Final Report

Course: TI2806

Context Project: Tools for Software Engineering

GitHub Division

By The
Rubber Duck Debuggers

23 June 2016

Youri Arkesteijn	-	yarkesteijn (4357876)
Mitchell Dingjan	-	mdingjan (4348516)
Robin van Heukelum	-	rvanheukelum (4369815)
Mathias Meuleman	-	mmeuleman (4375629)
Maarten Sijm	-	msijm (4361083)

SE TA: Bastiaan Reijm
Context TA: Aaron Ang

Context Teacher: Alberto Bacchelli

Lectori salutem,

In front of you lies the final report on our division of a software product that we call ‘Octopeer’. The product has been developed, implemented and validated at the Technical University of Delft by us, a group of five enthusiastic students and their rubber duck, that call their team the ‘Rubber Duck Debuggers’. This final report has been written as the final deliverable for the Context Project, which is a second years course of the bachelor ‘Computer Science’, that wraps up what has been achieved during the during the project. We’ve worked on this project from 18 April 2016 until 24 June 2016.

The problem for this context has been provided by Dr. Alberto Bacchelli, our context teacher, and Aaron Ang, our context teaching assistant. This is the problem that we’ve been working on during the just described timeframe and from which resulted our software product. During the project, our software engineering teaching assistant, Bastiaan Reijm, has supported us on the software engineering part of our product. Both Aaron and Bastiaan have provided feedback on a weekly basis on our software product in the most general sense of it; from details of the context perspective to details of the software engineering perspective. Especially in the last couple of weeks of the project we’ve had several meetings with Dr. Bacchelli in which he provided us with feedback on the entire software product as well.

We’d like to thank our context teacher and our teaching assistants for their guidance and support during this process. Also, we’d like to thank the students and teaching assistant that have participated in the user tests for our human-computer interaction research. All of their feedback has been taken into account and has improved the process and product even further.

We’ve learned a lot during this project and we hope that you will too by reading this report.

On behalf of the Rubber Duck Debuggers,
Mitchell Dingjan

Delft, 22 June 2016

Table of Contents

Abstract

In this document, you will find information about the developed, implemented and validated software product. The software product's set of users can be divided into two subsets: software engineers, or more specific: code reviewers, and researchers. These users have their own requirements, which are taken into account during the development process. The product itself is separated into the front end and the back end of which the back end can be separated into the content page and the background page. The content page part covers data selection from the pull request pages and the background page part manages the logging of the data to the database. The front end part is mostly related to the user and for this a human-computer interaction research has been performed and documented.
From a software engineering perspective, the product has been well designed and well implemented and the process has been well organised. Also, when reflecting on the product, all must-have and should-have requirements are implemented and the main requirements are taken into account, so we're happy with the implemented product.
The product is also very extensible and can be used in researches. The product can be extended by for example adding a feature for generating feedback or creating a more advanced visualisation for the visualisation plugin.

1. Introduc(k)tion

In the first week of the project we (and two other groups) were given the following description of what had to be achieved with the to be developed product:

"The goal of Octopeer is to model behaviour of code reviewers by collecting data to enable analytics"[1]

Our group-specific assignment was to create a Chrome extension that collects data while its users are reviewing pull requests on GitHub and if we had time left, we were also allowed to extend it by implementing a basic way of visualising collected data. The data that is collected has to be sufficient in both quality and quantity so that the user's behaviour can be modelled in order to enable analytics. The end-users of our software product are therefore software engineers, or code reviewers in order to be more specific, that, for our division, use GitHub as their software version control platform. Also, researchers are included in the set of end-users as they, for our division, can use the collected data and the visualisations of the collected data as a basis for their researches.
In relation to this, the main requirements of our extension are *usability*, *maintainability*, and *extensibility*.

- *Usability* is an important aspect as every software developer should be able to use the product effectively, efficiently and according to their satisfaction; if the extension is not easy to use, the extension makes their browser too slow or the user might not benefit from it, why would they use it? In order to model the behaviour of code reviewers more precisely, we will need a lot of software developers using the product; usability thus is a must have.
- *Maintainability* is an important aspect as well, as GitHub might change elements on their pull requests pages and our extension should be able to cope with these changes, without causing the extension to break down.
- *Extensibility* is then, of course, an important aspect too, as whenever GitHub might remove/change/add elements to their pull request pages, the extension has to be extended so that it can track these elements too. Besides, the (optional) basic visualisation that is provided by the extension should also be extensible; future projects on our extension might imply a more in-depth visualisation of the collected data. The provided options by the extension may also be extended so that users can indicate whether they agree with logging of possible new types of data.

In addition to these high-level requirements, the product has to meet the requirements that have been specified, together with the product owner, in the user requirements document (from the Product Vision document).

Section 2 gives an overview of the developed and implemented product. A reflection on the software product and its development process from a software engineering perspective is given in section 3, which is called 'From a Software Engineering Perspective'. Section 4 presents the main functionalities of the product. A documented research on our software product with respect to human-computer interaction can be found in section 5. Section 6 gives an evaluation of the functional modules and the entire product, including a subsection that is designated to a failure analysis. The document is closed by section 7, which gives an outlook regarding possible improvements for the future and the strategy to achieve these improvements.

[1] This is a quote from the first presentation in which we were introduced to the assignment.

2. The Software Product

This section gives an overview of the developed and implemented software product.
The software product consists of two main parts. Section 2.1 will cover the data collection part. Section 2.2 will cover the Visualisation part.

2.1 Data Collection

Data collection has been implemented via a Chrome plugin. The functionality of the plugin can be divided into roughly three parts; the front end, the content page and the background page. These three parts are explained in the following subsections. The backend consists of a content page and a background page, which will be discussed respectively in section 2.1.2 and 2.1.3.

2.1.1 The Front End

Via the menu of the extension, which is a pop-up and the navigation bar in the pages of the extension, the options, the about page and an integrated visualisation (which is a plugin from a group that is responsible for the visualisation part of Octopeer) can be reached. Also, the extension can be turned off and on from here, of which changes are visualised by the icon of the extension that is also included in the pop-up menu.
The options page allows the user to change what data is logged by the extension and what is not. Also, the user can turn the extension on and off from here. The changes that are made by the user, are automatically saved. The personal preferences, with respect to the options, are taken into account by the controllers, which will be discussed in section 2.1.2 and 2.1.3. Additionally, the options page has a button that allows the user to restore the default options of the extension.
The about page informs the user about what Octopeer is, what it does with the collected data, why you should use it and who developed it.
As discussed earlier, the icon indicates whether the extension is turned on or off. But that's not just it, the icon also indicates when a user is reviewing a pull request, based on the active tab, and when an error occurs. If the user is not reviewing a pull request and the extension is turned on (and no error occurs), then the extension is set to standby, which is also indicated by the icon. Therefore the icon gives feedback to the user on what state it is currently in.
The front end of course also has to communicate with the backend. This is done via the Chrome storage, which allows an extension to store data. The backend retrieves data that is stored in the Chrome storage or sets data to it, which can then be used by the front end again. The same holds the other way around.

2.1.2 The Content Page

Via the content controller of the extension, element-event bindings, HTML element selectors and event trackers are created. It also uses a matrix of which element-event bindings and element selectors are allowed to be matched. The content controller itself hooks event handlers to the DOM-tree and is able of sending the full HTML DOM-tree. It also provides the element-event bindings with a database adapter, which sends messages to the background page, so that data can be posted to a real database.
Element-event bindings exist so that specific callbacks (elements) can be bound to an event so that they can be tracked later on. They can be hooked to and removed from the DOM-tree.

HTML element selectors are created so that a specific element selection behaviour can be linked to an element-event binding.
The element-event bindings actually use these selectors in order to keep track of the elements and the event trackers manage the raw data events and log them to the database.
The content page needs to communicate with the background page too. This is done via Chrome runtime messages. These messages are from the main controller and indicate whether to hook and unhook from the DOM.

2.1.3 The Background Page

Via the main controller of the extension, options are taken into account, status changes are passed to the front end, the content controller is activated, the validity of pull request URLs of tabs are checked and messages from the content script are logged to its current database adapter. The main controller therefore is a manager.
Options are taken into account via changes to the attributes in the Chrome storage, that are caused by changes at the front end. An observer has been added to notify the main controller about changes in the options.
Status changes are deduced based on whether the active tab is a tab of a pull request, whether the extension is turned on or off via the options and whether errors have occurred in the backend.
On initialization of the main controller, the content controller is directly activated, so that it is ready to prepare for creating trackers for the elements.
The validity of pull request URLs is checked via a URL handler, which formats and parses the URLs so that this information can easily be retrieved by the main controller for making decisions.
Database adapters are used for interaction with the database. The main controller logs data to it that it has retrieved via the content page and raw data events are also directly logged to it.
As a result, data is collected and logged to a database.

2.2 Visualisation

Section 2.1.1 already quickly mentioned that our Chrome extension has integrated the visualisation Chrome plugin of another group, that is responsible for the visualisation part of Octopeer. Next to this integrated Chrome plugin, the software product consists also of an own visualisation plugin; a SonarQube plugin[2]. This plugin stands apart from the Chrome plugin that has been developed for the data collection part and allows a user to also visualise the collected data from the database to which the data has been logged.
The visualisation of the plugin is for the moment very basic and extensible, since it is a project on its own. When a user is reviewing a pull request, his name is also logged to the database. The visualization part takes this username into account when creating and displaying overviews and graphs concerning the user's behaviour. This way the user gets a personal overview, which is much more useful to the user, because this way they can make changes in behaviour.

[2] Repository link: https://github.com/thervh70/ContextProject_RDD_SonarQube

3. From a Software Engineering Perspective

This section gives a reflection on the product and process from a software engineering perspective. To each of these two items, a section is dedicated.

3.1 The Product

The architecture of the product has been designed based on its functionality, while keeping an eye on the three main goals *usability*, *maintainability*, and *extensibility*. Therefore we've applied quite some design patterns; for example DatabaseAdaptable (Strategy pattern), several DatabaseAdapters (Adapter pattern), several Factories (Abstract Factory pattern) and several Singleton classes (Singleton pattern). Also, our software engineering TA pointed us several times on the possibility of applying a pattern, which helped us because it at least created a discussion. Applying the mentioned design patterns was definitely a good practice, as they helped to create an architecture that is very clean and easy to understand. As a side note, applying design patterns of course doesn't directly mean that the product has been well-designed, but applying them where it is beneficial for the design, of course, is.

Besides, the software product has been checked several times with a tool provided by the Software Improvement Group. The tool was provided from week 4.6 and we've used it weekly up till the last week. The tool provided us with some useful feedback, which has been resolved and therefore the product made progress based on the tool. Resolving just the 'useful' feedback (according to us and our TA) was a good practice, as it resulted in for example smaller functions.

The software product itself has been programmed mainly using the language 'TypeScript', which is transpiled to JavaScript. The reason we chose for this, is that the extension should run in Chrome, which requires JavaScript. TypeScript is a relatively new language and therefore the basics are there and work, but it lacks some advanced features, as we've experienced. For example, it doesn't have a HashMap that can easily be created using an already existing utility class (like in Java). However, TypeScript does have its advantages over JavaScript, for example static type inference, class and module support and syntax that is similar to the one of Java. Because of these advantages we've chosen to use TypeScript to develop our software product. We think that using TypeScript was a good decision as we've gained more experience using a language that can become interesting in the future, once it has been further developed. Besides, the quality of the product didn't suffer from this decision, as we were always able to come up with a smart way of implementing what we wanted in an efficient way.

Also the product has been tested well and a static analysis tool (which will be reflected on in section 3.2) has been applied in order to guard the quality of the code (as there were no more static analysis tools to be used, just one; TypeScript is relatively a new language, as we've already mentioned). The line coverage of the product is over 90% and while testing, several bugs have been exposed (and later on fixed). We already knew that testing is a good practice, but this once again proves it.

As a conclusion, we've learned a lot from a software engineering perspective in terms of experience; most of the good practices are things that we've already seen in the Software Engineering Methods course and applied during the practical of that course, but this project allowed us to apply these software engineering practices on a language that's quite new and totally different from software that we've built so far (building a data collector and visualisation tool). In a future project we might want to consider TypeScript again, if applicable, to program our software product. Besides, we would, of course, apply the good practices again wherever possible and use the experience that we've gained. Therefore we've learned a lot on applying (the mentioned) good practices from a software engineering perspective to a relatively larger scale software product, which is something valuable for the future.

When reflecting on the software product, from a software engineering perspective, it can be concluded that the software product has thus been well designed and well implemented.

3.2 The Process

At the start of the project we directly started on creating user requirements, which were documented and approved by our context TA. Because of this, both the team and the context TA had during the project something to look back at and base our progress on. This was definitely a good practice, as the document was a clear basis from where we built the software product.
Besides, during the project we applied the scrum methodology for the software development process. By doing this, we started on creating a plan for each next week and at the end of such a week, we would reflect on our progress and have a meeting with our TAs on it. This was a good practice too as it gave the chance to finish off our tasks completely, at the same time inform the others about it and discuss the progress on the project together with the TAs for feedback.
With respect to Git behaviour, we've improved ourselves during the project too. During the project, we started to create commits that were relatively smaller than before. We noticed this when reviewing the (amount of commits / amount of lines) ratio on GitHub. By doing so, we experienced that it makes the development process easier, as you can easily decide to, for example, cherry-pick a small commit or, if needed, revert a small change without having to cherry pick or revert the rest of the changes. It makes changes more maintainable and that's definitely a good thing.
During the software development process, we also used automated test building, via Travis CI, and automated coverage checks, via Coveralls. From the Software Engineering course, we already knew that automated test building was a good practice, but we hadn't applied automated coverage checks yet. It turned out that this was a good practice too, as for every pull request that is open, Coveralls adds an extra check to it, indicating whether the coverage has increased or decreased; this made us relatively more aware of our changes with respect to testing.
We also applied Waffle.io for managing our issues and keeping a nice overview, which is a good practice that we already knew, but this once again proves it.
For communication, we used Slack, in which we integrated a GitHub bot, a Travis CI bot and a Coveralls bot to post updates to their designated channel. This was a good practice too as the updates made us relatively quickly aware of changes by other team members and if a build would fail, it could always quickly be fixed.
As a conclusion, we've learned a lot from a software engineering perspective in terms of experience and new good practices; again, most of the good practices are things we've already seen and applied in the Software Engineering course. But we also have learned about new good practices, Coveralls for example, using smaller commits or having meetings frequently for discussing issues in a large project.
Therefore we've learned a lot on applying (the mentioned) good practices from a software engineering perspective to a relatively larger scale project, which is also something valuable for the future. When reflecting on the process, from a software engineering perspective, it can be concluded that the process has thus been well organised.

4. Developed Functionalities

This section elaborates on the functionalities of the software product by giving a description of the developed functionalities. Most of the functionalities have already been described in section 2, so this section will only generally describe Octopeer's functionalities.

The Chrome extension is able to log data, from specific semantic data and specific raw data to even the full HTML DOM-tree. Semantic data consists of semantic elements on the screen (a GitHub pull request page), for example a click on the 'merge' button, and raw data consists of positions in terms of coordinates on the screen, for example, the position of the mouse. The data can be posted to any database so that it can be retrieved for any purpose (visualisation for example).
The logging of data can be personalised via the options; the logging of provided specific semantic elements / raw data can be switched on and off via the options menu, based on the preferences of the user.
When the extension is logging data, it also gives feedback to the user by indicating its status changes via the icon in the front end.
Besides, the collected data can be visualised via an integrated visualisation extension of the group that is responsible for Octopeer's visualisation part. It is possible to go to this extension via our extension.
Also, we've implemented our own way of visualising data in a basic way via a SonarQube plugin, which is extensible.

Software engineers can use this tool so that peer review data on pull request is sent to a database. This data can be visualised and from this software engineers can deduce their own conclusions. For the moment no direct feedback is yet given, but the software product is made really extensible so that this is allowed in the future.
Therefore, also based on the user requirements with respect to the description above, the user's needs are satisfied to large extent, as the 'must haves' and the 'should haves' (from the Product Planning document) have all been implemented.

5. Human-Computer Interaction [ID]

This section describes the Human-Computer Interaction aspect of the software product. It documents an HCI research (usability study) in terms of three parts; the applied method, the results and the conclusion. To each of these three parts, a subsection is designated.

5.1 The Method

As there won't be much interaction in our data collecting extension, we've zoomed in on two specific things that do require user interaction: toggling the main option and saving. The main option is a switch that enables/disables the services of Octopeer.
For these two specific things we've implemented different ways of interacting with them (3 ways for interacting with the toggle and 2 ways for interacting with saving) and made it easy to quickly change these different interaction options during (possible) user tests.
The goal of the method that should be applied to these tests is then, of course, to find out what interaction possibility would be most appreciated by the user (based on ease, clearness, etc.). At the same time, we wanted to get feedback from the users about the whole extension, as it's not very time consuming to do a walkthrough and it would give us useful feedback in general on interacting with the extension.
We, therefore, decided to let the users perform a walkthrough based on a checklist containing a sequence of simple tasks to perform and observe their behaviour. This walkthrough is performed while using the empirical 'Thinking Aloud' method. By doing so, we will get feedback on which tasks users can perform and get an idea of how much time this takes. At the end of each walkthrough we would perform a quick debriefing using a few open questions ('So what do you think?' for example). Most of the feedback has been retrieved by applying the 'Thinking Aloud' method, but we also liked to give the users the opportunity to give any conclusions on their thoughts. Just a few general remarks, on what the users liked and what could be improved about the extension, would be sufficient. The entire walkthrough, including the quick debriefing, would take around six minutes and notes would be made on a computer while managing the experiment.
Before applying this method, we decided to do a small literature research on how many users would be sufficient for applying mainly the Thinking Aloud method. We found an interesting paper [1] on this subject in which the author recommended using 4 ± 1 subjects in a thinking aloud study (page 393). We finally decided to perform a user test on 10 subjects, of which ⅓ of the subjects would be unbiased (with respect to being involved in the assignment of the project) and the others would be biased. These 10 subjects are all members of groups that are related to the Tools for Software Engineering context. This relatively a larger number of subjects than the paper suggests, but the walkthrough wouldn't consume too much time and we wanted to have as much feedback and opinions as possible on the extension, on a reasonable scale. As time permitted it, we decided to put this plan into action.

5.2 The Results

In general, the remarks that were made during the walkthrough were quite positive and the behaviour that the users showed consisted of 'good signs'; most tasks were completed easily.
The tasks that were found to be easy were related to finding your way through the extension. The users quickly navigated their way through the extension, sometimes with some mumbling ("Oh wait a second.. *this* should probably trigger it.''), but most of the time based on a sequence of signs that indicated that the users knew what they were doing (while seeing the extension for the first time).

The tasks that were found to be relatively hard were related to switching specific options on and off, based on a short description of what they had to turn on and off. Users indicated via their behaviour and via the things they said while performing the tasks that quite some options were found to have an overlap. According to the quick debriefing, we had at the end, users experienced that they didn't really know what option did belong to what description of a task, as several options could belong to the description. The descriptions of the tasks related to the options weren't vague, but what option to choose for what purpose was.
The specific things (toggling the main option and saving) that we wanted to test with respect to user interaction, were all presented one by one during the walkthrough. In general, the users clearly seemed to have a preference for one of the options that we provided for different interaction possibilities. Most users clearly indicated that they preferred the interaction possibility for disabling the rest of the options (recoloring them to gray and making them unclickable) when clicking on the main option over the others, as it was most clear what happened and still provided the opportunity to see what options are there (instead of throwing them out of the page). Most of them also preferred automatic saving, as it's really easy and you don't have to press a submit button each time you would like to make a change in order to indicate your preferences.

5.3 The Conclusion

There were quite some surprising results, in our opinion, derived from the user test. We were happy to see that navigating your way through the extension as a user, happens smoothly and that when seeing the extension for the first time, it speaks more or less for itself on what each part of the front end stands for.
The surprising results included that users really seemed to have a clear preference for one of the provided interaction options for both toggling the main option and saving the options.
Something we didn't expect was that quite some options of the set of options would be seen as overlapping with other options. When reflecting on this and having had the debriefings with the user, we did also come to the conclusion that the options should be revised in order to make them more clear and merge some of them together.
Summing up all the conclusions, we think that they are representative for this usability research, as the users explained their way of reasoning via both the Thinking Aloud method and the quick debriefing, which really helped us on getting some insight on what a user would actually think about our front end.
As a recommendation based on this usability study, it is advisable to have another look at the options, to keep the interaction design option for toggling the main option in which the other options become gray and unclickable and to keep the interaction design option for automatic saving the options.
These recommendations have been taken into account and implemented in the extension.
As a final remark, we're happy with the feedback, as it really helped us on improving the usability of the software product.

6. Evaluation

This section contains an evaluation of the functional modules and the entire product. Also a subsection is designated to a failure analysis of the product.

6.1 An Evaluation of the Functional Modules and the Product

Coming back to the end-user's requirements that have been discussed in the introduc(k)tion, these are useful factors to evaluate the functional modules and the entire product. Not only all 'must haves' and all 'should haves' have been implemented, based on the requirements document, but also *usability, maintainability*, and *extensibility* have been taken into account during the whole project. We've already mentioned each of these three main important high-level requirements several times up till this section; usability has been extensively discussed in the Human-Computer Interaction section (the usability research, section 5), and all three of them have been extensively discussed in sections 2 and 3. The result of keeping the extension extensible will also be elaborated on in the final section (Outlook, section 7).
That we dedicate special attention to these requirements, which are the most important ones, in our report, reflects on the resulting software product and its functional modules.
As a final remark, we're happy with the way we've implemented the product, taking into account the requirements, and think that we satisfy the user's needs with our software product.

6.2 A Failure Analysis

Based on the conclusion of subsection 6.1, the reader is allowed to conclude that we've met the requirements and that there is little to discuss as a failure analysis.
There are no major failures in the software product, so there is little to analyse.

7. Outlook

This section finalises the final report by discussing possible future improvements and a possible strategy on how to achieve these improvements.

When discussing the future of our software product, we want to emphasize again the extensibility of our software product.
The software product can be extended in a lot of ways; for example, the Chrome extension could be extended by collecting more (semantic) element data if GitHub changes its pull request pages, and the SonarQube plugin could be extended by adding more advanced ways of visualising the collected data.
Another important way of extending our software product is to actually generate feedback for the user on their peer review behaviour with respect to pull requests, as it would make it even more attractive to use the software product for a software developer. For example, if the user has a pull request to review that contains about 500 lines of changes in the code and reviews this pull request in under five minutes, and repeats this way of reviewing pull requests (with a similar ratio), feedback can be generated that notifies the user that he might want to consider taking more time for reviewing pull requests. Besides, it could make peer reviewing more effective, as you can learn from the extension how to improve.
These future improvements can be applied via various strategies; they can for example be processed in Bachelor End Projects, by researchers or by any other enthusiastic software developer. They would do that so that they can contribute to a product which can actually be used frequently by software engineers.

The software product itself can already be used for research purposes, so why not extend it?

Bibliography

[1]. Nielsen, J. Estimating the number of subjects needed for a thinking aloud test (30 april 2002). In conclusions. para. 3. [Online] Available from: http://ac.els-cdn.com/S1071581984710652/1-s2.0-S1071581984710652-main.pdf?_tid=0f7368dc-32e6-11e6-82a9-00000aab0f6b&acdnat=1465987616_dc70ca9434fbf5be2e654875cd142628 [Accessed during May and June 2016]

Appendix A: Sprint Plans and Retrospectives

This appendix refers to all sprint plans and sprint retrospectives that have been created during the process.

Week	Sprint Plan	Sprint Retrospective
1	[pdf]	[pdf]
2	[pdf]	[pdf]
3	[pdf]	[pdf]
4	[pdf]	[pdf]
5	[pdf]	[pdf]
6	[pdf]	[pdf]
7	[pdf]	[pdf]
8	[pdf]	[pdf]