribution consisting of a documentation change than *intermediate* or *expe-*

Classification / Experience	Novice-Intermediate	Novice-Experienced	Intermediate-Experienced
Bug Fix	(0,703; 0,381; 0,017; No)	(0,762; 0,303; 0,009; No)	(0,861; -0,174; 0,005; No)
Documentation	**(0,003; 2,966; 0,138; Yes)**	**(0,0002; 4,271; 0,126; Yes)**	(0,389; 0,861; 0,024; No)
Enhancing a Feature	(1,000; 0,000; 0; No)	(0,712; -0,369; 0,011; No)	(0,651; -0,453; 0,013; No)
New Feature	(0,185; -1,325; 0,062; No)	(0,018; -2,357; 0,069; No)	(0,219; -1,230; 0,035; No)
Refactoring	(0,044; -2,011; 0,093; No)	(0,0153; -2,424; 0,071; No)	(0,849; -0,190; 0,005; No)
Testing	(0,119; -1,557; 0,072; No)	(0,114; -1,580; 0,046; No)	(0,857; 0,180; 0,005; No)

Table 4.15: Tabular overview of the Mann-Whitney significance results and effect size for each category label and associated pair of developer experience levels, presented as a tuple consisting of (p-value, z-value, effect size, significance indicator) for each pair with a significance level $\alpha = 0,01$.

rienced developers, with an effect size of 0,138 between notice and intermediate developers and 0,126 between novice and experienced developers, indicating that the *novice* category influences the likelihood to contribute a *documentation*-related change to a small degree. This difference can be explained by the level of confidence developers may have obtained over time. Since a newcomer developer has little to no experience with software development or perhaps with the programming language employed within a repository, they are more likely to consult the documentation as to understand intricacies of the project and language that it employs, or perhaps to identify domain related knowledge related to the project. It has been found that developers that are confident and experienced do not exhibit such tendencies, where they are more likely to be certain in their programming abilities [45] and, as a result, not consult the documentation to the same extent as newcomers. As these newcomer developers consult the documentation more often, they are more likely to uncover undocumented or unclear regions within the documentation and change them. This results in a type of opportunity-benefit relationship causing additional contributions by *novice* over *intermediate* and *experienced* developers. Many smaller documentation changes were also exhibited by newcomer developers, such as a sentence re-ordering or a fix of a typographical error, which supports the aforementioned findings. It should be noted that the effect size factor for *documentation* contributions between novice-intermediate and novice-experienced indicates that this association is weak, despite it being the strongest of all experience-label pairs.

New Feature Creating a new feature appears to be a task performed more by *experienced* and *intermediate* developers as opposed to *novice* developers, with a difference of 6,6% between extrema, but under the constraint that the effect size indicates no effect with values of 0,062 to 0,035. The creation of a new feature requires, among others, the ability to plan, consider architectural requirements, and have a good knowledge of the existing codebase and its possibilities. It can be assumed that these traits are more likely to be found with experienced developers, who have already been involved in other projects or have a greater degree of mastery of a programming language. As a result, this knowledge combined with their tendency to be more certain of their programming abilities [45] results in *experienced* developer being more likely to contribute new features and *novice* developer to less likely make a first contribution containing a new feature.

Refactoring The most significant value difference between developer experience categories per taxonomy label is found for the refactoring category, in which *novice* developers made almost no first contributions whereas *intermediate* and *experienced* developers made many *refactoring* contributions, despite the effect size indicating no effect with a mere 0,093 to 0,071 for *novice* developers between the remaining categories. This is likely explained by refactoring complexity when considering what is required to perform a good re-write of an existing component of the codebase. A developer must understand the component of the codebase they wish to refactor to a great degree, as to not break any existing functionality within the codebase. Additionally, developer perception of code re-writing and refactoring is one of high complexity [48], meaning the barrier to performing perceived refactoring for newcomer *novice* developers may be higher. If the overall attitude to a task is one of complexity, it is unlikely a newcomer will be willing to perform it.

Testing *Novice* developers tend to make first-time contributions related to the testing label significantly more often than *intermediate* or *experienced* developers, with a 10,6% difference existing between the extrema. Testing provides the ability to contribute meaningfully to a codebase without necessarily needing to understand the codebase fully, as practice states that tests should allow for a black-box evaluation of the codebase. A newcomer may notice that certain features or components are not tested to a sufficient level, and may contribute using existing tests or by creating a few tests of their own. Analysis indicated that numerous *novice* test-related contributions consisted of a test aiming to check a misunderstanding they had, such as whether parsing a certain type of file format was allowed. Additionally, prior research indicates that although *experienced* developers are more effective at creating and contributing unit-based testing solutions, *novice* developers only struggle to identify best practices for test implementing, yielding a small performance penalty, but do not appear to suffer from an inability or less-than-optimal rate of creating tests. These combined forces help a *novice* developer learn more about the codebase while still contributing useful content to it.

4.6 Surveying Developer Perception of *Good First Issue*s

The results of the survey are presented in this section with the goal of identifying qualitative outcomes and considerations regarding *good first issue*s and the labelling of tasks for newcomers. A numerical overview with the results of the survey is provided in Section 4.6.1, whereas an analysis of the data of the survey is provided in Section 4.6.2.

4.6.1 Numerical Results of the Survey

A total of 537 developers were invited to participate in the survey, of which a total of 23 responses were given, garnering a response rate of 4,28%. Of the 23 responses, a single response was found to be committed in malicious intent and containing obscenities, and as a result of this it was filtered out of the response pool. As a result, the final response population consists of 22 responses, or a response rate of 4,09%. The results of the survey will be presented on a question-by-question basis, with the first question being split to account for the difference in positive and negative surveys having been sent to developers. The question given to the developers will also be provided alongside each question for ease of reading.

Question 1 (Positive Variant)

You were identified to have made at least one first contribution to an (open source) project that was related to a task/issue labelled as a *good first issue*. Did you make this contribution due to this label, or in part as a reference from this label? Can you describe how you identify your first possible contribution when joining a project?

First Contribution Influence	Responses	Percentage
Good First Issue Label	3	23,08%
Good First Issue Label with Additional Factor	3	23,08%
External Location of Suggested Tasks	1	7,69%
Work or External Requirement	2	15,38%
Random Selection or Other	4	30,77%
Total	13	100%

Table 4.16: Numerical results for the first question of the positive survey variant filtered by response type and keywords in the responses.

The first question within the positive variant of the survey aimed to identify the relevancy of the *good first issue* label to the developer's first contribution and to identify and possible alternative motivations for contributing to a repository. As indicated in Table 4.16, a total of six responses indicated that they rely on *good first issue*s to obtain a task when they first join a project or repository. Three developers state they rely exclusively on the *good first issue* label, whereas the remaining three developers state that they utilize *good first issue*s but employ an additional factor to determine their first contributions. Of these developers, one states they also wish to have an affinity with the problem at hand (preferring *good first issue*s related to their domain of choice), whereas two state they look for a combination of a *good first issue* and have some form of unfamiliarity with the project. Additionally, one developer indicated they based their first contribution on an external location of suggested tasks, in which *good first issue*s existed.

Of the developers that indicated that they did not make their contributions based on a *good first issue*, four indicated that this was due to them using random selection of a task or ignoring labels altogether, and two indicated they chose their first contributions based upon work or external requirements. External requirements consisted of relevancy to personal projects. One developer left this answer blank.

Question 1a (Negative Variant)
Your first contribution to the indicated (open source) project was not related to a task/issue labelled as a *good first issue*. Why did you not employ a task/issue assigned as a good first issue for your first contribution? Can you describe how you identify your first possible contribution when joining a project?

The first question of the negative variant of the survey aimed to identify why developers did not utilize a *good first issue* for their first contribution and what factors drive them to contribute instead of a label. Based on the results displayed in Table 4.17, four developers indicated that their first contributions tend to be guided by their personal needs. This could consist of making changes that suited their personal project needs by enhancing components they require. Three developers indicated that their first contributions were based on a

First Contribution Influence	Responses	Percentage
Interest in Project (Component)	3	33,33%
Personal Needs	4	44,44%
Fixing Issues Affecting Developer	2	22,22%
Total	9	100%

Table 4.17: Numerical results for the first question of the negative survey variant filtered by response type and keywords in the responses.

personal interest in a certain project component, such as by preference or educational background. Finally, two developers mentioned their first contributions are often based around fixes required to make a project or dependency work within their personal environments, such as bug fixes.

Question 1b (Negative Variant)
Do you prefer or try to find issues/tasks labelled as *good first issue*s when you want to make an initial contribution to a project/repository?
Answer Options: **Yes**, **No**

It is worth noting that two developers indicated that they contributed to many projects that did not have *good first issue* labels at the time of their first contribution. When considering whether developers would personally consider *good first issue*s to be useful, 7 respondents representing 77,78% stated they do find them useful, whereas 2 developers representing 22,22% indicated they do not find them useful.

Question 2
You were identified to have made at least one first contribution to an (open source) project that was related to a task/issue labelled as a *good first issue*. Did you make this contribution due to this label, or in part as a reference from this label? Can you describe how you identify your first possible contribution when joining a project? Why do you believe that labelling tasks as suitable for newcomers is as you indicated above?
Answer Options: **Not Useful**, **Somewhat Useful**, **Very Useful**

The second question aimed to identify a numerical value that developer place upon *good first issue*s. Each developer was able to indicate a weight value between zero (indicating that they believed *good first issue*s were useless) and 100 (indicating that the developer found *good first issue*s to be very useful) on a scale with increments of 25, and the amalgamation of that data was used to obtain an average score of 70,45. The build-up of this score and all associated values are given in Table 4.18.

Question 3
What types of tasks do you prefer doing when making a first contribution to a project or software repository? Why do you prefer these types of tasks?

Weight	Number of Developers Indicating Weight	Percentage
0	2	9,10%
25	1	4,55%
50	5	22,73%
75	5	22,73%
100	9	40,91%
Total	22	100%

Table 4.18: Numerical weighting results for developer indicated usefulness of *good first issue*s in the survey.

Question three aimed to identify the reasoning behind the scores given in question two. The identified trends per weight are presented in this section.

When considering responses from developer having assigned a usefulness weight of 0 or 25, one survey respondent states that the identification of *good first issue*s takes place from a point of view of intrinsic ability and motivation. It is difficult to consider multiple types of motivation, as these can be dependent on work required by each developer or their personal desires. They also state that a mismatch likely exists between the intention of *good first issue* labels and newcomer developers, with one respondent summarizing it as "a developer doesn't randomly join a project to contribute whatever." Developers are claimed to contribute due to requirements or desires they have, and steering through issues will not change that. The respondents also mention that they believe more experienced developer are likely to find an issue during development, fix it with a pull request, and ignore labelling altogether, negating the effects of newcomer task recommendation.

For the 50 weight category, the respondents indicated that they believed *good first issue*s and labelling are only partially effective due to the difference between novice developers and more experienced developers. They also mention that existing contributors are unlikely to find any usefulness of tagging tasks for newcomers, but could possibly cause them to avoid such tasks, perhaps causing indefinite postponement of the task's completion. The respondents also state that, like the 0 to 25 weighting respondents, they believe many developers will join projects with their own intrinsic goals, not necessarily caring about tasks that need to be done from the project's perspective. One respondent within this category stated that they believe that although labelling tasks as *good first issue*s is a good thing, the effort required to organize such labelling and to perform evaluation of which tasks would be suitable outweigh the benefits.

The most positive categories are the 75-100 weight categories. Respondents highlight the increased navigability in a repository with many issues, as newcomers can be directed to the *good first issue* labels. They also believe that, since maintainers that create the labels know the codebase the best, they are best suited to suggest newcomer tasks. In addition to their expertise, allowing newcomers to get to know maintainers and make a contribution without having to worry of additional barriers is seen by multiple respondents as a positive aspect of these labels, as newcomers know exactly which tasks they can pick-up that are

Category	Responses	Percentage
Bug Fix	13	37,14%
Documentation	7	20%
Enhancing a Feature	3	8,57%
New Feature	4	11,43%
Refactoring	1	2,85%
Testing	2	5,71%
Package Management	1	2,85%
Tasks Developer Needs	2	5,71%
No Preferences	2	5,71%
Total	35	100%

Table 4.19: Numerical preference results for developer preferred tasks for *good first issue*s or first contributions, with one respondent being able to give multiple responses.

desired. The majority of the justifications revolve around increased accessibility for newcomers as aforementioned. Another respondent mentions how *good first issue*s and task labelling can prevent newcomers from doing a task that is too difficult to them, preventing them from joining a project due to the perceived difficulty.

Question 4 (Both Variants)
How would you inform developers about tasks/issues that are good for newcomers if you were a maintainer of a project?

To assess the types of tasks developers prefer when making an initial contribution, and with the goal of identifying possible *good first issue*s, the fourth question investigated what types of tasks developers prefer. Just as with the taxonomy from Section 4.3, each developer was able to give multiple category preferences for this question. The results, as shown in Table 4.19, indicate that developers prefer first contributions to consist of *bug fixes* with 13 preference indications, followed by *documentation* contributions with seven preferences. First contributions consisting of *new features* are preferred with four indications, whereas *feature enhancements* are preferred with three indicators. All other categories in the taxonomy have either one or two indicators.

In addition to the taxonomy labels, additional responses were identified outlining *package management* as a good first contribution with one indication, and any task related to *developer needs* and having *no preference for tasks* both coming in at two indications each.

Question 5
If you wish to be informed of the completion of the thesis report and access it, you may leave your e-mail address here. We will send you a notification once research is completed. Please note that this is optional and you may skip this question.

Method	Responses	Percentage
Issue Labelling	15	53,57%
Effort & Impact Labelling	1	3,57%
README or Documents	7	25%
Refer to Existing Developers	1	3,57%
Social Media	2	7,14%
Do not Recommend Tasks	2	7,14%
Total	28	100%

Table 4.20: Results of the newcomer task direction preferences for developers as indicated by the survey results.

To identify alternative methods to *good first issue* labelling or *Github* issues, developers were asked how they would provide an overview of *good first issue*s to newcomer developers. Just as with question four, a single developer could indicate multiple methods for newcomer task direction. Over half of the respondents indicated they would perform issues labelling akin to *good first issue*s. One respondent mentioned that within their project they employ so-called *effort & impact* labelling to not only direct newcomers to possible tasks, but to also indicate how much effort is required for the task and what the effect of their contribution would be upon the codebase. Seven other respondents indicated they would direct good newcomer tasks within a *README* file or another identical document within a project repository.

Beyond labelling and in-repository documentation, one developer stated they preferred newcomers being referred to an existing developer within the project to help them with a smaller component of their task. This would provide a form of mentoring relationship between newcomer and existing developer. Two other indications consisted of utilizing social media (such as *Facebook*[6] or *Twitter*[7]) to direct newcomers to pools of tasks that would be suitable for them.

Two respondents indicated they would not do any form of newcomer task recommendation, stating they would prefer to have newcomer identify these tasks themselves as to account for differences in knowledge and developer experience among newcomers.

4.6.2 Analysis of Survey Results

To analyze the outcome of the survey, the questions must be grouped based on what can be extracted from their responses. Questions one through three focus on identifying developer usefulness perception of *good first issue*s, whereas question four aims to identify the type of tasks that developers prefer to do for initial contributions. Question five assesses alternatives to *good first issue*s. These categories are analyzed separately within this section.

[6]Located at: `https://www.facebook.com`

[7]Located at: `https://twitter.com`

Perceived *Good First Issue* Usefulness Developers appear to perceive *good first issue*s as useful, with the majority of developers indicating they find them to be useful with an average of 70,45 out of 100. Based on the responses given in question one and three, it can be seen that over half of developers employ *good first issue*s to identify any form of task that is suitable for them, indicating effectiveness. If developers perceive the *good first issue*s as being effective, they are more likely to utilize them and refer others to them. The most significant complaint regarding issues relegation for newcomers appears to come from the idea that it may not be directly connected to the reality of open source development. Developers indicate they do not join a project to contribute, but rather contribute because they already identified issues within the codebase. As a result, developers indicate additional consideration should be made for issues that are within a broader range of domains and regions within the codebase. Overall, the perception of *good first issue*s is positive.

Preferred Initial Contribution Types The indicated developer preferences in Table 4.19 indicate that *bug fix*es and *documentation*-related initial contributions are preferred with 37,14% and 20% respectively, which stands in partial contrast to the mined results given in Section 4.3.1 in which the results indicate 21,3% and 35,1%, respectively. The ordering between the preferences for *bug fixing* and *documentation* are reversed between the two, but the most significant difference is the *refactoring* label. Developers in the survey indicate that *refactoring* is one of their least preferred tasks with a 2,85% response rate, whereas initial contributions consist of *refactoring* tasks for 24,3% of the total sample population. Analysis of this difference related to *refactoring* is identified when comparing many *refactoring* first commits and the answers within the survey. Respondents indicate they believe that *refactoring* is a complex task which requires knowledge of the codebase and is not a task usually done by newcomers that have not been exposed to the codebase. This is in contrast with first commits related to *refactoring*, of which many perform small optimizations that appear to have a local effect, such as refactoring a for-loop into a lambda expression or changing a component to employ feature introduced in a new release of a programming environment, such as converting output to a *stream* in *Java*. Developers appear to overestimate the difficulty of *refactoring* to newcomers and do not appear to understand that *refactoring* can consist of smaller tasks.

Alternatives to *Good First Issue*s Despite aiming to identify alternatives to *good first issue* labelling, the majority of the respondents indicate that they would utilize issue labelling to some degree, with 53,57% stating outright they would keep the recommendation system as a label. One project maintainer mentioned that they would include effort and impact labelling instead of only *good first issue* labelling, raising the concept of effort and impact. If newcomers can be informed not only of a task's difficulty but also the desirability of that task's completion within a repository and the expected effort, that would increase the likelihood that newcomers are willing to perform certain tasks, according to developers. Another 25% of the respondents indicate they would prefer labelling, but within a document. This may provide the ability to allow for more nuance than within an issue, such as adding additional labels or components that *Github* does not support.

Feature	Value	Percentage
*Good First Issue*s taken by Newcomers	279	45%
*Good First Issue*s taken by Existing Developers	340	54,9%
Total	619	100%

Table 4.21: Numerical results of the sampled issues and their completion status weighted to remove deprecated or incomplete *good first issue*s.

Beyond these alternatives, developers indicate they would appreciate more social methods of task recommendation, such as by being referred to an existing developer in the project. This would allow the newcomers to gain insights and a contact within the codebase, increasing the likelihood that they will be able to get help when they need it. Additionally, 7,14% of respondents indicate they want social media of a project to be utilized. Novice developers might be more likely to join and contribute to a project if their social, non-developer channels also help newcomer developers to join the project. Only 7,14% of respondents indicate they would not do any labelling.

4.7 Effectiveness Assessment of Good First Issues

To identify the overall effectiveness of *good first issue*s, an assessment must be made of the data and analysis results obtained in previous sections of this chapter. To assess the effectiveness, data related to newcomer adoption of *good first issue*s is assessed in Section 4.7.1, a numerical assessment of *good first issue* effectiveness is given in Section 4.7.2, developer experience assessment is taken into account in Section 4.7.3, and the survey's findings are assessed in Section 4.7.4.

4.7.1 Issues taken by Newcomers

The number of issues that have been indicated as *good first issue*s and sampled that were taken by newcomer developers was 32,5%, but this includes deprecated and closed issues that were either not completed or implemented. When considering only implemented commits as shown in Table 4.21, 45% of *good first issue*s are implemented by newcomer developers. This means close to half of the implemented *good first issue*s are taken by newcomer developers. When considering sample constraints, such as there likely being more *good first issue*s than there are developers and that certain tasks might be tied to a time constraint meaning they will be completed at some point irregardless of completion status, it helps to assess why certain tasks end up being performed by non-newcomer developers. With a fulfillment rate of 45% of the sample population, the tasks suggested for *good first issue*s appear to be suitable for newcomers.

4.7.2 Numerical Good First Issue Effectiveness

When considering the numerical data as presented in Sections 4.1 and 4.2, the matches between the distribution of first contributions and *good first issue*s can be compared. As shown in Figure 4.1, the distributions between issues and commits contain a somewhat similar structure, but there are significant label mismatches for *bug fixing*, *documentation*, and *feature enhancement* classifications. This suggests that *good first issue*s related to those three labels are not being accurately assigned, and that less *bug fix*-related and *feature enhancement*-related tasks should be assigned to new developers, and that *documentation* related tasks warrant more assignees for a *good first issue*. When it comes to the remaining labels, the rate of *good first issue* suggestion matches with the rate at which first contributions are being made by newcomers.

As a result, when it comes to the effectiveness of the suggested issues, it seems that the types of tasks being recommended to newcomers only partially match the actual trends identified. *Good first issue*s can be improved upon by re-evaluating the number of *bug* and *feature enhancement* tasks assigned to newcomers, and by increasing the number of *documentation* related tasks being suggested to compensate.

4.7.3 Good First Issues Effectiveness by Experience

A focus for *good first issue*s should be to attract developers with little to no experience. Within the context of having analyzed the *novice* developer category, it appears that the suggested *good first issue*s as discussed in Section 4.7.2 feature the same shortcomings regarding their classifications as the overall dataset. *Novice* developers have a tendency to prefer *documentation* related changes, but also appear to have a bias towards *testing* related changes. To further improve the effectiveness of *good first issue*s for newcomers, additional *testing* related issues should be created within a project, and the suggestions regarding the balance between *documentation* and *bug fixing* or *feature enhancement* categories as suggested in Section 4.7.2 need to be adopted.

4.7.4 Developer Indicated Good First Issue Effectiveness

Developers appreciate the existence of *good first issue*s and utilize them to identify *good first issue*s. Developers do believe that assigning *refactoring* related tasks to newcomers is not to be done, whereas numerical data suggests that *refactoring* work is suitable for newcomers. It can be suggested that assigning simple *refactoring* tasks to newcomers in the form of *good first issue*s would be recommended to reflect the actual contributions versus the perceived contributions. Additionally, developers suggest that *good first issue*s indicate effort and impact upon the project, and employ additional non-technical social means of onboarding developers, such as by assigning a mentor for a *good first issue* or advertising tasks through social media.

Chapter 5

Discussion

The discussion section aims to analyze and identify possible shortcomings and issues related to the research methods proposed in Chapter 3 and the results obtained in Chapter 4. Limitations exist within the boundaries of the conclusions that were made that could possibly threaten the validity of the research, and by stating these and assessing their impact the validity of the conclusions can be considered. The limitations related to the dataset are described in Section 5.1, numerical data issues are discussed in Section 5.2, and the analysis' limitations are considered in Section 5.3. The survey is assessed in Section 5.4. Finally, threats to validity are given and evaluated in Section 5.5.

5.1 Limitations Related to Dataset

When considering the dataset generated for this research, an evaluation of aspects of the dataset can help to place conclusions within a certain scope or range. Assessing its validity requires an analysis of the sample size, as given in Section 5.1.1, and an assessment of its representativeness when considering a wider array of repositories, as done in Section 5.1.2. Additionally, shortcomings and a reflection upon the sampling procedure is given in Section 5.1.3.

5.1.1 Sample Size

A limitation with the research work done is the limited sample size that was utilized for the data set. The data set that was generated and sampled consisted of 105 repositories, of which 46 were utilized for sampling. A total of 301.380 issues were sampled, of which 4.792 were good first issues. Regarding the number of commits, a total of 1.465 commits were sampled. As per the 2019 *Github* year in review, the platform contains more than 40 million repositories, had at least 87 million commits within pull requests, and contained 20 million active issues[1]. When put into perspective over at most 0,0000026% of existing repositories, the total number of issues sampled represent at most 0,000015% of the total issue base, and the commits sampled represent at most 0,000055% of the total commits on

[1]Additional data available at: `https://octoverse.github.com/#community-overview`

Github. The sample size is, as a result, relatively small compared to the total possible base of samples in existence. Due to the large diversity existing within the types of software development that are created (on *Github*) and the differences that occur between each of these domains, the generalizability of the research could be harmed by the small sample size. If a sample population was obtained which due to its small size had accidentally obtained an outlying group, that would impact and, to a degree, act as a threat to the validity of the findings of this thesis work.

To account for the small sample size of this research, an effort was made to ensure that the dataset obtained was representative as discussed in Section 5.1.2 and to ensure that the methodology employed as outlined in Chapter 3 consists of procedures guaranteeing equal opportunity for all data to be sampled. This equality was ensured by employing randomization in chosen samples, resulting in each sample obtained of having had a probability of $\frac{1}{n}$ of being selected, in which n consists of the total number of issues or repositories within a project. This guarantees that no form of bias entered the sample selection procedure, thereby ensuring greater generalization of results. This focus on generalization means the results from this thesis work can be applied with a greater generality to the larger population. By accounting for this small sample size and ensuring that equal opportunity sampling exists within all mined data, the generalizability of this work increases. Therefore, the sample size should be considered when drawing conclusions from the the findings of this work, but not void them.

5.1.2 Representativeness of Dataset

Within the field of software development there are many domains which each have their own expectations, development styles, and domain requirements. Some examples of domains consist of web development, game development, financial domains, and medical software. These domain-specific attributes can have an effect on the practices within a repository, and can thus possibly affect the way in which issues are created, the type of first contributions made, and even influence developer perception of related components within a project. Due to the limited dataset size, a possibility existed that the repositories sampled and obtained for analysis would be biased towards a certain domain. This would, in turn, reduce the generalizability of the findings of this report. As a result, this aspect of the dataset was given special attention.

To ensure that a broad spectrum of project and domain types was represented, the decision was made as shown in Section 3.1 to obtain sampling repositories from multiple sources. The *Github* trending page provided opportunity-sampled repositories, but was liable to bias, since a clear preference could be seen to projects within the web development domain. As a result, additional samples were obtained from the most popular projects given in *StackOverflow*'s 2019 developer survey, which contained a number of popular projects that were not within the web development domain. When analyzing the repositories sampled as shown in Table 5.1, it can be seen that this combination of sampling techniques yielded a diverse set of domains within the sample base, but with some clearly dominating domains. Of the 105 repositories, 19 were focused on *application development*, a category relating to repositories that assist developers in doing development-related tasks that do not

Domain Type	Instances	Percentage
Application Development	19	18,10%
Web Development	17	16,19%
Machine Learning	11	10,48%
Educational	8	7,62%
Database Technology	7	6,67%
User Interface Design	6	5,71%
Administration Software	5	4,76%
Anti-Censorship/Firewall Evasion	3	2,86%
Crawling	3	2,86%
Data Analysis	3	2,86%
Distributed Data Storage	3	2,86%
Software Testing	3	2,86%
Integrated Development Environment	2	1,90%
Mobile Development	2	1,90%
Operating System	2	1,90%
Search Engine	2	1,90%
Academic Research Repository	1	1,00%
Artificial Speech System	1	1,00%
Cloud Computing	1	1,00%
Content Management System	1	1,00%
Game Development	1	1,00%
Illegal Cracking / Jail-breaking	1	1,00%
Mathematical Software	1	1,00%
Provisioning Software	1	1,00%
Reader	1	1,00%
Total	105	100%

Table 5.1: Numerical results of the sampled issues and their completion status weighted to remove deprecated or incomplete *good first issue*s.

fit in other categories. Web development continues to dominate the repositories distribution with 17 instances, or 16,19%, with machine learning trailing it by 10,48%. Educational, database, and user-interface based repositories trail, followed by domains such as mobile development, software testing, data analysis, game development, and crawling domains. The domain pool of the sample set even includes illegal activities, such as a jailbreaking repository and anti-censorship or firewall evasion, often focusing on evading the restrictions enforced by the legislature known as the *Great Firewall of China*.

Based upon the data presented in Table 5.1, the dataset covers a large number of domains and appears to be representative for a larger software development population. Major domains such as general software development and web development are represented, as are education and machine learning. Larger and less known categories, such as operating system development, game design, and mathematical software are also represented. It should be noted, however, that there is still a dominance by the web development and machine learning domains, which should be taken into consideration when drawing conclusions from this research.

5.1.3 Sampling Procedure

Upon reflection, the sampling procedure can be improved to ensure a larger number and a more diverse subset of repositories could be obtained. Although this research purposely limited itself to *Github*, additional *Git* hosting providers should be considered in future research. These providers can include public *Gitlab*[2] repositories or public *Bitbucket*[3] repositories. There is no reason to believe that significant differences would exist between hosting providers, and this would provide a larger number of data sets for analysis and sampling. During the initial obtaining of data, it was seen that many repositories that did not employ *Good First Issue*s on *Github* would use them within mailing lists or upon other platforms. This was especially the case for *Apache*'s[4] projects, which feature communication and task-assignments that take place off of the *Github* platform and usually among their self-hosted alternatives.

Additionally, to increase reproducibility, the sampling methodology should rely on static documented resources. The disadvantage of utilizing the *Github* trending repositories category is that no manner exists to verify whether the repositories indicated are correct. When considering the ability to reproduce the findings of this research, static resources (such as *StackOverflow*'s developer surveys) should be employed. This removes the opportunity or time-limited aspect of the research, thereby increasing reproducibility.

5.2 Limitations Related to Numerical Data

The dataset was utilized for sampling as outlined in Sections 3.1.2 and 3.1.3, which brings additional limitations within the research that must be assessed. These limitations can be considered when drawing conclusions, since the numerical data have the ability to pose as a threat to validity by undoing the measures taken within the sampling procedures discussed in Section 5.1, thereby negatively impacting the research work. To assess the limitations of numerical data, a discussion about the weight of individual repositories for sampling is given in Section 5.2.1, whereas an automatic sampling procedure is outlined in Section 5.2.2.

[2]Website can be found at: `https://about.gitlab.com/`

[3]A list of public repositories can be found at: `https://bitbucket.org/repo/all`

[4]Website can be found at: `https://www.apache.org/`

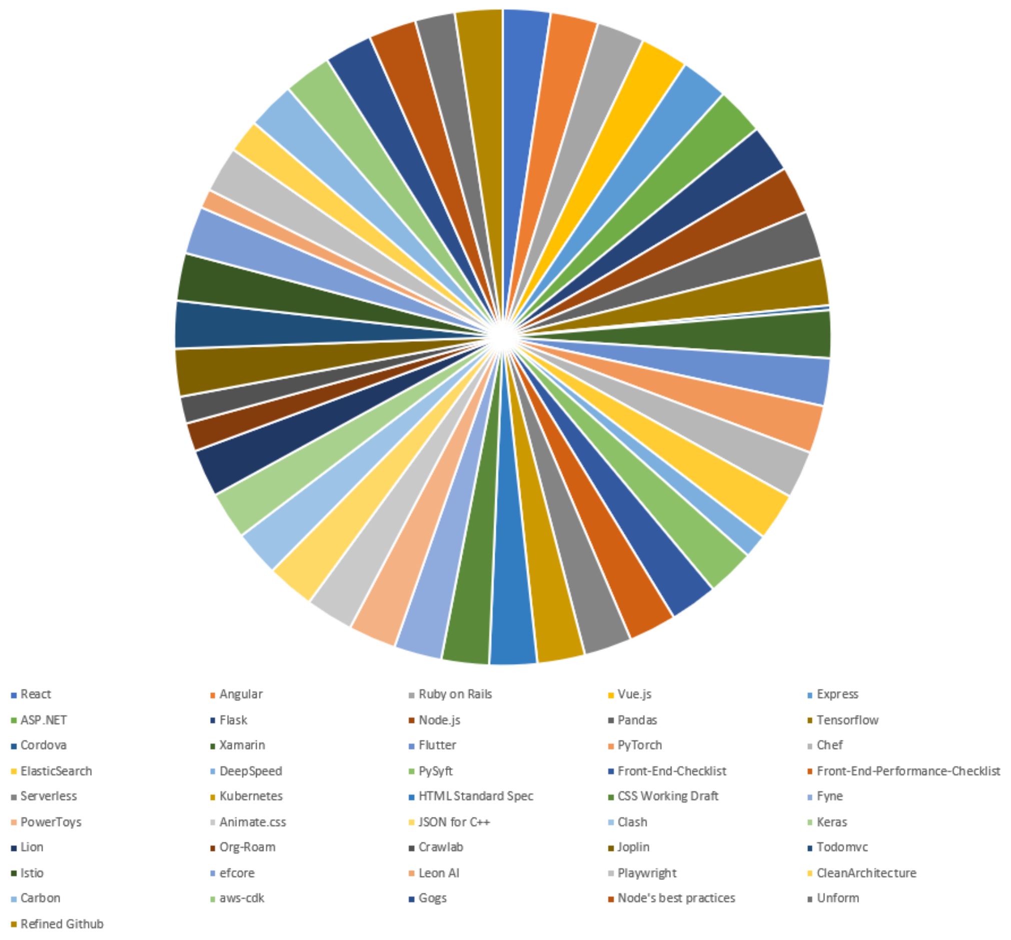


Figure 5.1: Overview of the sampled commit distribution and the sample's commit population per repository.

5.2.1 Weight of Individual Repositories

The sampling procedure relies on a maximum of 30 samples being taken for both issues and commits from each available repository, as to allow the analysis phase to be manageable due to its requirements for manual analysis. This raises the issue in which larger repositories can dominate smaller repositories, possibly impacting the generalizability. If a small subset of repositories dominates the obtained sampling then the analysis can only be applied to those categories of repositories. To investigate this, a pair of repository impact assessment charts was created as shown in Figure 5.1 for commit distribution and Figure 5.2 for issue distribution, which aimed to identify whether this repository domination took place within the sample population.

When considering the commit population, no repository appears to be dominating in any form, with a relatively equal distribution taking place among all repositories. The only projects that are dominated are *Cordova*, *Leon AI*, and *DeepSpeed*, which are a diverse

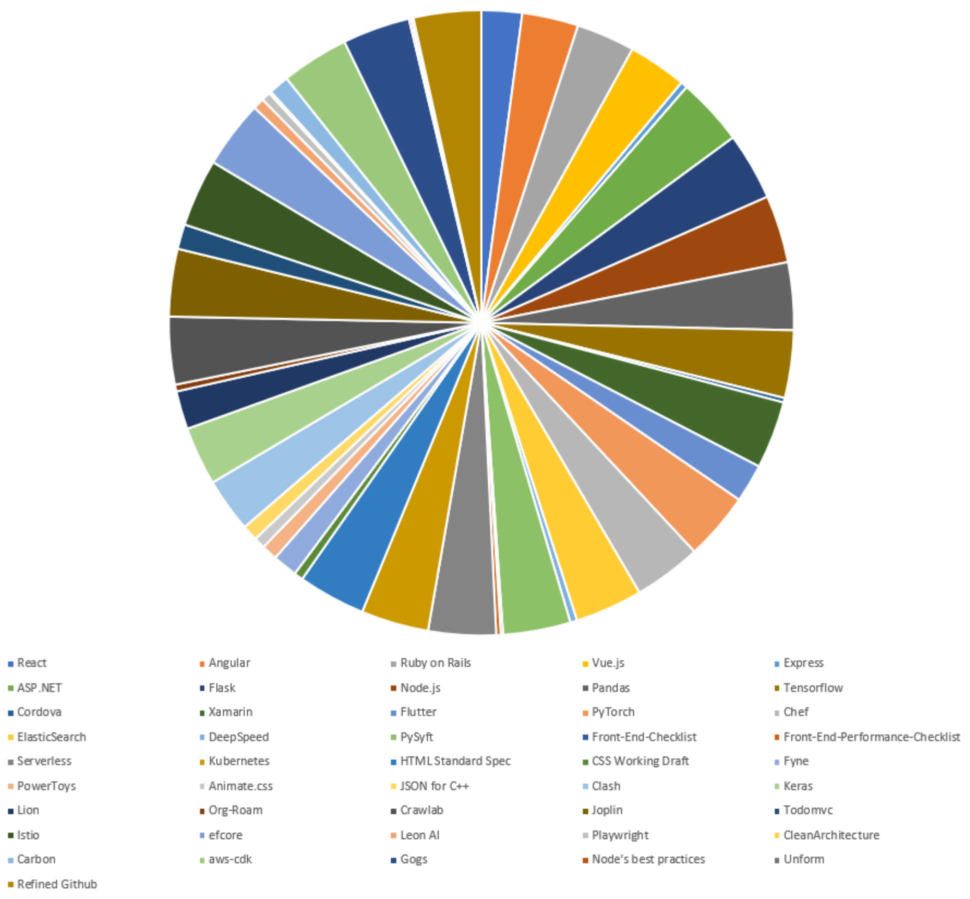


Figure 5.2: Overview of the sampled issue distribution and the sample's provided issue population per repository.

set of dominated repositories, meaning the findings of Section 5.1.2 could still hold. On further inspection, however, it can be calculated that a significant statistical difference exists between the populations, with a p-value of 0,00088. As a result, the commit sampling is biased towards larger projects.

For the sampled issues, there is a greater level of variance. A total of 14 repositories are dominated, consisting of the *Express*, *Cordova*, *DeepSpeed*, *Front-End (Performance) Checklist*, *CSS Working Draft*, *PowerToys*, *Animate.CSS*, *JSON for C++*, *Org-Roam*, *Leon AI*, *Playwright*, *CleanArchitecture*, *Node Best Practices*, and *Unform* projects. It should be noted that most of these repositories are smaller projects, with a small number of contributors. This results in a domination factor of 30,43%, indicating that the sampled *good first issue* population is biased towards larger projects and repositories. This is reinforced further when considering a perfectly distributed population of 46 projects containing 30 sample points each versus the actual sample population, a p-value of 0,000521 is obtained, indicating that the difference is statistically significant.

As a result, it must be stated that the thesis research cannot be generalized for the commit or issue results, as the sample and analyzed population is biased towards larger projects and repositories. It is possible that this influenced the data, since larger projects may have tendencies and processes that effect developer onboarding, possibly causing differing results, and are not generalized as a result. To alleviate this, the sampling work would have to be done again, and a proportion of commits and issues should be obtained per repository. That is, instead of selecting a fixed number of issues and commits per project, a proportion $\frac{1}{n}$ of commits and issues should be taken based on the repository size. In this proportion, n could represent the number of contributors, the number of commits, or another size-based indicator that could discriminate based upon project size.

5.2.2 Automatic Randomized Selection of Commits and Issues

During the sampling of numerical data, a random number generator was outlined as stated in Section 3.1 which generated a set of numbers, indicating commits and *good first issue*s that had to be sampled. These would then be manually obtained from the output CSV files as shown in Figure 3.4. This was a time-consuming process, and although it was done manually with the purpose of verifying the data mining, it was time that would have allowed additional repositories to be sampled and analyzed.

It would be beneficial for any future instances of this research to extend or optimize the scripts developed for this research to do the random selection themselves. This would prevent costly time from having to take place to relocate the sample population into their own CSV files.

5.3 Limitations Related to Analysis

The analysis of this thesis report contains the majority of the findings and proposed conclusions regarding *good first issue*s. This procedure must be analyzed to not only evaluate its generalizability and the validity of its conclusions, but also to identify factors that may influence the outcome of any repeated experiments related to these findings. An explanation and justification of the significance value of 0,01 is given in Section 5.3.1, whereas the manual workload for the analysis is evaluated in Section 5.3.2. An indication of the limited adaptation possibilities of the dataset is given in Section 5.3.3.

5.3.1 Significance Value of 0,01

The decision was made to utilize a significance value of 0,01 to determine statistical significance for multiple analyzed components as opposed to using the standard 0,05 statistical significance value. A significance value of 0,05 is often chosen as part of a tradition within research [8], and a better justification than “*tradition*” should be used. Notions exist which favor the selection of a significance level that matches the effects of a type I or II error [27], and this seems like a fairer criteria to identify a possible significance level. The effects of a type-I error would have a negative effect on this research, but may be considered to be somewhat negligible, since a rejection of an existing correlation can allow it to be identified through other research. The focus must be placed on the prevention of a type-II error occurring, in which a correlation is identified without it actually existing. For this research, that would lead to changes being suggested for *good first issue*s and developer onboarding that would lead to less effective newcomer assistance, a fact which must be avoided. It should also be stated that although this would be a negative consequence it cannot be classified as "*dangerous*," as is possible within medical hypothesis testing. Based upon the findings in the aforementioned works, and based on personal quantification, a decision to utilize a significance value of 0,01 is therefore justified. It prevents incorrect conclusions from being made, but does not discard possible associations to the degree that may be required for medical or financial research.

An additional reason for the selection of 0,01 as a significance level relies around the closely coupled nature of this research. Many classifications presented in the taxonomy feature concepts deriving from the computer science domain, specifically software development. There may be a natural tendency for these to be correlated from the onset, possibly leading to findings that are too general to be conclusive. Thus, selecting a smaller significance value of 0,01 is beneficial to ensure that findings are likely to be due to actual relatedness, and not due to a tight coupling or domain-implied relevance. Possible improvements in this front could include changing the significance value to 0,001 or 0,0001 to see if the findings still hold.

It should be noted that a significance value of 0,05 would have changed the outcome of certain components of this research as shown in Table 5.2. For the direct classification comparison between *good first issue*s and commits, the *refactoring* label would have been deemed statistically relevant as indicated in Table 4.6, which would have allowed *refactoring* tasks to be considered for additional *good first issue*s. Within the analysis of the label combinations, the results in Table 4.15 indicate that *refactoring* for the *novice-intermediate* and *novice-experienced* categories would become statistically significant, possibly explaining the label that loses contributions to the advantage of the *documentation* label. It would also have caused the *new feature* label difference between *novice-experienced* levels to become significant.

5.3.2 Manual Workload for Analysis and Automation

A significant limitation of the scope of this research was caused by the manual workload required to analyze each repository. The decision was made to select 30 issues and 30 commits for each repository, if available, and manually categorize them according to the taxonomy introduced in Sections 3.2.2 and 3.2.3. Additionally, each issue and commit sampled was verified manually to ensure that any additional information, such as emerging labels or possible explanations for anomalies, could be identified. This meant that each repository required 60 manual comparisons to be made, in which certain components of the analysis were more time consuming than others, such as the identification of a pull request related to a *good first issue* or anomalies within an issue itself. This procedure was the reason for the small sample size of the research, and could have been improved by increasing automation on a few points.

The identification of a pull request that is associated to a *good first issue* did not have to

Criterion and Associated Table	p-Value	0,05 Significance?	0,01 Significance?
Refactoring Difference (4.6)	0,016234	Yes	No
New Feature Novice-Experienced (4.15)	0,018	Yes	No
Refactoring Novice-Intermediate (4.15)	0,044	Yes	No
Refactoring Novice-Experienced (4.15)	0,0153	Yes	No

Table 5.2: Aspects of the research and their p-values that would have varied if a significance level of $\alpha = 0,05$ or $\alpha = 0,01$ was chosen.

be done manually, as the *Github* API provides means of identifying pull requests associated with a specific issue. During manual inspection, it was seen that most *good first issue*s either had an empty associated pull request field, indicating the issue was not implemented into the codebase, or an association. This would have saved a lot of time, although the manual verification of links was not without reason. During analysis, it was found that the *Github* API would often associate an incorrect pull request with an issue, possibly leading to incorrect data points for consideration. However, as a starting point, future work should consider directly mining associated pull requests for speedier analysis.

Additionally, the ability to directly analyze pull requests is also provided by the *Github* API. This could have been utilized to obtain an automatic association between *good first issue*s and their associated pull requests, and allow for an additional dimension to be added to the thesis work. However, after experimentation with the pull request components of the API, it was found that they were faulty and unreliable. Multiple pull requests sampled were found to contain links to incorrect pull requests (meaning the title of the sampled request didn't match the link or its contents), and some pull requests were found to be cast as issues as stated within the API. This increase the likelihood of a certain *good first issue* being sampled multiple times, once as a pull request, and once as an issue. As a result of this lack of reliability from the API, the manual analysis was warranted, but if the API is improved or a more reliable method can be identified, the process can be sped up significantly.

The largest removal of manual workload would have been to identify a manner in which the taxonomy could be applied automatically. Within the *Github* API the ability to sample the *labels* array is available for each issue and commit, possibly allowing for automatic application of the taxonomy. It was found during manual analysis, however, that the taxonomy utilized within this research differed from how certain repositories and projects would use their labels. Some repositories state that a *bug fix* can be a refactoring, whereas the act of refactoring would fall within its own label in this taxonomy. To account for these differences, a manual analysis of the keywords within the taxonomy was warranted, but this was the most time consuming component of the analysis.

5.3.3 Adaptation of Analysis

The analysis relies on the taxonomy introduced within Sections 3.2.2 and 3.2.3, and the analysis that was performed is focused on identifying the labels for each of the *good first issue*s and first commits from developers. Although this taxonomy helps identify factors related to this research, the taxonomy is limited in scope and prevents the analysis from being useful beyond the purpose of this research and directly related work. A broader taxonomy or analysis could have provided additional data that might help future related work, and the analysis' limited scope must be stated. However, for the purpose of this research, the analysis' scope and use of taxonomy sufficed.

5.4 Limitations Related to the Survey

The survey that was sent to developers whose work was sampled for either a *good first issue* or a first commit contains a number of limitations that must be considered both for the drawing of conclusions and to understand its purpose. The goal of the survey was not necessarily to be a numerical evaluation, but to identify information and aspects related to *good first issue*s that were not derivable from numerical data. To this extent, the limitations emerging from the sample size are discussed in Section 5.4.1, and the bias from sample attributes and respondent experience is given in Section 5.4.2.

5.4.1 Sample Size

The total sample size of the survey was 22 after correcting for a non-serious respondent with a response rate of 4,09%. This sample size is, as a result, very limited both in terms of the scope of the research and in terms of the developer population as a whole. Thus, the results provided by the survey cannot be seen as representative, and in context of the numerical data, are much less representative. However, the focus of the survey was to obtain insights from developers that were not related to numerical aspects. As a result, the sample size for the survey does not invalidate the findings, since the numerical components can be derived from additional data, and the findings from the survey exist to complement the other findings, and do not serve as the main focus of research.

If numerical data from the survey is to be considered for conclusions related to *good first issue*s, a larger sample size must be obtained. This can act as a window for future work.

5.4.2 Sample Experience and Attributes

To conform to the ethical requirements as stated in Section 3.3.3, no personal or identifiable information was obtained from survey respondents. However, when considering the distribution of developer experience as shown in Section 4.5 and given in Table 4.11, it can be stated that the survey's respondents were more likely to be developers with some form of prior experience under the *experienced* category. Based upon the results given and the answers given by certain individuals, a few of whom had the perspective of a project integrator, these assumptions are further reinforced.

As a result, the survey results should be interpreted differently with a focus on understanding that experienced developers gave additional insights, not the *larger* developer population. Reformulating the results as presented in Section 4.6 to acknowledge this can help place them into the appropriate perspective and prevent nullification of these findings due to sample bias.

5.5 Threats to Validity

Threats to validity can influence the extent to which the conclusions of the findings outlined in Chapter 4 can be applied. Accounting for these threats to validity and identifying possible points of improvement can help indicate areas for future research or possible weaknesses in the methodology. To account for these issues, construct threats are discussed in Section 5.5.1, internal threats are discussed in Section 5.5.2, and external threats to validity are discussed in Section 5.5.3.

5.5.1 Construct Threats

Construct threats to validity are related to incorrect measurements or test methodology. These threats can decrease the validity of the identified conclusions or yield incorrect results.

The taxonomy that was created for this research was built and constructed to be as broad as possible, but was constructed for this research. As a result, the metric obtained for the taxonomy are limited to only this research and are not generalized from a broader framework. Despite the best efforts to make this framework as broad as possible, disagreements are possible among certain categories for the analyzed classifications. This may yield differing conclusions between reproducers of this research. To accommodate this, the research should be repeated in its current state by others, while also having the research be repeated with existing categorization frameworks or taxonomies, such as domain-specific taxonomies [49] or the Buckley taxonomy [4].

The decision to utilize a significance threshold of $0,01$ was outlined in Section 5.3.1 resulted in multiple conclusions being declared as insignificant, as their p-value was greater than 0,01 but less than 0,05. Had it been decided to utilize a greater value, the conclusions would be different. To accommodate for this, the identified p-values are given in the tables in Chapter 4 such that any possible disagreement considering the significance threshold decision justified in the aforementioned section can draw the conclusions under those assumptions.

5.5.2 Internal Threats

Internal validity refers to the extent to which the cause-effect relationships identified in this thesis research are valid. If these relationships are not valid or cannot be generalized, the conclusions that can be drawn from this research are weaker.

Due to the manual time and complexity constraints outlined in Section 5.3.2, a set of at most 30 *good first issue*s and 30 first commits were analyzed per repository, adding a degree of randomness to the analysis. Although this randomness and the number of sampled content should be sufficient, it makes reproducing the research difficult as it is difficult to get the exact same sample. To alleviate this concern, two lists consisting of sampled issues and commits were created and provided publicly to allow anyone to validate and reproduce the findings of this research[5].

Many of the conclusions and identified relationships within this research relied on a small sample size and had to be validated for statistical significance to accommodate for this. By employing association rule mining and by determining the significance through both effect weighting and the employment of Mann-Whitney U testing, these limitations were considered and put into perspective. As a result of these tests, multiple possible outcomes were invalidated or nullified due to a lack of significance or a lack of effect. This helped place the findings into the required perspective and helped ensure that conclusions derived from the research are representative and not a result of a small sample size.

5.5.3 External Threats

External threats to validity consist of issues related to the representation and generalizability of the study. These may cause the overall representativeness and applicability of the identified conclusions to be limited.

When considering the sample set as outlined in Section 5.1.1, the size of the dataset was small due to time constraints. To account for this, random sampling was utilized with the goal of still creating a representative dataset. Future research should focus on repeating this research with a larger set of repositories to identify whether conclusions still hold for a larger dataset. Additional effort was made to ensure the dataset was representative and not biased towards a single domain or sub-domain as outlined in Section 5.1.2, but future research should be done to ensure conclusions hold and, if new domains were to emerge within programming or computer science, to see if these conclusions extend to those domains.

The platform upon which the research was done was limited to *Github* due to both time constraints and due to the application of issues upon the platform, meaning it cannot be stated that *good first issue* adoption rate and completion are identical on other platforms. Some work was done to identify *good first issue*s upon external and self-hosted platforms, but not to a degree that provides certainty that the research can be generalized. Future work should consider alternative platforms to ensure that the research can be generalized to the software development community as a whole.

As indicated in Section 5.2.1, this research is biased towards larger projects and repositories due to a larger sample of *good first issue*s and first commits being taken per project. This can result in the results being valid for only larger projects, but falling short or not being upheld when analyzing smaller repositories and projects. To improve this for fu-

[5]Lists can be found at: `https://github.com/dalderliesten/Good-First-Issue/tree/master/Analysis`

ture research, a ratio should be utilized of sampled and analyzed content per repository as opposed to a fixed upper limit.

Chapter 6

Conclusions and Future Work

This chapter contains the conclusions of the thesis research based upon the results found in Section 4 and the evaluation over those results provided in Section 5. The conclusions and findings drawn from this research based upon the research questions introduced in Section 1.3 are given in Section 6.1, suggestions based upon these conclusions are given in Section 6.2, and suggestions for possible future work are given in Section 6.3.

6.1 Findings

To study the usefulness of *good first issue*s and to identify areas of improvement that were possible, repositories were analyzed and their associated data were mined such that an assessment could be made of the impact of *good first issue*s upon developer onboarding. A number of research questions ranging from adoption rate to developer perception were included to achieve this goal. Based upon the sampling work done, it was found that out of 105 repositories a total of 46 utilized *good first issue*s, indicating an adoption rate of 43,8%. For each of the 46 sampled repositories the total body of *good first issue*s was also mined and stored, indicating that a total of 4.792 *good first issue*s existed across 46 repositories. The total body of issues found within the sampled projects amounts to 301.380, indicating that *good first issue*s represent 1,5% of the total issue population. These findings suggest that *good first issue*s see widespread adoption and are a known and well-employed method of onboarding new developers to a project.

RQ1 - Result
What percentage of projects within the Github ecosystem employ good first issue labels for their issues?

Based on the sampling methods utilized, 43,8% of repositories & projects on the *Github* platform utilize *good first issue*s as a label for newcomer task recommendation.

RQ2 - Result
What percentage of issues within repositories are labelled as good first issues

out of the total number of issues?

Based on the samples obtained, *good first issue*s represent 1,5% of the total issue set on *Github*.

Furthermore, an analysis was made in which manual application of a taxonomy was done towards at most 30 individual *good first issue*s per project to identify the types of tasks that are suggested for newcomers by project maintainers and staff. When considering the types of tasks that are suggested as *good first issue*s, it was found that 29,2% of *good first issue*s focus on the fixing of *bugs* and issues that cause unintended behavior, 24,2% of *good first issue*s require a developer to contribute an *enhanced feature* to the project that extends existing functionality, 19,9% consist of *refactoring*-based contributions that change the codebase to provide greater efficiency or easier maintainability, and 18,5% of *good first issue*s focus on the creation or editing of *documentation* within a project. Additionally, 12,7% of *good first issue*s recommend a newcomer developer a *new feature* that does not yet exist within the codebase, and 8,7% of *good first issue*s recommend that newcomers perform tasks related to *testing*. This indicates that project maintainers and those carrying the responsibility for *good first issue* recommendation see *bug fixes*, *feature enhancements*, and *documentation* as suitable tasks for newcomers.

RQ3 - Result

What types of tasks and issues are generally recommended as good first issues?

Within the sample population, the most recommended tasks for newcomers in the form of *good first issue*s consist of *bug fixes* with 29,2%, *feature enhancements* with 24,2%, and *documentation* with 19,9%.

In addition to the types of tasks developers were completing, an analysis was done to identify the rate at which *good first issue*s were completed by newcomers as opposed to existing developers. It was found that of the 858 *good first issue*s sampled, 279 were completed by new developers, whereas 340 were completed by existing developers of a project. 239 *good first issue*s were not completed or deprecated. When accounting for the deprecated issues, a total of 45,07% of *good first issue*s are completed by newcomer developers, indicating a high percentage. This suggests that *good first issue*s are used and are effective at onboarding new developers, considering that almost half of them are taken by newcomers.

RQ4 - Result

Do new developers complete the tasks labelled as good first issues?

When weighting for completed tasks, 45,07% of *good first issue*s are completed by developers that are new to a project.

The aforementioned issue analysis was repeated for a set of 30 sampled initial contributions for a developer per project to identify what types of tasks new developers have a tendency to contribute. The types of tasks that were found to be contributed most often were *documentation*-related tasks consisting of 35,1% of the total share, followed by *refactoring* contributions at 24,3% and *bug fixes* at 21,3%. Newcomers would *enhance a feature* in 17,1% of the initial contributions, with entirely *new feature*s being contributed at 10,8% of the time. The lowest category assigned to first contributions was the *testing* label at 5,3%. When comparing these contributions to the labels applied to tasks, there is a significant difference between the *bug fix*, *documentation*. *feature enhancement*, and *testing* labels. Analysis indicates that *documentation* and *refactoring* contributions occur more often than their associated labels, whereas *bug fix* and *feature enhancements* are suggested more often than they are contributed. Combinations of labels were also studied, finding that although *good first issue* labelled tasks did not have combination tendencies, commits and contributions by newcomers featuring a *bug fix*, *refactoring*, or a *feature enhancement* were likely to also include *testing* related contributions.

Additional analysis was done into the effect of developer experience upon the initial contribution that is made to a project. Findings indicate that developers with more than a year of experience tend to show no significant deviation for any certain taxonomy label for their first contribution, but *novice* developers with less than a year of experience were found to favor *documentation* related tasks for their initial contribution, a difference that was found to be statistically significant. Although no statistically significant difference was found in other categories, findings suggest these *documentation* contributions for *novice* developers come at a cost of *new feature* and *refactoring* related contributions.

RQ5 - Result

What types of contributions are made by newcomer developers within a project?

Within the sample population, initial contributions to a project by newcomers consist mostly of *documentation* related changes at 35,1%, *refactoring*-based changes at 24,3%, or *bug fixing* contributions at 21,3%. When a contribution is related to a *bug fix*, *refactoring*, or *feature enhancement* it is also likely to contain some form of *testing*. Developers with less than a year of development experience were found to disproportionately favor *documentation*-based contributions for their initial commit.

To investigate the developer perception of *good first issue*s and task labelling for newcomers, a survey was sent out which yielded a perceptive numerical usefulness value from a developer viewpoint of 70,45, indicating that developers find *good first issue* labelling and newcomer task suggestion to be a worthwhile endeavour. Approximately half of respondents that contributed through a *good first issue* indicated that they utilized *good first issue*s to select their first task, with many developers that did not employ them stating that external requirements often drive their initial contribution, and not suggested tasks. When questioned which alternatives could be used to *good first issues*, developers mostly reiterate labels as being an effective means at 53,57%, followed by a suggestion to utilize a *README* document at 25%, indicating developers do find newcomer task suggestions to be useful.

RQ6 - Result
How do (new) developers perceive the labelling of good first issues and their usefulness?

Developers rate *good first issue* utilization within a project with an average score of 70,45. Developers state that contributions to a project are usually driven by external requirements or personal interests, and not necessarily task suggestion.

When considering these individual components, it can be stated that the *good first issue* label is effective at indicating tasks within a software project that are suitable for newcomers, but that there is room for improvement by performing additional vetting of the task types according to the discrepancies in the taxonomy. Overall, *good first issue*s appear to help onboard new developers to projects and to point them in the direction of suitable tasks.

MRQ - Result
Is the good first issue label effective in indicating tasks within an open (source) software project that are taken by newcomers?

*Good first issue*s are effective at indicating tasks that are suitable for newcomers, but adjustments need to be made when considering developer experience and the type of tasks suggested to greater align them with newcomer preferences. Inexperienced developers would benefit from seeing a larger focus on *documentation* related tasks. Overall, a larger focus should be placed upon *documentation* and *refactoring* related tasks, as these are contributed more often by newcomers than they are suggested by *good first issue* labelled tasks.

6.2 Suggestions for *Good First Issue* Improvements

The findings given in Section 6.2 indicate a number of suggestions based off of this research related to *good first issue*s and newcomer onboarding. These suggestions can be adopted by (open-source) repositories with the goal of increasing the rate at which newcomers join the project, and to help with the selection of suitable newcomer tasks.

Good First Issue Utilization: Findings of this research indicate that *good first issue*s and task recommendation for newcomers works, with almost half of the tasks having been completed by newcomers and developer perception of task recommendation being rated with a score of 70,45. Projects and repositories that do not feature some form of newcomer task recommendation are potentially seeing decreased level of developer onboarding, and are encouraged to start utilizing *good first issue*s.

Types of Task Labelled: When considering the types of tasks that should be labelled as a *good first issue*, there is a discrepancy between the types of tasks that are labelled as good for newcomers and actual newcomer contributions. Projects employing task recommendation or *good first issue*s should aim to focus on tasks related to *documentation* or minor *refactoring* labels to ensure a greater rate of developer onboarding through issue labelling. Projects should also aim to provide less *bug fixing* and *feature enhancement* tasks in favor of the aforementioned labels, as these tend to be contributed to a lesser degree by newcomers.

Increasing Initial Test Contributions: Projects and repositories that wish to see more *test* related contributions from new developers need to couple these tasks with *bug fix*, *refactoring*, or *feature enhancement* tasks. The focus of these *good first issue*s must rest upon the *bug fix*, *refactoring*, or *feature enhancement*, as the *test* related contribution has a tendency to be provided as a result of these contributions, not as an antecedent.

Increasing Novice Developer Onboarding: *Novice* developers were found to have a different initial contribution pattern than *intermediate* or *experienced* developers due to their preference for *documentation* related tasks. Projects and repositories looking to increase the number of *novice* developers, at which developers with little to no development experienced are referred to, should focus on creating additional *good first issue*s that require a *documentation*-related addition or change to be contributed.

Utilizing Social Media: Developers indicated in the survey that using external communication channels, such as social media, to communicate suitable tasks for newcomers. This is already being done with the *Twitter* handle *@goodfirstissue*[1], but could be adopted by more projects. This would allow newcomers that may not be within the *Git(hub)* ecosystem to participate and identify tasks they could contribute, possibly increasing the overall rate of onboarding.

6.3 Future work

Multiple issues were identified in the evaluation as given in Chapter 5, and certain additional angles of possible research were identified within the conclusions. This section aims to provide a number of these alleys for future work.

Improving or Differing the Sample Set: The sample size and set for this research were limited due to selection criteria. An improvement can be made by repeating this research with a larger sample set, possibly by obtaining a randomized set of *Github* repositories and utilizing more sources than only *StackOverflow*'s developer survey and the *Github* trending page. The sample set for this research was also biased due to external occurrences, such as the *COVID-19* repositories taken for sampling. A better procedure can help increase the applicability of the research. Additionally, attempting to reproduce the findings of this research with a different dataset would help affirm its reproducibility. Another focus should be placed upon re-sampling in which each repository or project contributes a proportion of

[1]Account can be found at: `https://twitter.com/goodfirstissue`

its *good first issue*s and first contributions to be analyzed as opposed to a fixed limit of 30 each per project, as this was found to create a bias in favor of large repositories for this research.

Additional Hosts: The research was limited to the *Github* platform, but additional public free hosts for *Git* repositories exist, such as *Gitlab* and *BitBucket*. The research should be repeated for these additional hosts to identify whether the trends and findings of this thesis research apply generally or are more limited to the scope of *Github*.

Automated Analysis: The manual analysis utilized within this research required a significant time investment and prohibited a larger sample set from being studied within the given time constraints. A large shortcoming within this research was the limited sample size for both *good first issue*s and commits, and these constraints were in place due to the amount of manual labor required per issue and commit analyzed. If this process can be done automatically, the research can be done to a sample size that is more representative. Additionally, automatic analysis may identify certain relationships between developer or task aspects that were not caught in this research due to the manual nature of the analysis.

Investigating Recurring Contributions: The focus of this thesis report was upon the investigation of first contributions, but the consequences of the type of contribution made was not investigated. Future research could attempt to identify whether a certain taxonomy label or preference yields consequential results. For example, it may be found that *novice* developers that make an initial contribution consisting of a *new feature* are more likely to contribute *testing* related changes for their secondary or tertiary contribution. These investigations can help identify whether projects and repositories might want to propose certain tasks more often than others due to their domain. A repository that is in need of core developers may then want to focus on contribution types that cause newcomers to make *new feature* or *feature enhancement* contributions over time.

Studying Label Effectiveness: One assumption of this thesis work was that labelling is effective at a global scope, focusing on identifying the effectiveness of *good first issue*s within the broader issue population. However, an interesting follow-up could be to identify if task labelling is effective at all, effectively removing the focus on initial contributions. Perhaps labelling does not correlate to increase contributions of those types of tasks, or perhaps developers do not care for task labelling. This could help identify whether labelling is a worth investment of time, and if alternatives need to be utilized. This type of research could also help put this research work into a larger perspective, and could possibly act as an extension to Labuschagne and Holmes' [30] work.

Bibliography

[1] John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Trans. Softw. Eng. Methodol.*, 20(3), August 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000794. URL `https://doi.org/10.1145/2000791.2000794`.

[2] Andrew Begel and Beth Simon. Novice software developers, all over again. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, page 3–14, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582160. doi: 10.1145/1404520.1404522. URL `https://doi.org/10.1145/1404520.1404522`.

[3] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367, 2017.

[4] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005. doi: 10.1002/smr.319. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.319`.

[5] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. Developer onboarding in github: The role of prior social links and language experience. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 817–828, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786854. URL `http://doi.acm.org/10.1145/2786805.2786854`.

[6] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. Developer onboarding in github: The role of prior social links and language experience. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 817–828, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786854. URL `https://doi.org/10.1145/2786805.2786854`.

[7] Stefano Comino, Fabio M. Manenti, and Maria Laura Parisi. From planning to mature: On the success of open source projects. *Research Policy*, 36(10):1575 – 1586, 2007. ISSN 0048-7333. doi: https://doi.org/10.1016/j.respol.2007.08.003. URL `http://www.sciencedirect.com/science/article/pii/S0048733307001709`.

[8] Michael Cowles and Caroline Davis. On the origins of the. 05 level of statistical significance. *American Psychologist*, 37(5):553, 1982.

[9] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1086-4. doi: 10.1145/2145204.2145396. URL `http://doi.acm.org/10.1145/2145204.2145396`.

[10] Marco D'Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. *Analysing Software Repositories to Understand Software Evolution*, pages 37–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-76440-3. doi: 10.1007/978-3-540-76440-3_3. URL `https://doi.org/10.1007/978-3-540-76440-3_3`.

[11] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486844`.

[12] O. Elazhary, M. Storey, N. Ernst, and A. Zaidman. Do as i do, not as i say: Do contribution guidelines match the github contribution process? In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 286–290, 2019.

[13] F. Fagerholm, A. Sanchez Guinea, J. Borenstein, and J. Münch. Onboarding in open source projects. *IEEE Software*, 31(6):54–61, Nov 2014. ISSN 1937-4194. doi: 10.1109/MS.2014.107.

[14] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, page 1–9, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328562. doi: 10.1145/2593812.2593813. URL `https://doi.org/10.1145/2593812.2593813`.

[15] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 26–33, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135947. doi: 10.1145/585058.585065. URL `https://doi.org/10.1145/585058.585065`.

[16] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C. Murphy, and Jean-Rémy Falleri. Impact of developer turnover on quality in open-source software. In *Proceedings*

of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 829–841, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786870. URL http://doi.acm.org/10.1145/2786805.2786870.

[17] Golara Garousi, Vahid Garousi-Yusifoğlu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664 – 682, 2015. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2014.08.003. URL http://www.sciencedirect.com/science/article/pii/S095058491400192X.

[18] Marko Gasparic and Andrea Janes. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software*, 113:101 – 113, 2016. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.11.036. URL http://www.sciencedirect.com/science/article/pii/S0164121215002605.

[19] G. Gousios, M. Storey, and A. Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 285–296, May 2016. doi: 10.1145/2884781.2884826.

[20] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL http://dl.acm.org/citation.cfm?id=2487085.2487132.

[21] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 358–368, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818800.

[22] Jungpil Hahn, Jae Yun Moon, and Chen Zhang. Emergence of new project teams from open source software developer networks: Impact of prior collaboration ties. *Information Systems Research*, 19(3):369–391, 2008. doi: 10.1287/isre.1080.0192. URL https://pubsonline.informs.org/doi/abs/10.1287/isre.1080.0192.

[23] A. E. Hassan. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57, Sep. 2008. doi: 10.1109/FOSM.2008.4659248.

[24] Kim Herzig and Andreas Zeller. *Mining Your Own Evidence*, chapter 27. O'Reilly Media, Inc., October 2010. ISBN 9780596808327.

[25] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings*

of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pages 426–437, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970358. URL `http://doi.acm.org/10.1145/2970276.2970358`.

[26] D. Izquierdo-Cortazar, G. Robles, F. Ortega, and J. M. Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10, Jan 2009. doi: 10.1109/HICSS.2009.498.

[27] Jae Kim. How to choose the level of significance: A pedagogical note. `https://mpra.ub.uni-muenchen.de/69992/`, 2015.

[28] K. Kohl Silveira, S. Musse, I. H. Manssour, R. Vieira, and R. Prikladnicki. Confidence in programming skills: Gender insights from stackoverflow developers survey. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 234–235, May 2019. doi: 10.1109/ICSE-Companion.2019.00091.

[29] Mehmet Kosa and Murat Yilmaz. Gamifying the onboarding process for novice software practitioners. In Christian Kreiner, Rory V. O'Connor, Alexander Poth, and Richard Messnarz, editors, *Systems, Software and Services Process Improvement*, pages 242–248, Cham, 2016. Springer International Publishing. ISBN 978-3-319-44817-6.

[30] A. Labuschagne and R. Holmes. Do onboarding programs work? In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 381–385, 2015.

[31] Sang-Yong Tom Lee, Hee-Woong Kim, and Sumeet Gupta. Measuring open source software success. *Omega*, 37(2):426 – 438, 2009. ISSN 0305-0483. doi: https://doi.org/10.1016/j.omega.2007.05.005. URL `http://www.sciencedirect.com/science/article/pii/S0305048307000898`.

[32] N. Li, W. Mo, and B. Shen. Task recommendation with developer social network in software crowdsourcing. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 9–16, 2016.

[33] B. Lin, G. Robles, and A. Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*, pages 66–75, May 2017. doi: 10.1109/ICGSE.2017.11.

[34] C. Liu, D. Yang, X. Zhang, B. Ray, and M. M. Rahman. Recommending github projects for developer onboarding. *IEEE Access*, 6:52082–52094, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2869207.

[35] Chao Liu, Dan Yang, Xiaohong Zhang, Haibo Hu, Jed Barson, and Baishakhi Ray. A recommender system for developer onboarding. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 319–320, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356633. doi: 10.1145/3183440.3194989. URL `https://doi.org/10.1145/3183440.3194989`.

[36] W. Ma, L. Chen, Y. Zhou, and B. Xu. Do we have a chance to fix bugs when refactoring code smells? In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 24–29, 2016.

[37] K. Mao, Y. Yang, Q. Wang, Y. Jia, and M. Harman. Developer recommendation for crowdsourced software development tasks. In *2015 IEEE Symposium on Service-Oriented System Engineering*, pages 347–356, 2015.

[38] Martin Michlmayr and Benjamin Mako Hill. Quality and the reliance on individuals in free software projects. In *in 3rd Workshop on Open Source Software Engineering*, pages 105–109, 10 2011.

[39] The Executive Board of Delft University of Technology. Tu delft regulations on human trials, 2016. URL `https://d1rkab7tlqy5f1.cloudfront.net/TUDelft/Over_TU_Delft/Strategie/Integriteitsbeleid/Research%20ethics/HREC-Articles_of_Association.pdf`.

[40] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185, 2017.

[41] Raphael Pham, Stephan Kiesling, Leif Singer, and Kurt Schneider. Onboarding inexperienced developers: Struggles and perceptions regarding automated testing. *Software Quality Journal*, 25(4):1239–1268, December 2017. ISSN 0963-9314. doi: 10.1007/s11219-016-9333-7. URL `https://doi.org/10.1007/s11219-016-9333-7`.

[42] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, 28(6):595–606, 2002.

[43] Gregorio Robles and Jesus M. Gonzalez-Barahona. Contributor turnover in libre software projects. In Ernesto Damiani, Brian Fitzgerald, Walt Scacchi, Marco Scotto, and Giancarlo Succi, editors, *Open Source Systems*, pages 273–286, Boston, MA, 2006. Springer US. ISBN 978-0-387-34226-9.

[44] Y. Saito, K. Fujiwara, H. Igaki, N. Yoshida, and H. Iida. How do github users feel with pull-based development? In *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 7–11, 2016.

[45] Hugo H. Schoonewille, Werner Heijstek, Michel R.V. Chaudron, and Thomas Kühne. A cognitive perspective on developer comprehension of software design documentation. In *Proceedings of the 29th ACM International Conference on Design of Communication*, SIGDOC '11, page 211–218, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309363. doi: 10.1145/2038476.2038517. URL `https://doi.org/10.1145/2038476.2038517`.

[46] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press. ISBN 9781450355735. doi: 10.1145/3236024.3264598. URL `http://dl.acm.org/citation.cfm?doid=3236024.3264598`.

[47] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work &#38; Social Computing*, CSCW '15, pages 1379–1392, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2922-4. doi: 10.1145/2675133.2675215. URL `http://doi.acm.org/10.1145/2675133.2675215`.

[48] Ewan Tempero, Tony Gorschek, and Lefteris Angelis. Barriers to refactoring. *Commun. ACM*, 60(10):54–61, September 2017. ISSN 0001-0782. doi: 10.1145/3131873. URL `https://doi.org/10.1145/3131873`.

[49] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology*, 85:43 – 59, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.01.006. URL `http://www.sciencedirect.com/science/article/pii/S0950584917300472`.

[50] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 401–405, Sep. 2014. doi: 10.1109/ICSME.2014.62.

[51] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 805–816, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786850. URL `http://doi.acm.org/10.1145/2786805.2786850`.

[52] G. Viviani and G. C. Murphy. Reflections on onboarding practices in mid-sized companies. In *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 83–84, 2019.

[53] Mike Volpi. How open-source software took over the world, Jan 2019. URL `https://techcrunch.com/2019/01/12/how-open-source-software-took-over-the-world/`.

[54] Jianguo Wang. Supporting developer-onboarding with enhanced resource finding and visual exploration, 2012. URL `https://digitalcommons.unl.edu/csetechreports/146/`.

[55] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 367–371, May 2015. doi: 10.1109/MSR.2015.42.

[56] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[57] Minghui Zhou and Audris Mockus. Developer fluency: Achieving true mastery in software projects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 137–146, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587912. doi: 10.1145/1882291.1882313. URL `https://doi.org/10.1145/1882291.1882313`.

Appendix A

Glossary

This appendix provides an overview of frequently used terms and abbreviations that are used within this thesis report. These terms are meant to provide the reader with a better understanding of the concepts introduced and discussed within the thesis report.

Accidental Sampling: a sampling technique in which samples are chosen based on opportunity, and not on probabilistic factors.

Active Experience: a period of time for which a user made a certain number of commits to be defined as active. Within the context of this research, a developer was deemed active if they made at least 10 contributions within a year.

API: abbreviation for *application programming interface*.

Application Programming Interface: a set of tools that allow simplified communication or data passing.

Branch: a developing part of a codebase that is maintained in isolation until it is ready to be added to the *master* branch.

Build: an instance of *continuous integration* in which all tests and associated procedures are executed. Can be seen as a single *run*.

Centralized Version Control: a version control variant in which one central location maintains a *master* copy of the repository, which developers can propose adjustments for.

Closed Source Software: software that cannot be publicly inspected or viewed.

Code: the content that makes up a software application.

Codebase: a term referring to all code for a certain software application.

Code Smell: term used to indicate stylistic or organizational issues within a code, such as a lot of lines within a class or method.

Comma Separated Value: a type of file for the storage of a spreadsheet or data that separates data entries and fields through the utilization of commas.

Company Sttering: a repository or project that is maintained and/or guided by a company.

Contributor: a person contributing to an open source software project, see also *developer*.

Continuous Integration: a term used to describe automated processes within a pull-based development model.

CI: see *continuous integration*.

CSV: abbreviation for *comma separated value*.

Δ Difference: The difference between two percentile statistics, when considering the first statistic value as the baseline.

Developer: a person who contributes to an open source project or piece of software.

Distributed Version Control: a version control variant in which each developer maintains a local copy of the *master* copy of the repository and updates this over time.

Drill: the action of searching through a large dataset for specific data or queries.

End User: a person that uses software.

Experienced Developer: a developer with more than two years of experience on *Github*.

Git: a version control system developed by Linus Torvalds.

Github: a popular host of (free) Git repositories.

Good First Issue: a label given to tasks that are indicated to be simple or good for new developers within a project.

Intermediate Developer: a developer with more than a year and less than two years of experience on *Github*.

Issue: term given to a task stored within a repository in the *Github* ecosystem.

Medior: a term utilized for the research component of the project indicating a developer of *intermediate* level with between one and two years of developing experience.

Merge: the process of combining one *branch* into another *branch*.

Mining: see *software repository mining*.

MRQ: abbreviation for *main research question*.

Novice Developer: a developer with less than a year of experience on *Github*.

Onboarding: the process of actively finding and involving new developers within a project.

Open Software: a software project that is developed like an open source project but comes from or is owned by a for-profit corporation.

Open Source Software: a software project developed by uncompensated developers on their own accord with the ability to view, modify, and inspect the codebase publicly.

OSS: abbreviation for open source software.

Pull Request: code reviews in which comments and specific lines of code can be highlighted for evaluation and discussion.

Private Steering: a repository that is not maintained and/or guided by a company.

Project: term used to refer to an entity that produces content, organizes, or works in a *repository*.

Repository: a centralized location in which code or a codebase is stored.

Script: an alternative term for a code segment within the codebase.

Software Repository Mining: the process of extracting data related to software development from a repository.

RQ: abbreviation used to denote a *research question*.

Secondary Contribution: a contribution made after the first contribution to a project.

Smoke Test: a test which aims to test the overall functionality of a program versus only a small component of a program.

Task: an operation that must be performed by a developer.

Trending Repository: a repository that is shown on *Github*'s trending repositories page.

URL: abbreviation for a hyperlink.

Version Control: a system which manages changes within a codebase to allow multiple developers to work together upon a codebase.

Appendix B

Dataset

This appendix provides hyperlinks to relevant aspects of the generated dataset for this research.

Repository: The repository can be found on *Github* at:
`https://github.com/dalderliesten/Good-First-Issue`.

Sampled Repositories: The sampled repositories can be found as a *CSV* file at:
`https://github.com/dalderliesten/Good-First-Issue/blob/master/Sample%20Set/Sampled-Repositories.csv`

Repositories with *Good First Issue*s: The repositories sampled containing *good first issue*s can be found in a *CSV* file at:
`https://github.com/dalderliesten/Good-First-Issue/blob/master/Sample%20Set/Repositories-with-Good-First-Issues.csv`

Codebase: The codebase developed for this research is provided on *Github* as *Python* code at:
`https://github.com/dalderliesten/Good-First-Issue/tree/master/Code`

Raw Dataset: The obtained, non-analyzed data is stored as sets of *CSV* files and can be found at:
`https://github.com/dalderliesten/Good-First-Issue/tree/master/Data`

Analyzed Data: The analyzed data is given as pairs of *CSV* files and a number of overview files that can be found at:
`https://github.com/dalderliesten/Good-First-Issue/tree/master/Analysis`

Templates: The templates that were used for analysis are given as both *Excel XLSV* files and *CSV* files and can be found at:
`https://github.com/dalderliesten/Good-First-Issue/tree/master/Analysis`