Functions - When Developers Use Them and Why

By

Alexey Braver

A thesis submitted to the Hebrew University of Jerusalem
as a partial fulfillment of the requirements
of the degree of Master of Science
in the Faculty of Computer Science and Engineering

Under the supervision of **Prof. Dror Feitelson**

February 2022

פונקציות ־ מתי
מפתחים משתמשים בהן ולמה

ע"י

אלכסיי בראבר

האוניברסיטה העברית בירושלים
בית הספר להנדסה ולמדעי המחשב על שם רחל וסלים בנין
החוג למדעי המחשב

עבודת גמר לתואר מוסמך במדעי המחשב

עבודה זו הונחתה על ידי
פרופ' דרור פייטלסון

אדר א' תשפ"ב

Abstract

Software design has a huge impact on the product quality, performance, security, maintenance, costs and more. There are many attitudes about what the best design is, and many discussions about it, yet we can say with caution that there is a consensus about most of the cases, especially in the high-level design. However, the design practices have barely been experimented on in quantitative research.

In this research we have done such an experiment and tried to quantify the considerations behind different designs, especially in regard to function extraction. We tried to answer some difficult questions about functions length, such as - is there an ideal function length? If so, what is it? Is there such a term as "too short function" or "too long function"? What are the considerations for forming a function? Is there something in common among developers in the way they extract functions?

Our goal was to investigate the function extraction process in a real code used in the industry and among real developers. We have conducted an experiment that included 23 experimental subjects who were given a code refactoring task and some design questions about it. The task contained real code from the industry, that had been converted to one big function, and the experimenters were asked to refactor it by extracting functions.

To analyse the results we used some interesting methods such as: looking at the areas in the code where functions were extracted, creating histograms of the number of functions that were extracted and their length, calculating the part of code that was in functions, and more. After analysing the results from several angles, we saw that most of the experimenters (80%) extracted small functions (5-30 lines of code) in the code task, and the most common trigger for function extraction was the scope that was created by control structures such as "if" and "try". According to their answers on the experiment questions, we saw that the main consideration for function extraction was the logic cohesiveness, and most of them think that there is no an ideal function length. Moreover, they also answered that "making function as short as possible" and "making each function close to the ideal length" were the two least important considerations for function extraction. However, most of those who do think that there is an ideal function length answered that it should be 20-30 lines of code.

Our main conclusion from this research is that developers extract small functions (5-30 lines of code), although they believe there are much more important considerations for function extraction than length. We also provided a new approach to investigate development patterns among live developers, and showed new methods to analyze and compare code results.

תקציר

למבנה התוכנה יש השפעה עצומה על איכותו של מוצר, ביצועיו, תחזוקתו, עלותו ועוד. ישנן עמדות רבות לגבי מהו המבנה הטוב ביותר, ויש הרבה דיונים על כך, ועדיין אנחנו יכולים להגיד בזהירות שיש קונצנזוס לגבי רוב הדברים, במיוחד ברמת המאקרו. עם זאת, מבני התוכנה השונים כמעט ולא נבדקו בניסויים כמותיים.

בניסוי זה ביצענו ניסוי כזה, וניסינו לכמת את השיקולים מאחורי מבני תוכנה שונים, בייחוד בכל מה שקשור ליצירה של פונקציות. ניסינו לענות על מספר שאלות קשות לגבי אורך של פונקציות, כמו – האם יש אורך פונקציה אידיאלי? אם כן, מהו? האם יש מונח כזה "פונקציה קצרה מדי" או "פונקציה ארוכה מדי"? מה השיקולים ליצירת פונקציה? האם יש משהו משותף בקרב מפתחים באופן שבו הם יוצרים פונקציות?

המטרה שלנו הייתה לחקור את תהליך היצירה של פונקציות בקוד אמיתי, שנמצא בשימוש בתעשייה ובקרב מפתחים אמיתיים. ביצענו ניסוי שכלל 23 נסיינים שקיבלו משימת קוד ומספר שאלות לגביה. המשימה כללה קוד אמיתי מהתעשייה ששונה לפונקציה אחת גדולה, והנסיינים היו צריכים לשנות את מבנה הקוד ע"י יצירה של פונקציות.

כדי לבדוק את התוצאות השתמשנו במספר שיטות מעניינות כמו: להסתכל על האזורים בקוד בהם נוצרו פונקציות, יצירה של היסטוגרמות של מספר הפונקציות שנוצרו והאורך שלהן, וחישוב של חלק הקוד שחולץ לפונקציות. אחרי שבדקנו את התוצאות מזוויות שונות, ראינו שרוב הנסיינים (80%) יצרו פונקציות קטנות (5-30 שורות קוד) במשימת הקוד והטריגר הנפוץ ביותר ליצירה של פונקציה היה סקופ הקוד שנוצר ע"י מבני שליטה כמו "IF" ו- "TRY". לפי התשובות שהם ענו על שאלות הניסוי, ראינו שהשיקול המרכזי ליצירה של פונקציות היה הלכידות הלוגית, ורובם חשבו שאין אורך אידיאלי לפונקציה. יתרה מזאת, הם ענו גם שהשיקול "לעשות את הפונקציה קצרה ככל שניתן" היה השיקול הכי פחות חשוב ליצירת פונקציות. עם זאת, מכל מי שכן חשב שיש אורך אידיאלי לפונקציה, ענה שהאורך צריך להיות 20-30 שורות קוד.

המסקנה המרכזית שלנו מהניסוי היא שמפתחים יוצרים פונקציות קטנות (5-30 שורות קוד), למרות שהם מאמינים שיש שיקולים יותר חשובים ליצירת פונקציות מאשר האורך שלהן. במחקר הזה סיפקנו גם גישה חדשה לחקור תבניות פיתוח קוד בקרב מפתחים, והראנו מספר שיטות לניתוח והשוואה של התוצאות.

Acknowledgements

I would like to thank wholeheartedly my supervisor Prof. Dror Feitelson. I first met Dror as part of my BSc. studies in a software engineering course he taught. After starting my MSc. studies I enrolled to his software engineering seminar where I was exposed to many studies in the software engineering field. It was my first step in the research field, and after I have finished the seminar I decided to ask Dror to take me as his MSc. student. We searched a lot for a study where we can innovate and bring something new to the software engineering field. Happily, in the end we succeeded. Dror taught me a lot about research in general and research in software engineering in particular. He gave me lots of independence and a lot of advice that helped me get to this wonderful result.

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 1

Introduction

Nowadays, almost every field in our life includes computers and software. Many tasks that were once performed by humans are now performed by computers, and the number of these tasks keeps growing. Often, the quality of an organization or a field is determined by its technology, for example medicine is considered better if it has better technology, the army is considered stronger if it has better technology, and countries are considered a better place to live in if they have better technology.

One of the main components in technology is the software, thus it should be treated with the appropriate respect. Many resources are being invested to develop new algorithms, new functionalities, and new developments. Obviously, all of these include software, but in most of the cases the emphasis is on the functionality of the software and not on its design. The software is the hidden part of the product so it's not being examined by the customers. Moreover, in most of the cases the low-level design has no big impact on the software's performance. As a result, it's very easy to compromise on quality in this part and just make it as simple as possible. Because of that, there is also not much research in this field.

However, if we look on this from a different perspective, for example the development and maintenance of the software, we will see that the low-level design has a significant impact of the final product. A good design can save a lot of effort for the developers, and as a result decrease the development time of the product, which of course will reduce the costs. As we said before, low-level design has a very small impact on the software performance, and of course it does not change the software's functionality. This makes the software design hard to quantify and study in terms of comparing between different designs. In this experiment we will try to overcome this gap.

1.1 Software Design

Software design includes the definition of architecture, functions, objects, and the overall structure of the software. There are many different ways of designing software, almost all of which involve coming up with an initial design and refining it as necessary. Software design also includes the deployment on hardware, use of databases, and third-party frameworks of the software. This is the big picture of what is running where and how all the parts will interact. Software design has a big impact on the software quality, performance, stability, security, costs, maintenance efforts and more. For example, If the software design includes many redundant data stores, it will slow down the whole software, use more resources, and will lead to higher costs and lower quality. It's often common to divide software design into two main parts:

High-Level Design The overall design of the product in the macro level. Includes the description of the architecture, platform, modules, services, database and the relationships, connections, and integrations among all of them. It's the conversion between the business requirements into high-level solution. It is based on considerations like modularity and information hiding [Par72]. Mostly, it will be created before the low-level design by the software architect.

Low-Level Design The detailed design of the product in the micro level, based on refining the high-level design. Includes the logic of all the components, the algorithms, the design patterns and their actual implementation. Mostly, it will be created after the high-level design by the developers. Low-level design also includes part of the functions design based on considerations like that a function should only do one thing.

1.2 Low-Level Design Implementation

As we explained above [1.1], the low level design is the detailed design of the product in the micro level. The level of the details can differ between designs, but anyway during the implementation of the low-level design, actual code is being written. The level of the details can affect the freedom the implementer has to choose the names of the classes and variables, the data structures and the algorithms he is going to use, and the functions he is going to extract. However, although he may have to implement the given functions according to the low-level design, it doesn't mean that he can't extract additional functions. There are several reasons for function extraction as described below in Section 1.4, but what matters now is that it can be done in this phase. Of course these additional functions must not affect the functionality given by the low-level design, but they do provide a lot of freedom to the implementer to extract functions and use design patterns.

Due to this freedom, in this research we couldn't conduct a coding experiment just by giving the low level design of the task to the experimenters, and ask them to implement it, because it can be very difficult to compare the results. Because of that, we conducted a coding experiment in which we chose to provide a base-code to the experimenters, and asked them to refactor it.

1.3 Software Evolution and Refactoring

Product requirements always change over time, causing the software to evolve [Leh96]. New technologies are always developed, new requirements always arrive, improvement should be always considered. All of these may also require adaptions in the software design. The actions to change software design without changing its functionality are named software refactoring [FB18].

Code refactoring is defined as the process of restructuring and improving the internal structure of existing code without changing or adding to its external behavior and functionality. Generally, refactoring is done by applying a series of basic actions, sometimes known as micro-refactorings, which preserve the software's behavior and functionality. Among other things, these basic actions include extraction of variables, fields, and functions. If the refactoring was good enough, these changes will improve software readability, performance, and make it much easier to maintain. Many will say that the goal of code refactoring is to turn dirty code into clean code, which increases the project's readability. In this research we are going to investigate the function extraction part of software refactoring.

1.4 Reasons for Function Extraction

When we are refactoring, it may look odd to extract functions which are only called once. There is often an assumption that the main reason for extracting a function is so that it can be called from multiple places in the code base and prevent code repetitions. This is indeed a good reason for extracting functions, but it's not the only one. There are many additional reasons for extracting functions. Let's discuss them.

1.4.1 Readability

Extracting a function and naming it with an informative name certainly makes the code more understandable. It easily can replace the comments which are written by developers to explain the code they wrote.

1.4.2 Reuse

As we introduced before, this is probably the most obvious reason for extracting functions. When we understand that we wrote code that can be used in the future in more places, we should extract it to a function and use it again. But it does not end here. We also need to place this extracted function in an appropriate place where everyone can consume it easily. Many times, developers look for a function just by typing the name of the object and clicking on the dot key to see what functions the object has. If we don't place functions in the appropriate path, others may not find it and won't use it. As a result, duplicate code may appear or inline functions that developers wrote from scratch because they didn't find the function that has already been written.

1.4.3 Better Testing

When we are writing tests, functions have quite a big impact on our tests. If the functions are very big and have many responsibilities, it will be much harder to test them and as a result it will be hard to test the whole program. Moreover, sometimes such functions will even make the tests framework impossible to configure, and will require to add additional functions to compensate for the bad ones, because there is always a deeply nested code inside it that requires a lot of loops and conditions to reach and test it. It may even make developers skip tests or change production code because of these difficulties. There is even a methodology (TDD - Test Driven Development) that requires to write the tests before the functions themselves because often, in writing tests, we discover defects or missed opportunities for improving our code. TDD relies on a cycle of declaring intent in a test, writing code that works, scanning that code for more insights into how the code should be structured and improved, making improvements, and making the changes part of the code base. When we follow this cycle, functions can't help but be testable and small. They tend to also become well-named and readable due to the repeated scanning and opportunity for revision.

1.4.4 Code Explanation

Many times, we are facing complex conditionals such as:

"if ((x and not y) and not z) or ((x and z) and (q>7))"

However, we can make it more understandable just by extracting it to a function and name it with an informative name. For example:

"If (isValidInput())"

It's much clearer now what the condition is for. In the same way we can extract to functions small blocks of code, according to their logic, and name them with informative names that explain their logic. Comments become redundant after that. In addition, this shortens and simplifies the code, and makes it more testable and understandable. Since a large component of our work is reading and understanding existing code, this can save every team member minutes or hours every time the code must be modified in the future.

1.4.5 Understand Big Functions

There is no doubt that long and deeply-nested functions are hard to read. Many background operations such as converting, copying, filtering, error handling, etc. make it difficult to understand the real purpose of the function. In order to clearly see and understand the structure of such a function, we need to move details of sub-operations out of our way. Sometimes methods contain a number of variables that exist only to support the block of code we intend to move. When we extract the method and its variables, the original code becomes smaller and more obvious. Extracting a method gets it out of our way so that we understand the function better and manage it with a lower chance of accidentally injecting defects.

1.4.6 Expose API

When we extract functions and define them public, we expose their usage outside the class. Obviously, the main purpose of that is to allow us to use these functions outside the class, and it's a good reason on its own to extract functions – when we want to expose code outside the class. However, it also helps us to understand the functionality of a certain class and make the code more understandable. Code documentation and code completion are closely related to code reuse. Many developers rely upon their IDE to understand code. They use the tools in their IDEs to find where functions are called and where variables are used. They also type the name of an object and press the period key to see the list of functions that exist and are callable for that object. Even when functions have light usage, they appear in the list. This gives developers an understanding of what the class is for, how it is to be used, and what it means in its context.

1.5 Dirty Code

Dirty code is an informal term that refers to any code that is hard to maintain and update, and even more difficult to understand and translate. Dirty code is typically the result of deadlines that occur during development. This is the idea behind technical debt: if code is as clean as possible, it is much easier to change and improve in later iterations – so that your future self and other future programmers who work with the code can appreciate its organization. When dirty code isn't cleaned up, it can snowball, slowing down future improvements because developers will have to spend extra time understanding and tracking the code before they can change it. Some types of dirty code include:

- Monolith architecture, that is long blocks of code with multiple responsibilities and many functionalities that are hard to understand.

- Codes, methods, or classes that are so large that they are too unwieldy to manipulate easily.

- The incomplete or incorrect application of object-oriented programming principles.

- Superfluous coupling.

- Areas in code that require repeated code changes in multiple areas in order for the desired changes to work appropriately.

- Any code that is unnecessary and removing it won't be detrimental to the overall functionality.

1.6 Clean Code

Generally, clean code is code which is easy to read, understand, and maintain, thereby easing future software development and increasing the likelihood of a quality product in shorter time. But of course there is not only one correct definition. According to Martin [C M08, p. 7], there are probably as many definitions to "Clean Code" as there are programmers. So he asked some very well-known and deeply experienced programmers what they thought.

Bjarne Stroustrup,

inventor of C++ and author of The C++ Programming Language

«I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.»

Grady Booch,

author of Object Oriented Analysis and Design with Applications

«Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.»

"Big" Dave Thomas,

founder of OTI, godfather of the Eclipse strategy

«Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.»

Michael Feathers,

author of Working Effectively with Legacy Code

«I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code's author, and if you try to imagine improvements, you're led back to where you are, sitting in appreciation of the code someone left for you—code left by someone who cares deeply about the craft.»

Ron Jeffries,

author of Extreme Programming Installed and Extreme Programming Adventures in C#

«In recent years I begin, and nearly end, with Beck's rules of simple code. In priority order, simple code:

- Runs all the tests.
- Contains no duplication.
- Expresses all the design ideas that are in the system.
- Minimizes the number of entities such as classes, methods, functions, and the like.

Of these, I focus mostly on duplication. When the same thing is done over and over, it's a sign that there is an idea in our mind that is not well represented in the code. I try to figure out what it is. Then I try to express that idea more clearly.

Expressiveness to me includes meaningful names, and I am likely to change the names of things several times before I settle in. With modern coding tools such as Eclipse, renaming is quite inexpensive, so it doesn't trouble me to change. Expressiveness goes beyond names, however. I also look at whether an object or method is doing more than one thing. If it's an object, it probably needs to be broken into two or more objects. If it's a method, I will always use the Extract Method refactoring on it, resulting in one method that says more clearly what it does, and some submethods saying how it is done.

Duplication and expressiveness take me a very long way into what I consider clean code, and improving dirty code with just these two things in mind can make a huge difference. There is, however, one other thing that I'm aware of doing, which is a bit harder to explain.

After years of doing this work, it seems to me that all programs are made up of very similar elements. One example is "find things in a collection." Whether we have a database of employee records, or a hash map of keys and values, or an array of items of some kind, we often find ourselves wanting a particular item from that collection. When I find that happening, I will often wrap the particular implementation in a more abstract method or class. That gives me a couple of interestin advantages.

I can implement the functionality now with something simple, say a hash map, but since now all the references to that search are covered by my little abstraction, I can change the implementation any time I want. I can go forward quickly while preserving my ability to change later.

In addition, the collection abstraction often calls my attention to what's "really" going on, and keeps me from running down the path of implementing arbitrary collection behavior when all I really need is a few fairly simple ways of finding what I want.

Reduced duplication, high expressiveness, and early building of simple abstractions. That's what makes clean code for me.»

Ward Cunningham,

inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code.

«You know you are working on clean code when each routine you read turns out to be

> pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.»

However, if we want to interpreter it into actual code operations, it might be quite difficult. Since in this research we focus on function extraction, we will try to define "Clean Code" in terms of function extraction. Martin in his book [C M08, p. 34] suggests that functions should be very small(less than 20 lines of code), and we will try to check if there is agreement on that among professional developers.

Chapter 2

Background and Related Work

In the literature, there are many attitudes and papers about software design and refactoring. However, to our best knowledge, there are not so many experiments that compare software designs, in particular software's functions designs. Intuitively, we can deduce that longer code is more complex to understand, because there are more lines to read and remember, but does that mean that if we will refactor this code and move some of it into functions, it will be more understandable? According to Landman, Serebrenik, Bouwers and Vinju [Lan+15] the connection between code complexity to number of lines of code is not straightforward. However, there are also some studies that claim that bigger code procedures/modules with large size [D B+93], high complexity [KF91], and low cohesion [MB07] require significantly more time and effort for comprehension, debugging, testing and maintenance. From that we can deduce that if we reduce code size and complexity, but increase code cohesion, the code will be more understandable and easier for comprehension, debugging, testing, and maintenance.

Indeed, there are many studies that claim that long methods are a "bad smell" and try to identify when methods are too long and need refactoring, by using software metrics such as size, cohesion, complexity and coupling. Some of these studies even try to find the best metric for this task. For example, Yoshida, Kinoshita and Iida [YKI12] try to use only a cohesiveness metric to divide the code into functional segments. Another study [CAA15] claims that size and cohesion should be used together to get better results. There is even a newer study [Cha+18] that claims that all these metrics should be used together to get the best results. There are also studies that propose different approaches to automatically extract methods such as block based slicing [Mar01] or relying on complete computation of a given variable and the statements affecting the state of a given object [TC11]. All these approaches are about finding the best and the most accurate automatic method to identify function extraction, but in our study, we are trying to find the most common reasons human developers extract functions and if they believe that there is an ideal length of functions.

2.1 Clean Code - In Terms of Function Extraction

Martin explains in detail in his book [C M08] the best practices to write code. Among other things, he talks about clean code, extracting functions, error handling and more. In the functions chapter he explains why functions should be as small as possible. He also claims that the block of code within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call because it also adds documentary value because the function called within the block can have a nicely descriptive name. Concerning function length, Martin writes:

« The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. This is not an assertion that I can justify. I can't provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I've written several nasty 3,000-line abominations. I've written scads of functions in the 100 to 300 line range. And I've written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small. In the eighties we used to say that a function should be no bigger than a screen-full. Of course we said that at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. Nowadays with a cranked-down font and a nice big monitor, you can fit 150 characters on a line and a 100 lines or more on a screen. Lines should not be 150 characters long. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long. »[C M08, p. 34]

With this approach he emphasizes a significant reason for function extraction – explanation and documentation of the code. Size, complexity, coupling, and even cohesive metrics will not identify such code to be extracted to a function. That's why experiments among human developers are important as well, and not only among automatic tools. In this research we will try to check if these statements are applied among nowadays developers.

Chapter 3

Research Questions

Is there an ideal length of functions? If yes, what is it? This question is interesting, but complicated. As it was introduced in the introduction, if we compare functions with the same functionality, but with different length, the impact on the performance of the software in such a low level change is minimal and almost non-existent in human terms. The results of running the functions obviously will be the same since the functions are with the same functionality. Thus, we have no empirical metric to measure the difference between the functions in terms of quality.

So, how can we decide which length is better? We can simplify the question of ideal length by trying to answer it with more vague answers like "long functions" and "short functions". From this we can ask more questions like: Which functions are better? Longer or shorter? Is there such a term "too long functions"? "Too short functions"? What are the considerations for extracting functions? What is the most significant reason for function extraction? Would different developers divide into functions in the same way?

In this research we will try to answer these questions. We will ask developers what they think the answers for these questions are, and check if their answers are correlated with their actual coding habits. In addition, we will compare the results to the related work we have introduced.

Chapter 4

Methods

To answer our research questions, we decided to conduct an anonymous experiment among developers which includes some questions and a coding task. We had to do a lot of preparations before we did the experiment itself. We knew that we have only "one shot" since recruiting experimenters is quite a complicated task, and if we had done the experiment without thinking on all what we wanted to achieve, we could have lost all our experimenters. In addition, we had to think of ways we are going to analyse the results, and it wasn't easy since we had dozens of results and each result contains hundreds of code lines.

4.1 Code for Refactoring Experiment

Our goal was to see how different developers arrange their code and if there is any dependency between their design and their background and profile. It's a non trivial experiment and is quite hard to implement, because if we just give them a programming assignment that includes enough logic to get different designs, it's a lot of work, and they can come up with very different designs that are hard to compare. That's why we decided to start from a given code-base and just ask them to refactor. At first, we wanted to do the experiment by looking at the way the experimenters partitioning a given code to modules. The given code was supposed to be a designed and refactored code which was converted, manually by us, to a monolith. The experimenters needed to redesign and refactor it as they think best. We can then compare their designs to each other, and to the original design.

Since Java is an object-oriented language, we considered it as a good fit for this task, because a small project with a few classes can be converted to one class monolith, and the experiment will be a refactoring of this monolith. We started to search for a java project we can do the experiment on. We see a high value in a code which was being used in the industry, therefore we looked for a "real" code in open-source projects in GitHub. Our goal was to find code which is long enough to allow different results among the experimenters and is short enough to complete the experiment in 20-30 minutes. We couldn't find such code. Some of the reasons were the fact that the code was too big, and the classes were too interdependent. We couldn't find a break-point where we can cut the chain of the dependencies to make the code self contained. Eventually we decided to use an academic project. We found a project of an ATM that included 5 classes:

1. ATM
2. Account
3. Bank
4. Transaction
5. User

In this kind of project our challenge was to find a way to convert it to a monolith with as few as possible changes, to keep it as similar as possible to the original project. At first, we needed to remove the classes. The challenge was to find a way to represent the fields, the methods and the data that was stored in the classes. We decided to do that by creating a database which was represented by an array of arrays, and the deepest array represented the object with its fields. Each field had its unique index in the array, and the index was defined with a constant to make it easier to understand the order of the fields in the array.

After we had finished creating the monolith, we sent it to 4 experimenters in order to understand if it satisfies our experiment requirements. We mostly checked the variety of the results, and the time it took to finish the experiment. The conclusion was that from the time perspective it does satisfies our experiment requirements, but from a variety of other perspectives it doesn't. Although there was some variety between the results, it was not significant. All the experimenters created 4-5 classes with almost the same responsibility and almost the same names.

After that pilot, we decided to leave the idea of making a monolith from classes and try to make it from functions. The concept was almost the same - take some functions and convert them to one big function, kind of a monolith. In this way, we can significantly increase the variety of the results by increasing the number of functions, and not be worried about the time it will take, because the functions are potentially much smaller than classes. We called the procedure of converting the code from many functions to one function "flattening".

Since Python is a more procedural language than Java, we decided to search for a suitable code in Python. After we found such a code, we did the same pilot and sent again the experiment to 4 developers. The results we received were much better both from time consuming and from variety perspectives. The code we chose is from file "BaseAdapter.py" in "Requests" http python library [21]. Requests allows to send HTTP/1.1 requests easily.

4.2 Code Flattening

The code was flattened manually in the following way:

1. One function was chosen to be the main function("send"): This function was the biggest function in the original code as well. In the original code it included the main logic with all the smaller functions which were called from this function.

2. Every function which was only defined, but not called, was removed: There were some functions in the original code which were only defined but were called from another classes. We decided to remove them since we wanted the experiment code to be a one class code which is short enough for such an experiment.

3. Every function call line was replaced with the function definition: The code needed to be functionless, so to achieve this we inlined all the functions by replacing each function call by its code. The names of the function's arguments were replaced with the names of the arguments the function was called with.

4. All not related classes were removed: There were some additional classes which were only defined but not used. We remove them to prevent the code from being too long.

5. A function with ambiguous name was renamed to a different name: There was one function in the original code which was defined with the same name as a different function from a different module, and both functions were called in the code. To prevent the experimenters from being confused, and to give the scripts a way to differentiate between those two functions, we renamed the function which was defined in the code from "send" to "send_the_request".

6. All the comments remained as they are, except the description comments of the inlined functions: The comments in the original code which were added by its developers and described some flows in the code and the main function definition, remained as they are. We wanted the experimenters to refactor as real a code as possible, and comments is something that can be found in almost every code in the industry. However, the comments that described the inlined functions [3] were removed, since there was no a suitable place for them. If we were to write them above the code of the inlined functions, it could cause the experimenters to extract functions below these comments.

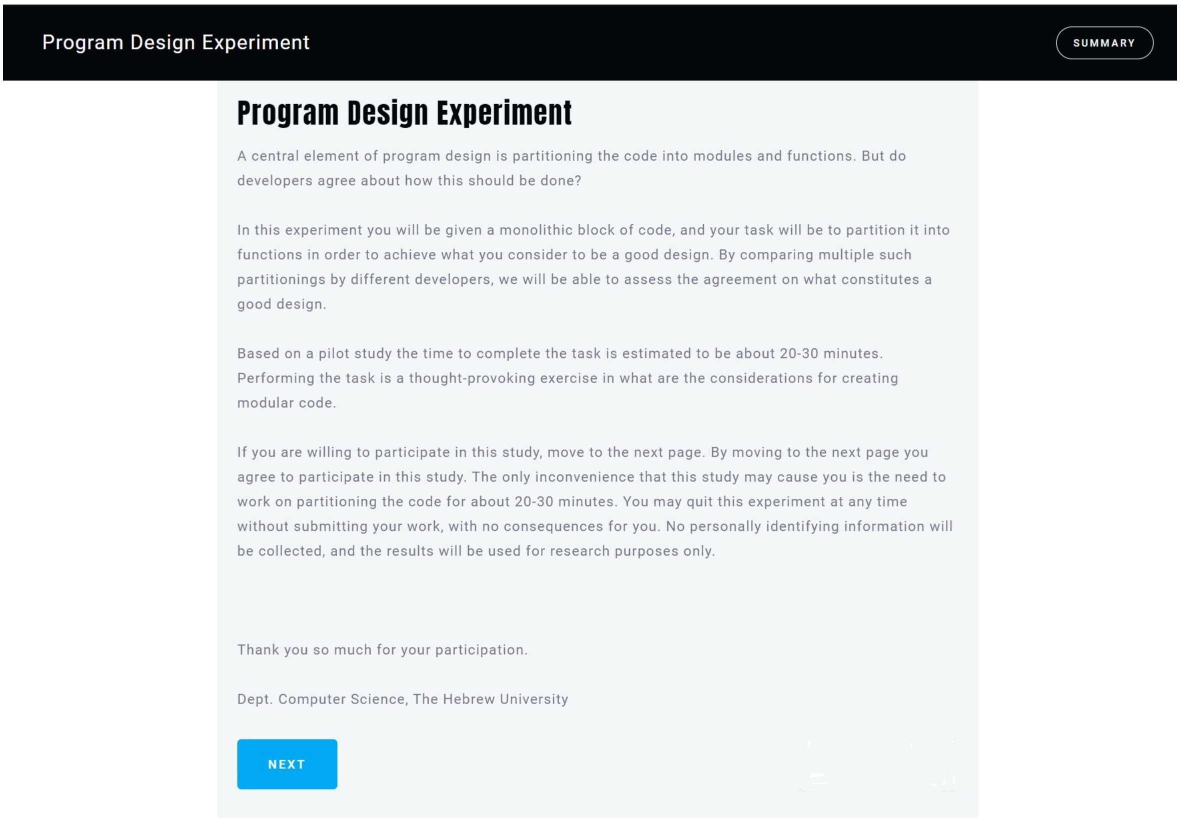
7. All the third-party functions calls remained as they are: Any function that was called from another module, remained as it is. We didn't replace literally all the functions calls in the code with their definition code, since it's not what we wanted to achieve. We wanted the experimenters to refactor a code in a specific context, and not in such large context as all the python language. We wanted to illustrate a code which was written in some company by some developer in a monolith design. Such code must contain third-party functions calls.

4.3 Ethics

To do this experiment we finished the "CITI Program" course of "Social & Behavioral Research - Basic/Refresher" under requirements set by the Hebrew University and got an IRB approval for the experiment. We informed the experimenters that the experiment and the questions are not mandatory, they can quit the experiment whenever they want, and that there is no personal information collected - The experiment is completely anonymous. We provided to the experimenters an introduction page in which we wrote all the notes mentioned above and informed them that by moving to the next page they agree with these conditions [Figure 4.1].

4.4 Experiment

The experiment was created with "Google Forms", but since "Google Forms" does not allow to upload files anonymously (It requires to log in to upload, and the email of the uploader is marked in the results), and the experiment required the experimenters to upload files, we used the services of "Formfacade" which integrates with "Google Forms" and allows to add more functionalities that one of them is the ability to upload files anonymously. Additional service of "Formfacade" we used is the website creation from the form. It didn't give us additional functionality for the experiment, but it was much more beautiful and convenient to distribute it like that. The form contained an introduction [Figure 4.1] and 3 sections:



Program Design Experiment

A central element of program design is partitioning the code into modules and functions. But do developers agree about how this should be done?

In this experiment you will be given a monolithic block of code, and your task will be to partition it into functions in order to achieve what you consider to be a good design. By comparing multiple such partitionings by different developers, we will be able to assess the agreement on what constitutes a good design.

Based on a pilot study the time to complete the task is estimated to be about 20-30 minutes. Performing the task is a thought-provoking exercise in what are the considerations for creating modular code.

If you are willing to participate in this study, move to the next page. By moving to the next page you agree to participate in this study. The only inconvenience that this study may cause you is the need to work on partitioning the code for about 20-30 minutes. You may quit this experiment at any time without submitting your work, with no consequences for you. No personally identifying information will be collected, and the results will be used for research purposes only.

Thank you so much for your participation.

Dept. Computer Science, The Hebrew University

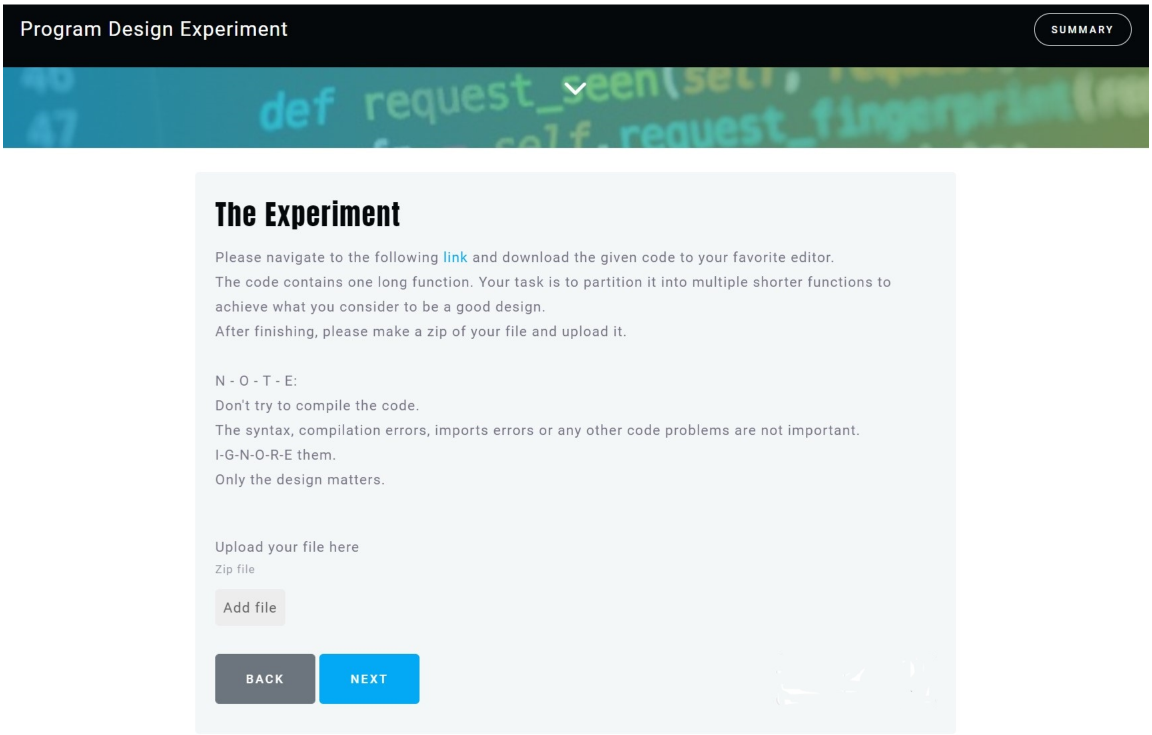
Figure 4.1: Experiment introduction

4.4.1 Background Questions

questions related to the experimenter background, such as age, programming education, development experience, etc.

4.4.2 Code Assignment

The experimenters got the code as a link to "GitHub" and needed to download it and open it with their preferable IDE. After they finished, they needed to make a zip file of their code and upload it. The instructions are in Figure 4.2.



The Experiment

Please navigate to the following link and download the given code to your favorite editor.
The code contains one long function. Your task is to partition it into multiple shorter functions to achieve what you consider to be a good design.
After finishing, please make a zip of your file and upload it.

N - O - T - E:
Don't try to compile the code.
The syntax, compilation errors, imports errors or any other code problems are not important.
I-G-N-O-R-E them.
Only the design matters.

Upload your file here
Zip file

Figure 4.2: Experiment instructions

4.4.3 Questions About the Code Assignment

Questions related to the code assignment the experimenters needed to do, such as refactoring considerations, importance of refactoring methods, etc.

4.4.4 Experiment Execution

The link to the website we created was sent to everyone we know that is familiar in some way with programming. In addition, the experiment was published in "Reddit.com" forum, in "r/SoftwareEngineering" and "r/Code". We wanted the experimenters to be as varied as possible.

The following graphs display the results of the experimenters' background distribution:

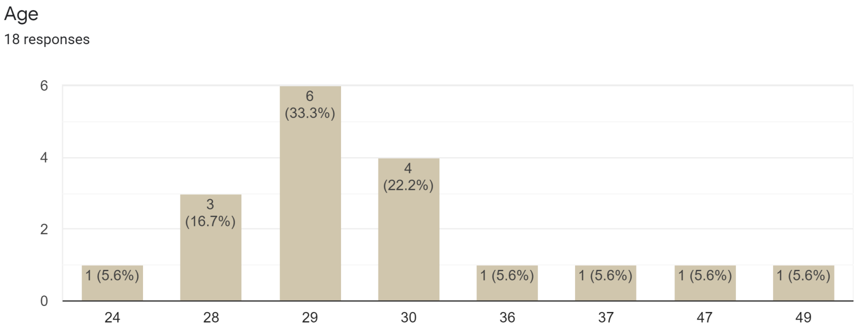


Figure 4.3: Experimenters' age distribution

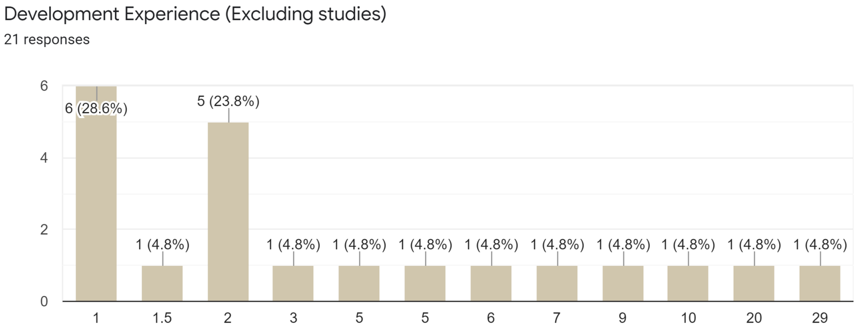


Figure 4.4: Experimenters' development experience distribution

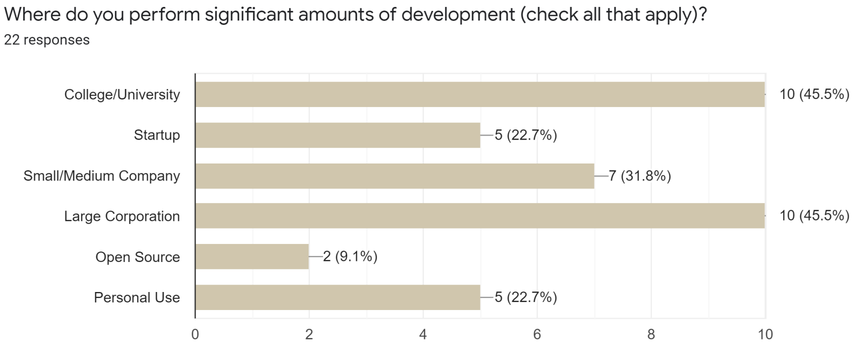


Figure 4.5: Experimenters' development environment distribution

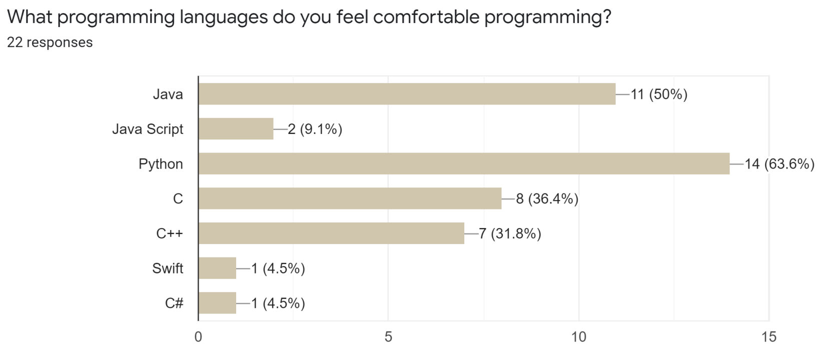


Figure 4.6: Experimenters' development language distribution

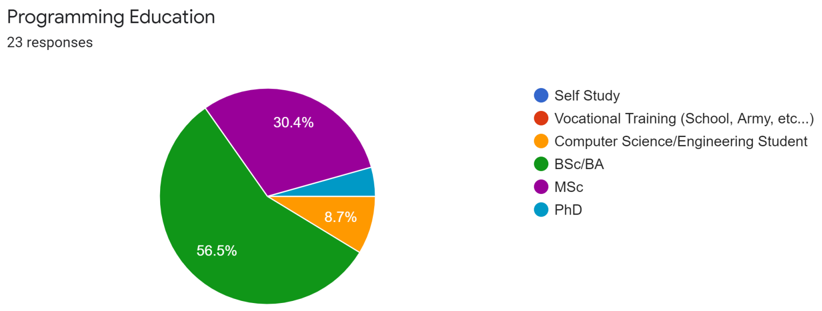


Figure 4.7: Experimenters' programming education distribution

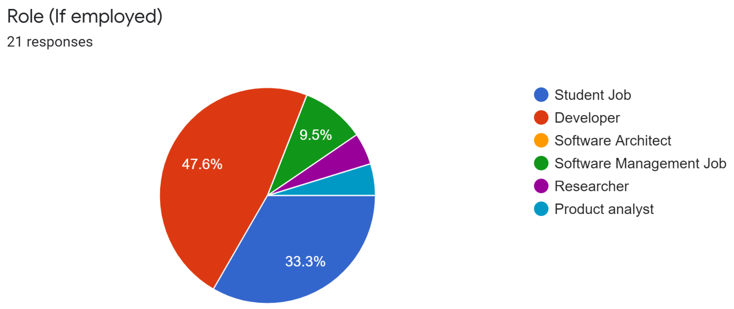


Figure 4.8: Experimenters' role distribution

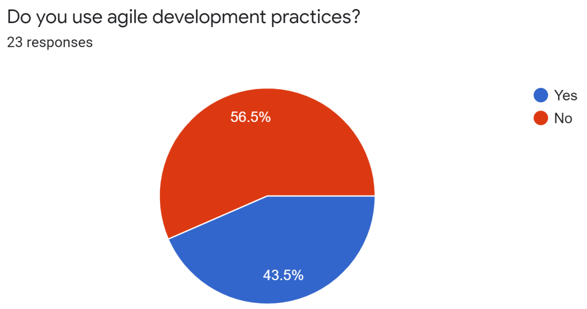


Figure 4.9: Experimenters agile development distribution

The number of responses varies because the questions in the experiment are not mandatory, and every experimenter can choose if he answers the question or not. In total there were 34 responses, but since 11 of them didn't include the code submission, they were removed from the experiment.

4.5 Scripts for Analysing the Results

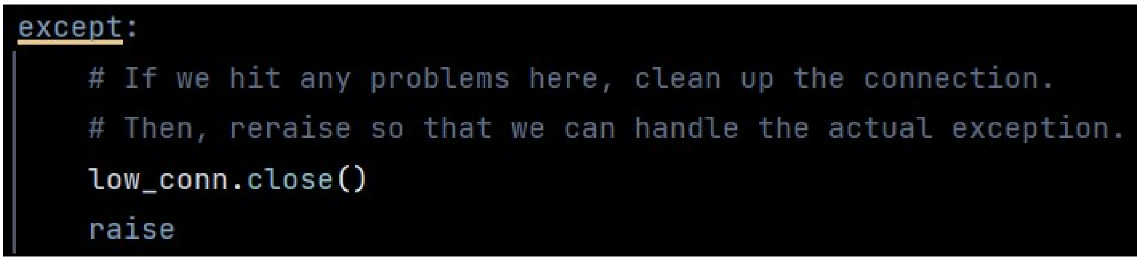
Since there were too many results for manual analysis to be accurate enough, we developed 10 scripts that clean, align, and analyze the results. Most of the analyses were done with regex matchers. Some of the scripts we developed are the following:

4.5.1 Stripping Script

The script strips the results from the irrelevant code parts such as whitespaces and blank lines. The purpose of this script is to align all the results and make them comparable by changing only the way the code was written, but not the content. Every developer has his own way to write code, and since the scripts are automatic, every whitespace, empty line, etc. can make two completely identical results to look different. The script works in the following way:

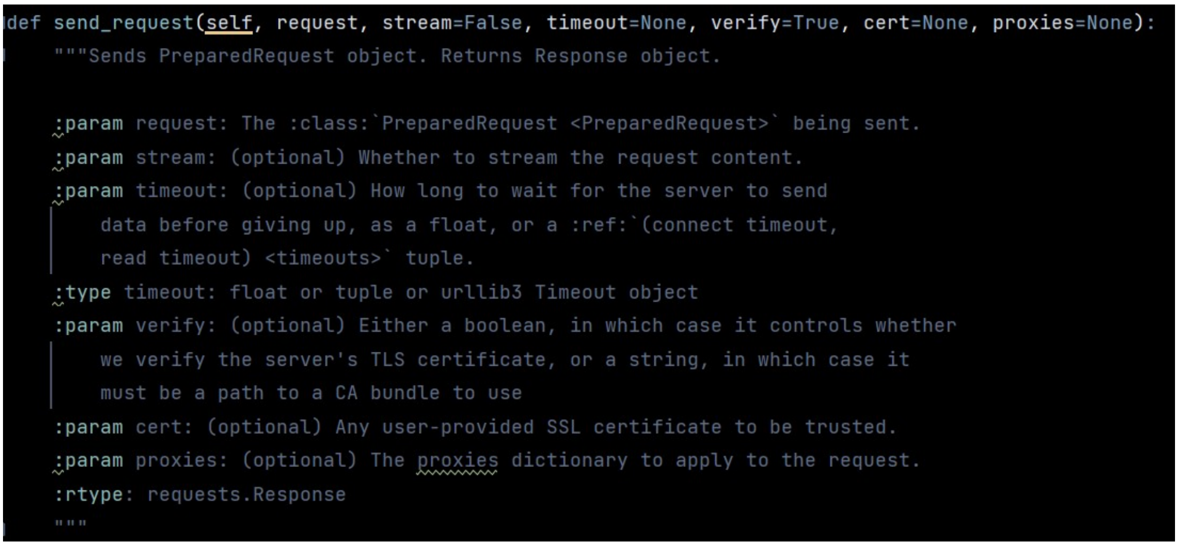
1. Removes all the import statements: The script runs over all the lines and checks each line if it starts with "from" or "import". If it does, the line is removed.

2. Removes all the comment lines: The script runs over all the lines and checks each line if it starts with <"""> or "#". If the line starts with "#", the line is removed. If the line starts with <"""> the line and every line after it are removed until there is another line that starts with <""">.



```
except:
    # If we hit any problems here, clean up the connection.
    # Then, reraise so that we can handle the actual exception.
    low_conn.close()
    raise
```

Figure 4.10: Example of lines that starts with '#'

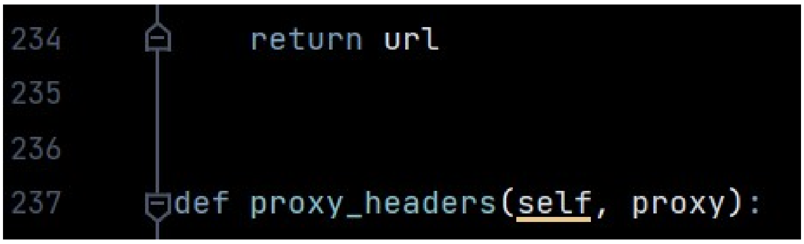


```
def send_request(self, request, stream=False, timeout=None, verify=True, cert=None, proxies=None):
    """Sends PreparedRequest object. Returns Response object.

    :param request: The :class:`PreparedRequest <PreparedRequest>` being sent.
    :param stream: (optional) Whether to stream the request content.
    :param timeout: (optional) How long to wait for the server to send
        data before giving up, as a float, or a :ref:`(connect timeout,
        read timeout) <timeouts>` tuple.
    :type timeout: float or tuple or urllib3 Timeout object
    :param verify: (optional) Either a boolean, in which case it controls whether
        we verify the server's TLS certificate, or a string, in which case it
        must be a path to a CA bundle to use
    :param cert: (optional) Any user-provided SSL certificate to be trusted.
    :param proxies: (optional) The proxies dictionary to apply to the request.
    :rtype: requests.Response
    """
```

Figure 4.11: Example of lines that starts with <""">

3. Removes all the spaces between lines: The script runs over all the lines and checks if the line is an empty line. If it is, the line is removed.

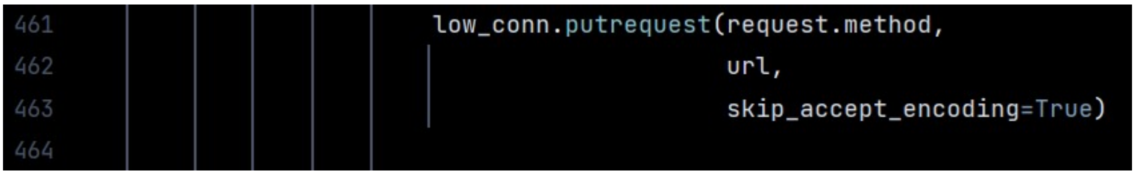


```
        return url


def proxy_headers(self, proxy):
```

Figure 4.12: Example of lines with spaces between them

4. If a line was divided to multiple lines, the script joins them to one line: The script runs over all the lines and checks if there is an opening bracket "(" but no closing one ")". In this case, every line after that is concatenated to the line until there is a line with a closing bracket.



```
low_conn.putrequest(request.method,
                    url,
                    skip_accept_encoding=True)
```

Figure 4.13: Example of line that divided to multiple lines

5. Removes all the white spaces at the beginning and at the end of each line: The script runs over all the lines and checks each line if it has white spaces at the beginning or at the end. If it does, these white spaces are removed. Since Python is a language where the white spaces are part of the syntax, this step must be done at the end. The purpose of this step is to make the comparison between the lines much easier.

4.5.2 Comparing Lines Script

The results were compared to the original code. This comparison was completely based on comparing lines. For instance, to identify which part of the experimenter's code was originally in a function, we needed to compare each line in the experimenter's code to the original code. At first, we compared between the lines just by using the Python "==" operator after we cleaned the white spaces, but it was not good enough. We had a lot of cases where the experimenter just changed the variable name, changed the argument name the function receives or just added underscore. This is only a small part of the cases we missed by comparing with the "==" operator.

Because of these problems, we moved to a different comparing method – "SequenceMatcher" from "difflib" Python library. By using this method, we could choose the similarity threshold, and all that was left was to find the best threshold to consider lines with small changes as equal, and completely different lines as unequal. The threshold we used for comparing lines was 0.95. The threshold for finding lines that were in the original code but were not in the results (lines that were not kept) was 0.75. (Will be explained in Section 5.6).

4.5.3 Functions Extractor Script

To even start talking about code refactoring and dividing it to functions we need first to extract the functions from the results. This script runs over all the lines in each submitted result, and extract the functions names and lines. The script retrieves the functions by counting the indentations and looking for the "def" word which defines a start of a function. It's done with the help of regex matchers. In more details - the script runs over all the lines and checks each line if it starts with "def ". If it does, the string after that and before the character "(" is taken as the name of the function. The number of white spaces before that is counted, and after that line the white spaces of each line are counted. As long as the number of white spaces is greater than the number of white spaces counted before the "def ", this line is taken as part of the function.

4.5.4 Functions Range Script

This script retrieves from the original code the range of lines that a given function was extracted from. At first, this script retrieves the lines themselves from a given function by using the script above [4.5.3]. After that, the line which starts the function is checked for its line number in the original code. If the line does not appear in the original code, the next line is checked. This process continues until a line appears in the original code. After that, the line which ends the function is checked for the line number in the original code in the same process. If the line does not appear in the original code, the previous line is checked until a line appears in the original code. At the end, the range is determined by these lines. In total, there were 61 ignored lines out of 142 functions. Most of the ignored lines (57) were the "return" lines which ended the defined functions. Obviously, these lines did not appear in the flattened code, because the flattened code contained only one main function. The remaining 4 lines were ignored due to extreme changes that were made by the experimenters or completely new lines the experimenters decided to add.

4.5.5 Order Change Script

This script checks if the lines' order was changed in relation to the original code. It's worth mentioning that the script does not check the order of the lines execution, but only their appearance order. At first, the script extracts all the functions in a submitted result with the help of the functions extractor script [4.5.3]. After that, the script runs over all the functions and for each function checks if there are 2 lines which appear in different order than they appear in the original code. This is done by looking at every pair of successive lines, retrieving their lines numbers in the original code, and validating that those numbers are also in the same order relation as the lines in the function. If the test fails, the script determines that the order was changed. If such lines were not found, the script determines that the order was not changed.

4.6 Manual Analysis

A manual scanning of the results was done. In this scanning we analysed the results and tried to find some patterns. We analysed the results by looking at the results from the scripts and analyse them in the code itself. We did that for two main purposes:

1. Verify the scripts: we wanted to do some manual testing to make sure the scripts are accurate.

2. Looking for patterns that can't be seen in the scripts: We tried to retrieve more information about the function extraction by understanding what the experimenters' main considerations for the refactoring were.

Chapter 5

Results

We analysed the coding task results both manually and automatically with scripts. Since our research questions are very complicated and vague as we have described in the introduction, we tried to analyse the results from as many as possible angels to get as much as possible information to compare. In addition, we also analysed the answers to the questions we have asked the experimenters. This chapter shows graphs of the results. The next chapter contains discussions of these results.

5.1 Histogram - Function Length

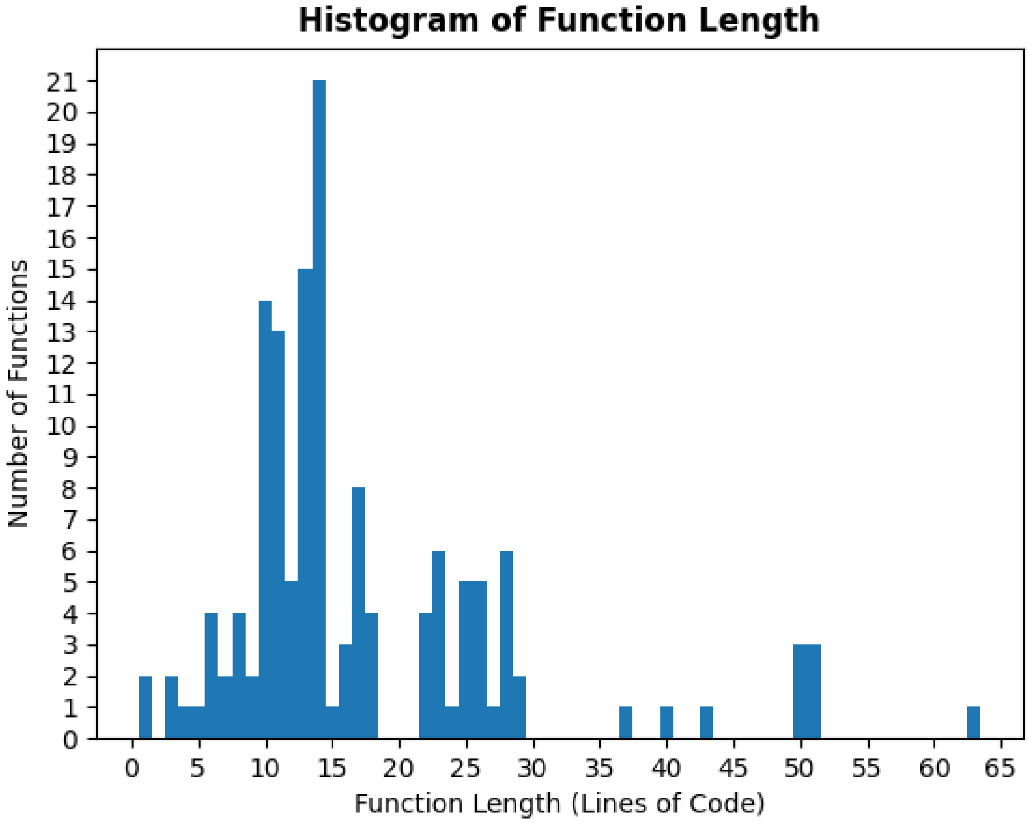


Figure 5.1: Histogram of function length

This histogram shows how long were the functions the experimenters chose to extract from the flattened code. This histogram was calculated regardless the experimenters themselves, that is there is no reference to the function owner. The main function was ignored. This is discussed below in section 6.4

5.2 Histogram - Number of Functions

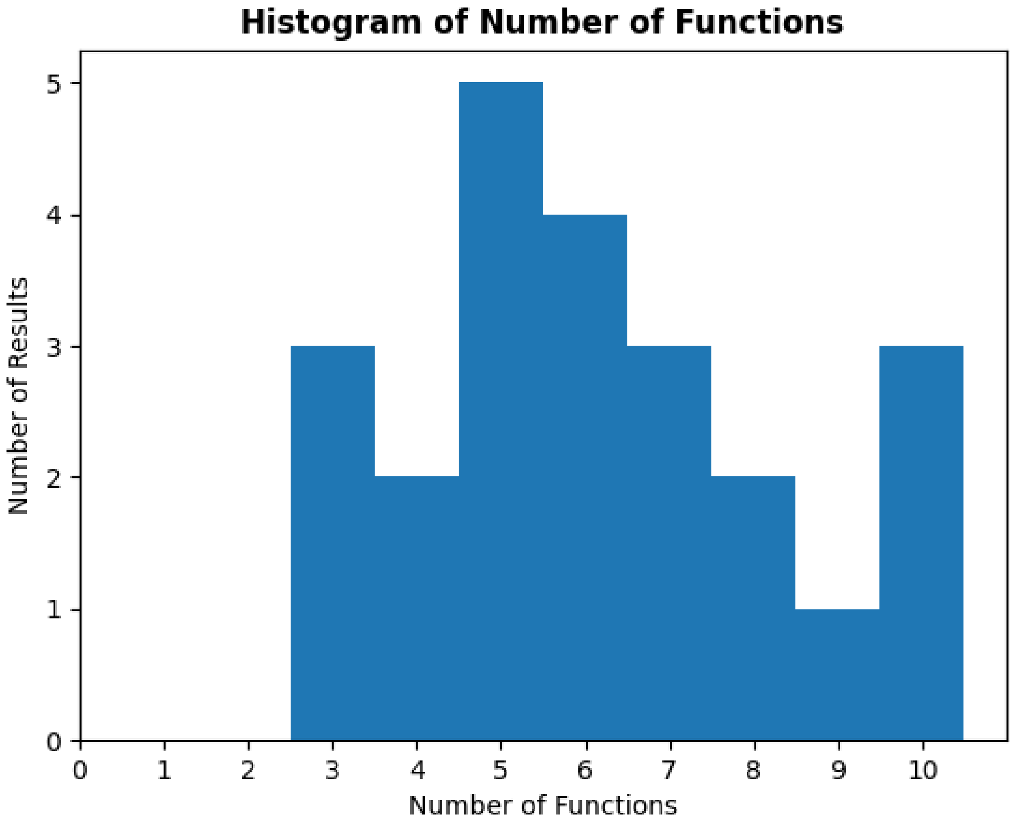


Figure 5.2: Histogram of number of functions

This histogram shows how many functions the experimenters chose to extract from the flattened code. This histogram was calculated by counting the number of functions each experimenter extracted. The main function was ignored. This is discussed below in section 6.5

5.3 Histogram - Lines That Start Function

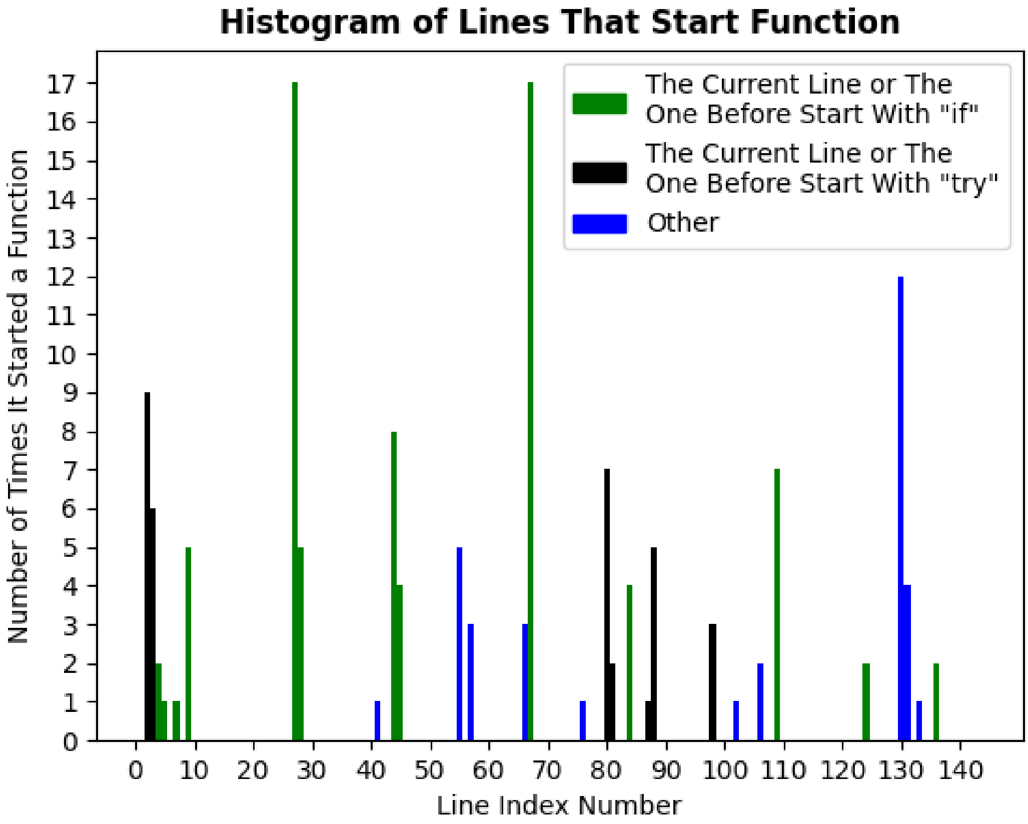


Figure 5.3: Histogram of lines that start function

This histogram was calculated by taking the flattened code as a reference, that is the number of each line that was found as a starting function line in the results, was determined by looking for its line number in the flattened code. If a line was not found in the flattened code, it was ignored. In this histogram there were no ignored lines. This is discussed below in section 6.1

5.4 Histogram - Lines That Start and End Function

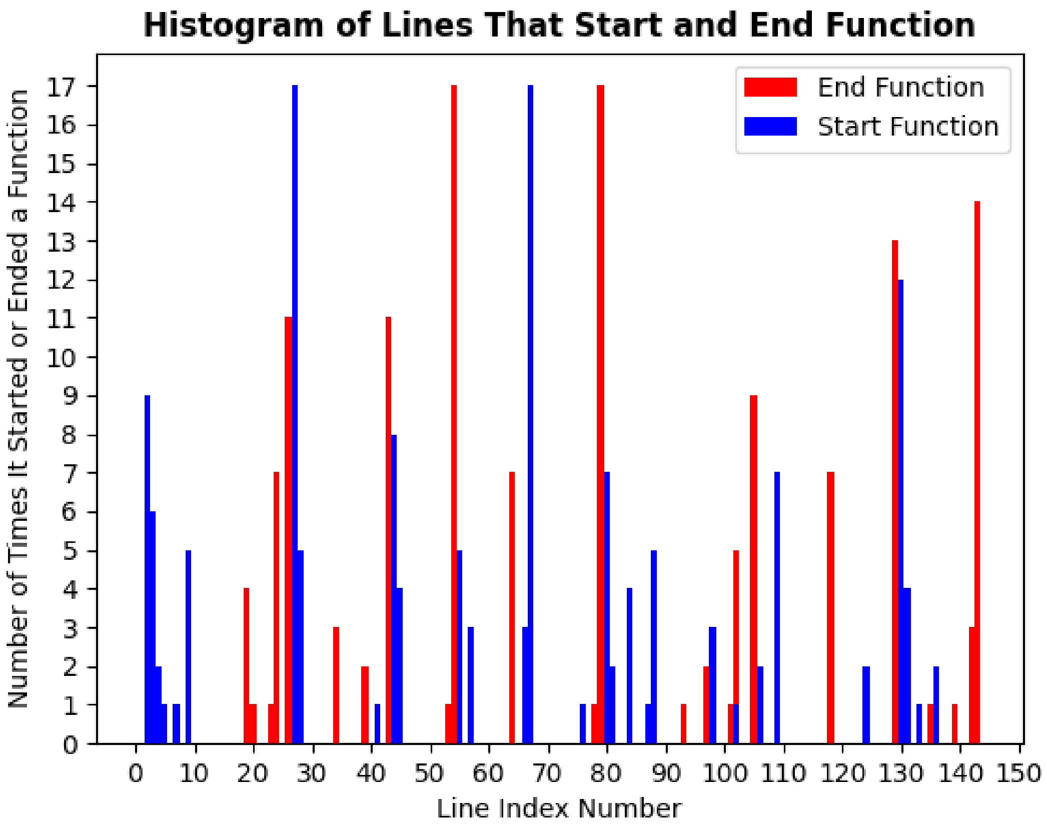


Figure 5.4: Histogram of lines that start and end function

This histogram was calculated by taking the flattened code as a reference, that is the number of each line that was found as an ending function line or starting function line in the results, was determined by looking for its line number in the flattened code. If a line was not found in the flattened code, it was ignored, and the closest line that was found was chosen, as described in the functions range script [4.5.4]. This script was used to create this histogram - the lines that start and the lines that end the functions were taken from the range created by the script. This is discussed below in section 6.1

5.5 CDF - Part of Code in Functions

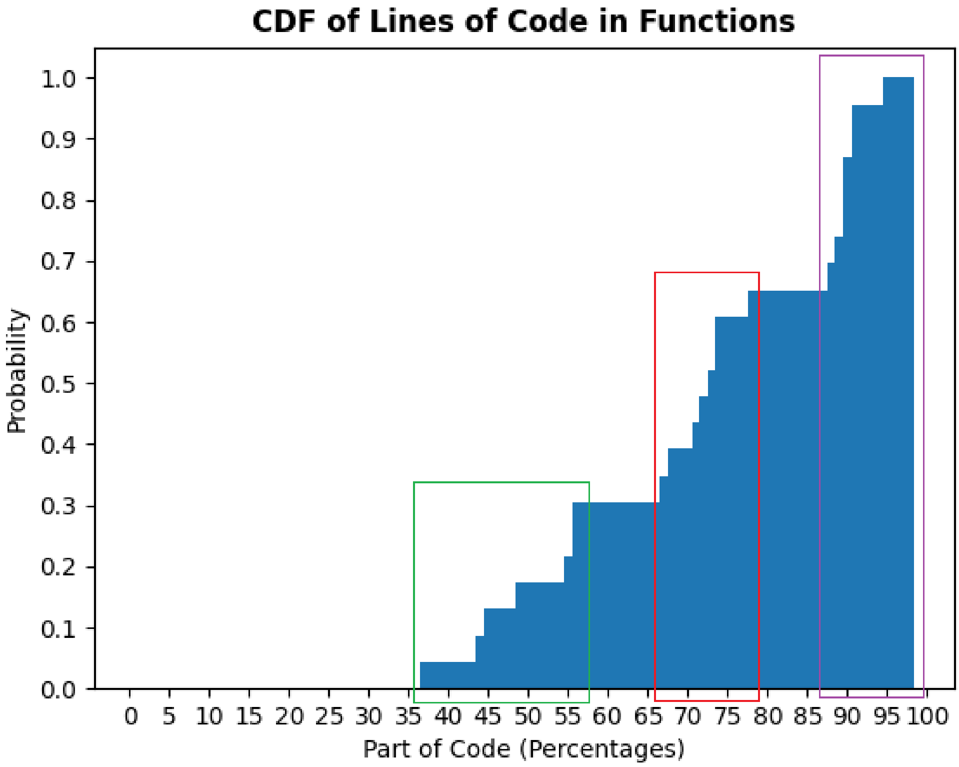


Figure 5.5: CDF of lines of code in functions

This CDF shows the part of code in percentages that was extracted to functions. This CDF was calculated by taking each result, counting the number of lines in functions (Except the main function) divided by the total number of lines, and multiplied by 100. From this data a CDF was created. Three main gaps were marked to be discussed below in section 6.3

5.6 Graph - Flattened Code in Functions Visualization

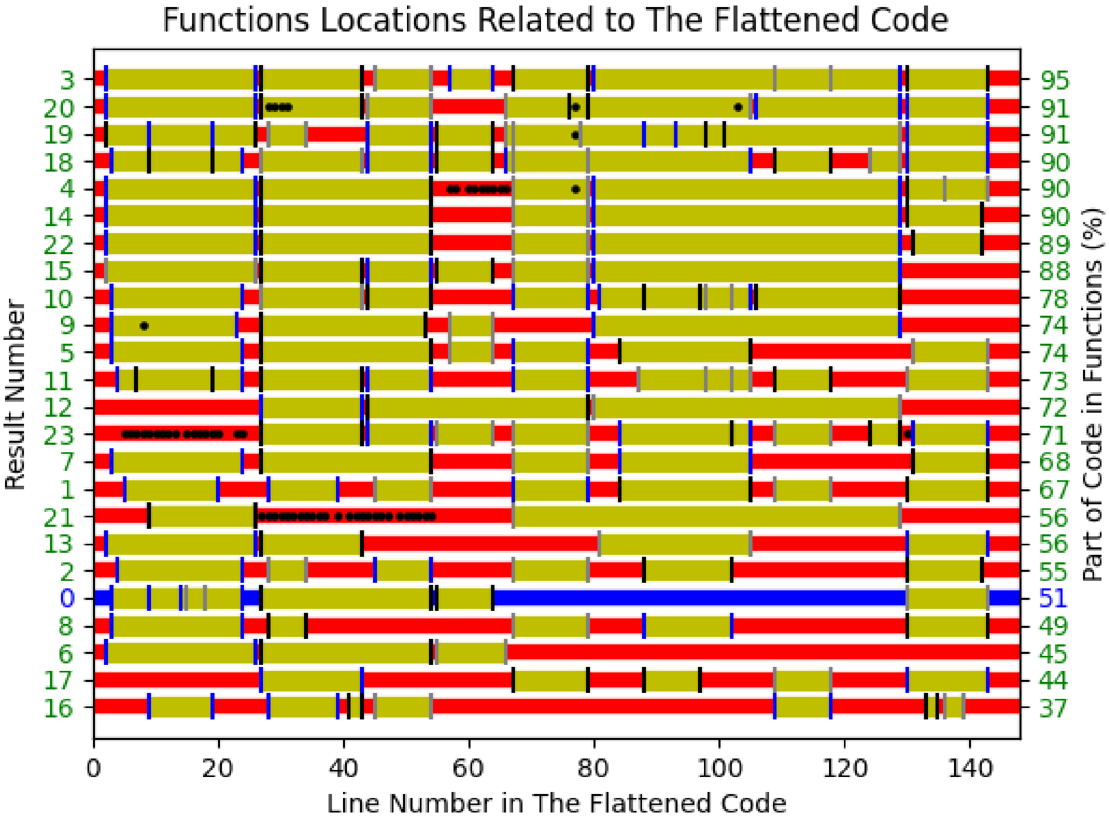


Figure 5.6: Visualization of functions locations related to the flattened code

This visualization shows the functions locations related to the flattened code. Every yellow block represents a function, and the red lines represent the code that was not extracted to a function. Even functions inside another function can be seen easily. The black spots represent the lines that were in the results but didn't appear in the original flattened code, meaning these lines were replaced by new lines or were changed too drastically to allow the comparator to find them in the original code. The results are sorted by the part of its code that was extracted to functions. The part of code in percentages that was extracted to functions is shown in the right Y axis. The original code is marked with the blue color. This is discussed below in section 6.3

5.7 Code Order Change Indicator

Only in result 19 the order of 1 line was changed. There is no explanation we could think of why it has been done.

5.8 Coding Opinion Questions

After the code assignment the experimenters were asked questions about refactoring methods and function extractions. Here are the Experimenters' responses to these questions:

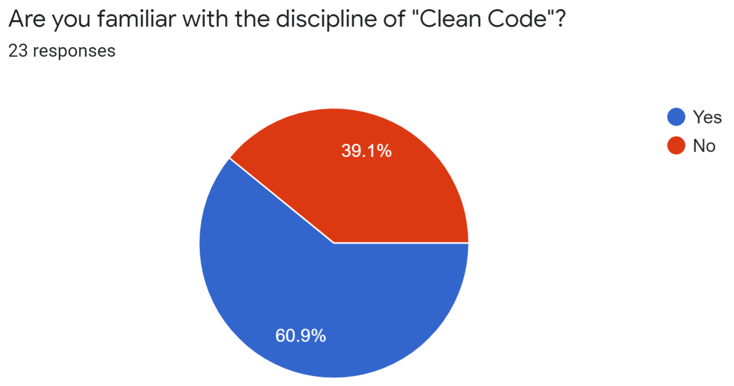

Are you familiar with the discipline of "Clean Code"?
23 responses

Figure 5.7: Familiarity with the "clean code" discipline distribution

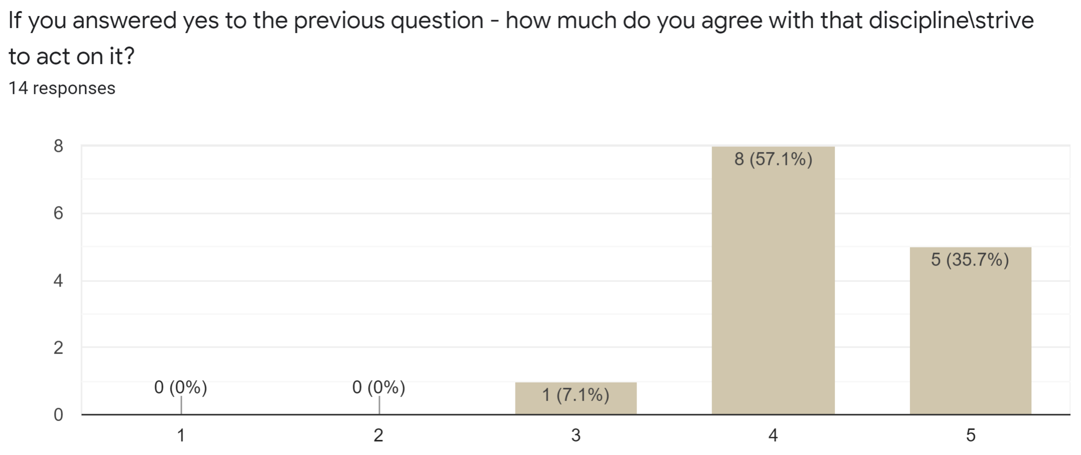

If you answered yes to the previous question - how much do you agree with that discipline\strive to act on it?
14 responses

Figure 5.8: The agreement level with "clean code" discipline
1 - Don't agree at all, 5 - Enthusiastically agree

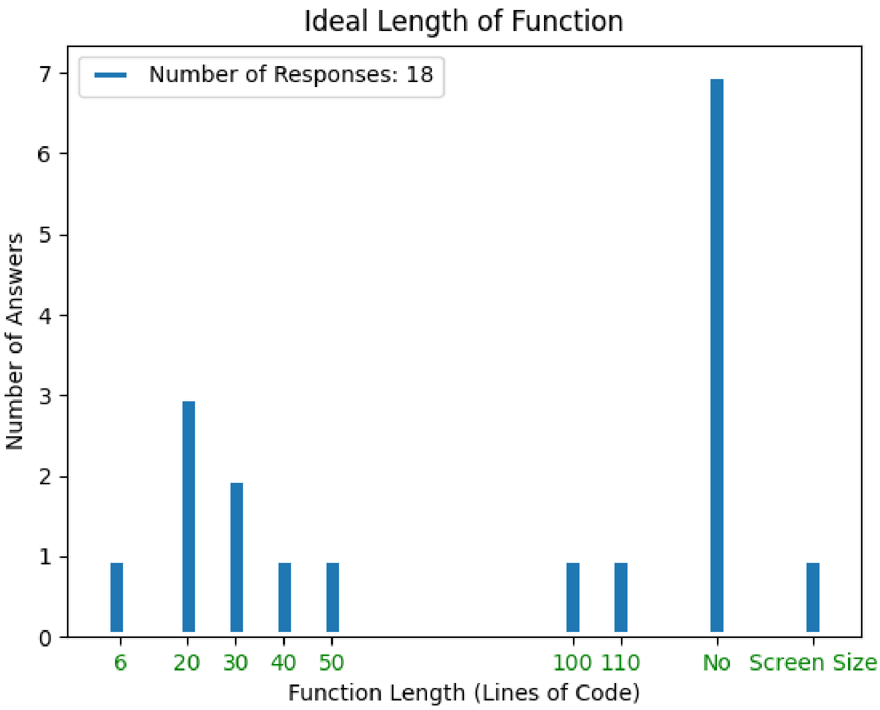
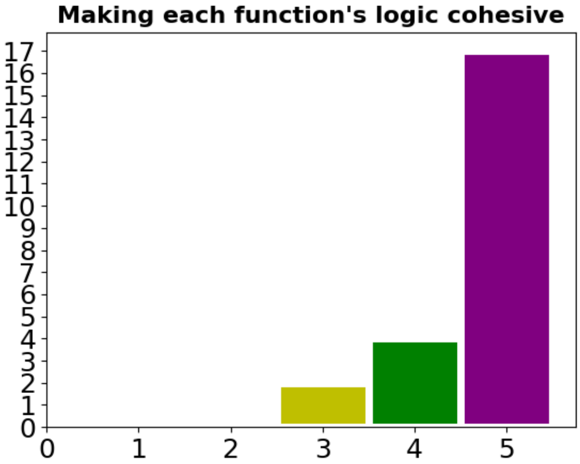
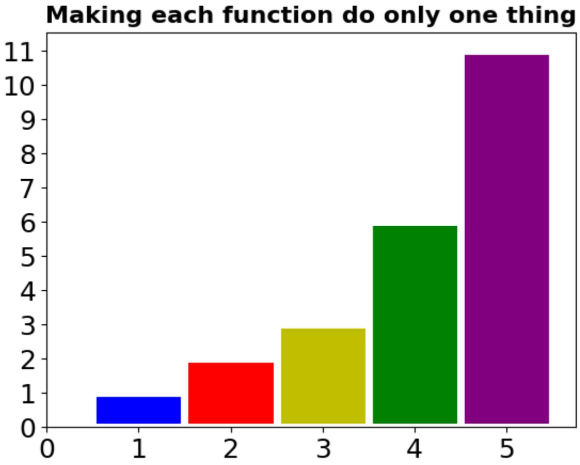
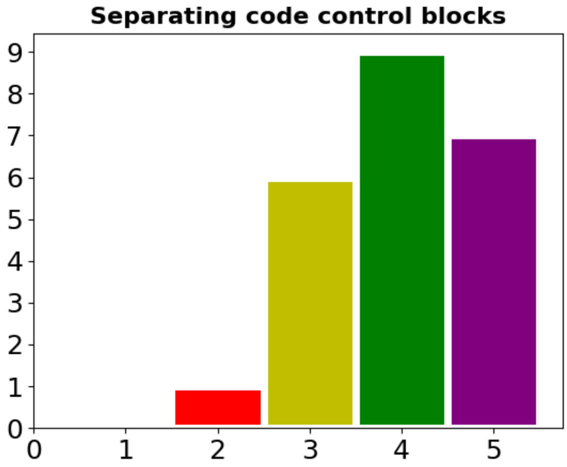
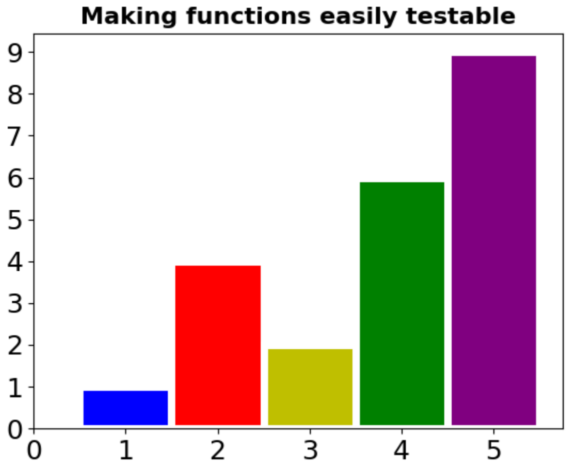
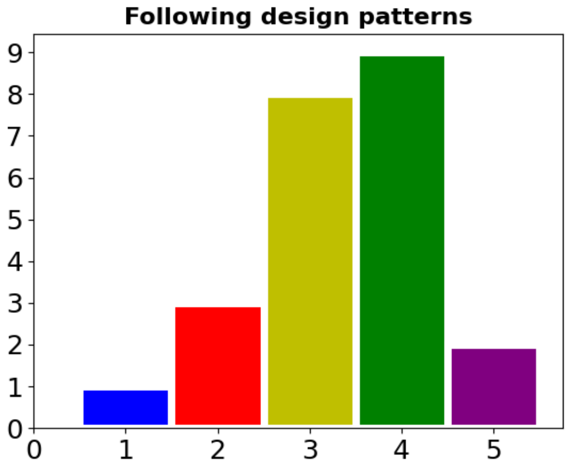


Figure 5.9: The ideal length of function

What were your main considerations when you extracted new functions? (1 - Not important, 5 - Very important)

	Consideration	Mean	Histogram
1	Making each function's logic cohesive	4.62	
2	Making each function do only one thing	4.10	

3	Separating code control blocks	3.93	
4	Making functions easily testable	3.89	
5	Following design patterns	3.41	

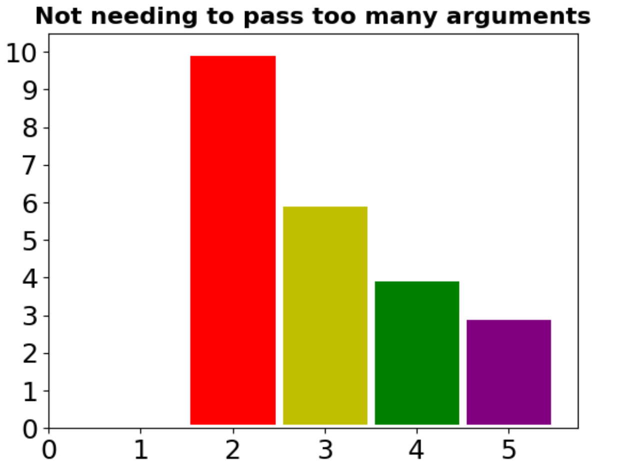
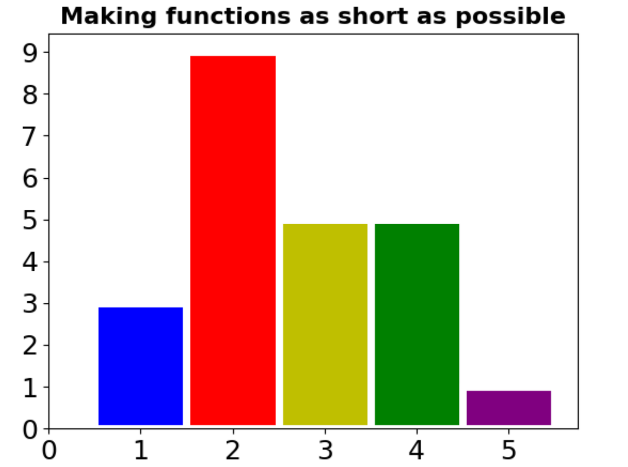
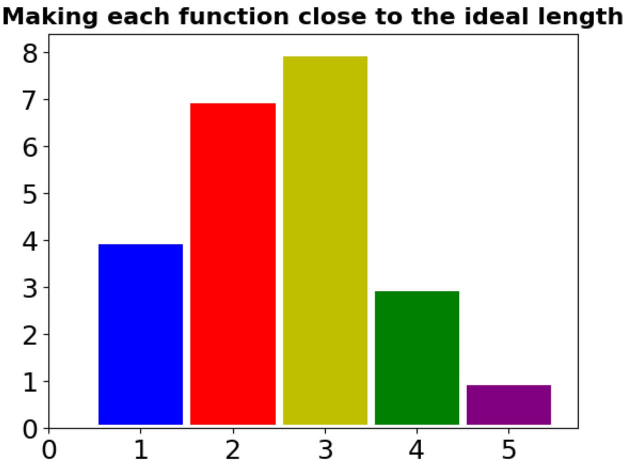
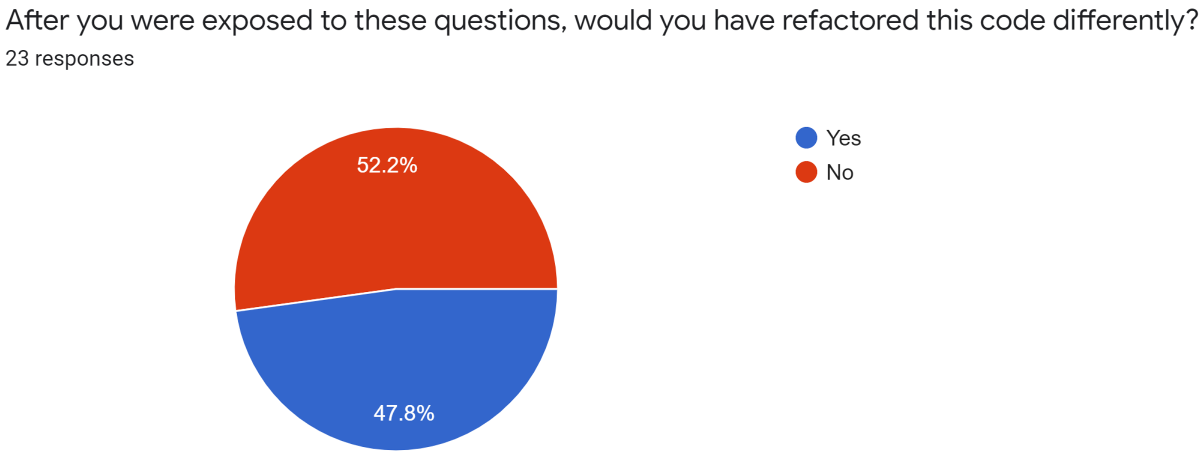
6	Not needing to pass too many arguments	3.10	
7	Making functions as short as possible	2.72	
8	Making each function close to the ideal length	2.65	

Table 5.1: Main considerations for function extraction



After you were exposed to these questions, would you have refactored this code differently?

23 responses

Figure 5.10: Different refactoring after being exposed to the questions

Chapter 6

Discussion

After analysing all the results we retrieved from the scripts, the manual analysis and the answers to the questions the experimenters have provided, we can conclude some conclusions.

6.1 Function Extraction

Our conclusion is that the main triggers for extracting a function are the "if" and "try" statements. As we can see in figure 5.3, most of the functions were extracted after these statements. We believe that there is some effect of the language syntax, since python is a language that is based on indentations, and when developers see a start of a new block of code, it may indicate to them to extract a function. When we look what was the trigger to end a function, our conclusion is that the main trigger for that was the same as starting a function – the end of an "if" or "try" block. In this case we also believe that there is a language syntax impact. Ending of a block of code can also indicate to end a function.

We also believe that extracting functions only by looking at blocks of code is the easiest way to extract (As we can see in the Reddit comments [appendix B]). It doesn't even require the experimenters to understand the code's logic, and since our experimenters were anonymous and were not paid for their time, we believe that they looked for the shortest path to do the experiment.

6.2 Considerations for Function Extraction

As we can see in Table 5.1, the most popular consideration was the "Making each function's logic cohesive" with an average score of 4.62/5. The consideration that fits to our conclusion discussed in section 6.1 is the third most popular consideration - "Separating code control blocks" with an average score of 3.93/5. We can't really measure the actual difference in the importance between these two considerations by looking in the results themselves, because "Making each function's logic cohesive" is a quite abstract consideration that can't be compared to the "Separating code control blocks" consideration that measured just by counting the lines that started a function after a statement that starts a new block of code. Because of that we do see a good fit of our conclusions to the experimenters' answers abut their considerations.

6.3 Code in Functions

As we can see in figure 5.5 and in figure 5.6 the majority of the experimenters extracted more than 70% of their code to functions. As we see it, on the one hand there is an understanding that code should be divided to functions and not be a part of a big main function, but on the other hand we think that there is a small bias that comes from the definition of the experiment. The instructions were to extract functions, and it might cause the experimenters to look only for function extraction. In addition, if we compare the results to the original code, the original code has only 51% of the code in functions, and it's placed in the fifth place from the end in this parameter, and that reinforces the conjecture that there may be a bias because of the experiment definition.

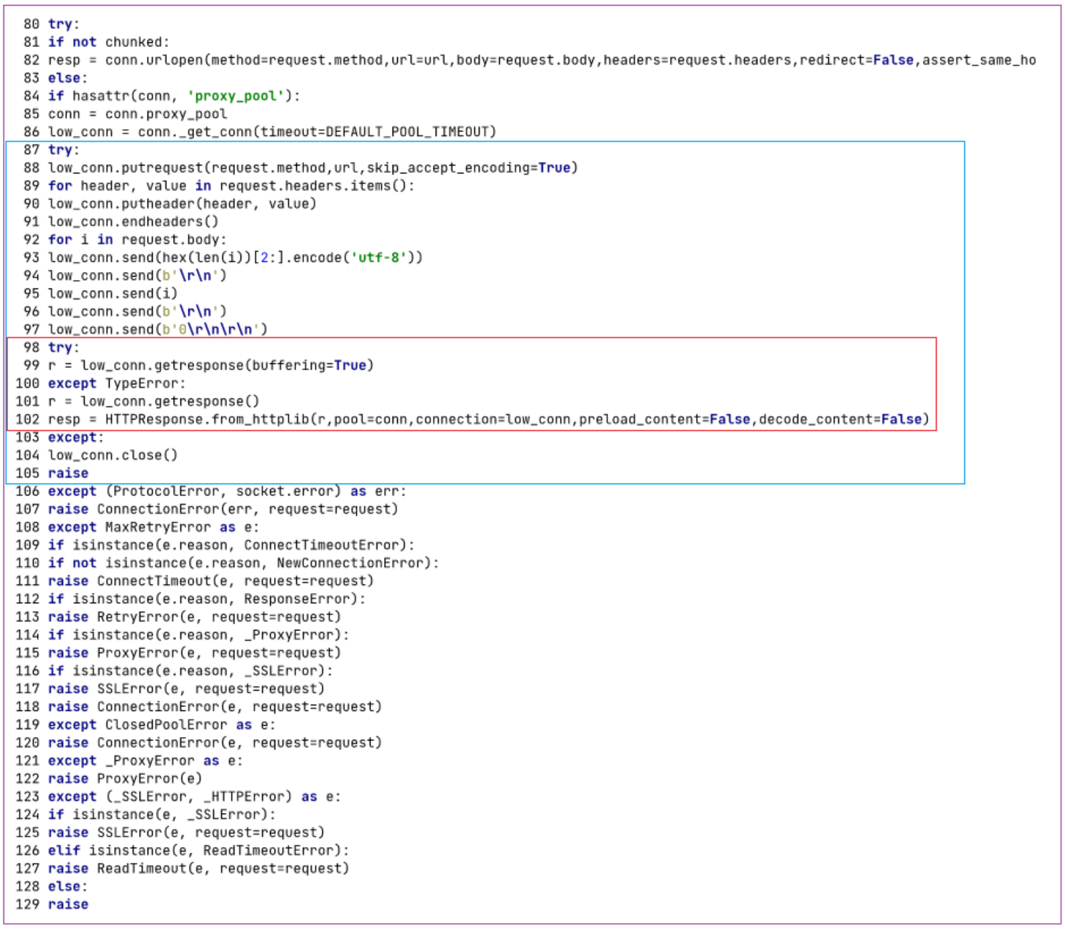
In figure 5.5 we mark three areas where the gap in the percentages was the biggest. These gaps represent the cases where all the function extractions before the gap are almost identical, but from this point the change is much more significant.

The first gap is the 56% to 67% change. We can see that most of the results below this gap, don't include any functions in the line range of 80-90. Only 2 of 8 results include functions in this area, compared to 3 of 8 results between the second and the third gap, and 8 of 8 results above the third gap. In total 11 of 16 results above the first gap have at least one function in this area. One more area to look at is the line range of 105-130. Only 3 of 8 results below the first gap include functions in this area, compared to 6 of 8 results between the second and the third gap, and 8 of 8 results above the third gap. In total 14 of 16 results above this gap have at least one function in this area. These parts of code contain quite a long error handling part, which includes three "try-except" blocks in total [Figure 6.1].

In the second gap, we can see that again the most significant part of the code that is different, is the same error handling code, that this time all of it was in one big function, and not just parts of it. The main difference between the gaps is which "try" block the experimenters chose to extract to a function, and whether the whole error handling code should be in one function, separated to several functions or not to be in a function at all.

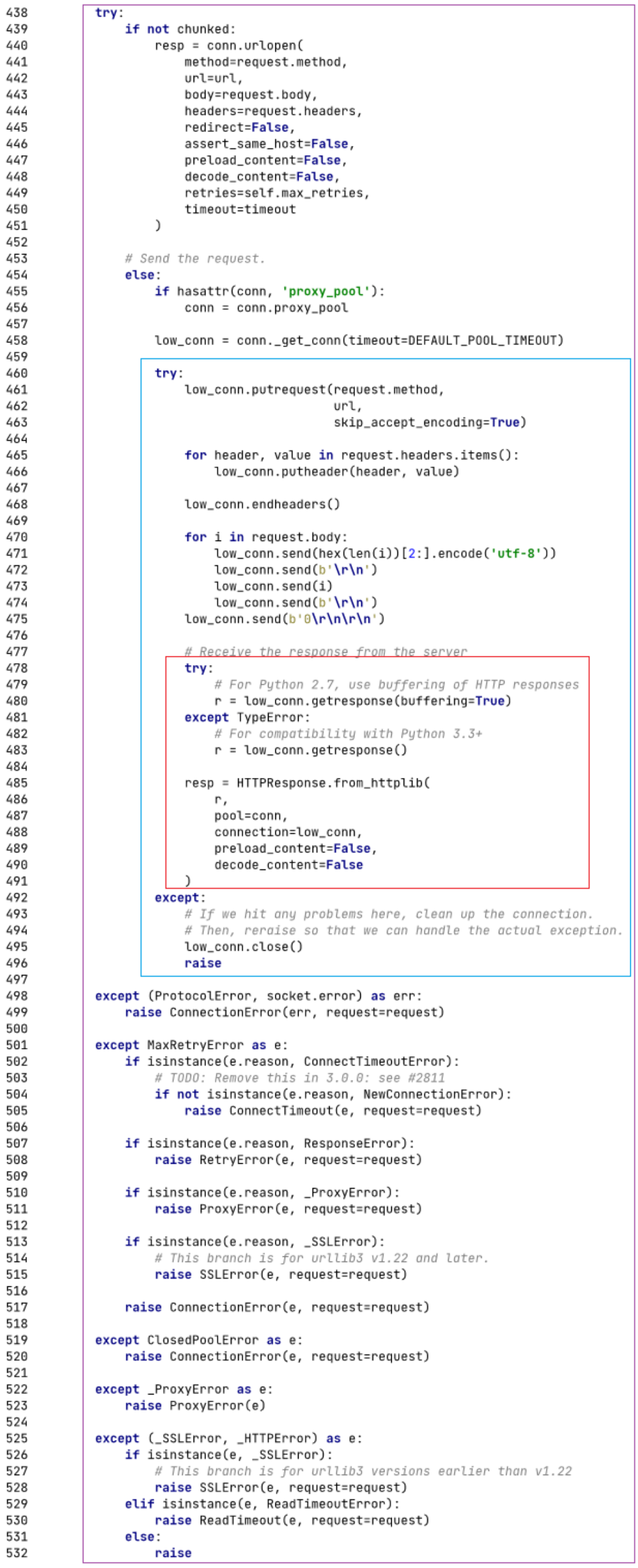
We can see that most of the experimenters who submitted the results below the first gap, chose to extract only the code in the second "try-except" block to a function, while most of the experimenters who submitted the results between the second and the third gap chose to include also the code inside the first "try" block (without the except block) into a function, and most of the experimenters who submitted the results above the third gap chose to include the whole "try-except" block, from the first one, into a function (include the except block).

In our opinion, the main reason for that is the fact that indeed there are different ways and designs to handle errors. Some believe that all the error handling logic should be handled with functions, and others believe that it should be one long code that handle all the errors together. It is worthwhile to mention that in the original code, this error handling part was not extracted to a dedicated function.



```
try:
if not chunked:
resp = conn.urlopen(method=request.method,url=url,body=request.body,headers=request.headers,redirect=False,assert_same_ho
else:
if hasattr(conn, 'proxy_pool'):
conn = conn.proxy_pool
low_conn = conn._get_conn(timeout=DEFAULT_POOL_TIMEOUT)
try:
low_conn.putrequest(request.method,url,skip_accept_encoding=True)
for header, value in request.headers.items():
low_conn.putheader(header, value)
low_conn.endheaders()
for i in request.body:
low_conn.send(hex(len(i))[2:].encode('utf-8'))
low_conn.send(b'\r\n')
low_conn.send(i)
low_conn.send(b'\r\n')
low_conn.send(b'0\r\n\r\n')
try:
r = low_conn.getresponse(buffering=True)
except TypeError:
r = low_conn.getresponse()
resp = HTTPResponse.from_httplib(r,pool=conn,connection=low_conn,preload_content=False,decode_content=False)
except:
low_conn.close()
raise
except (ProtocolError, socket.error) as err:
raise ConnectionError(err, request=request)
except MaxRetryError as e:
if isinstance(e.reason, ConnectTimeoutError):
if not isinstance(e.reason, NewConnectionError):
raise ConnectTimeout(e, request=request)
if isinstance(e.reason, ResponseError):
raise RetryError(e, request=request)
if isinstance(e.reason, _ProxyError):
raise ProxyError(e, request=request)
if isinstance(e.reason, _SSLError):
raise SSLError(e, request=request)
raise ConnectionError(e, request=request)
except ClosedPoolError as e:
raise ConnectionError(e, request=request)
except _ProxyError as e:
raise ProxyError(e)
except (_SSLError, _HTTPError) as e:
if isinstance(e, _SSLError):
raise SSLError(e, request=request)
elif isinstance(e, ReadTimeoutError):
raise ReadTimeout(e, request=request)
else:
raise
```

Figure 6.1: Error handling code - flattened and stripped
Purple box - first try-except block
Turquoise box - second try-except block
Red box - third try-except block



```
try:
    if not chunked:
        resp = conn.urlopen(
            method=request.method,
            url=url,
            body=request.body,
            headers=request.headers,
            redirect=False,
            assert_same_host=False,
            preload_content=False,
            decode_content=False,
            retries=self.max_retries,
            timeout=timeout
        )

    # Send the request.
    else:
        if hasattr(conn, 'proxy_pool'):
            conn = conn.proxy_pool

        low_conn = conn._get_conn(timeout=DEFAULT_POOL_TIMEOUT)

        try:
            low_conn.putrequest(request.method,
                                url,
                                skip_accept_encoding=True)

            for header, value in request.headers.items():
                low_conn.putheader(header, value)

            low_conn.endheaders()

            for i in request.body:
                low_conn.send(hex(len(i))[2:].encode('utf-8'))
                low_conn.send(b'\r\n')
                low_conn.send(i)
                low_conn.send(b'\r\n')
            low_conn.send(b'0\r\n\r\n')

            # Receive the response from the server
            try:
                # For Python 2.7, use buffering of HTTP responses
                r = low_conn.getresponse(buffering=True)
            except TypeError:
                # For compatibility with Python 3.3+
                r = low_conn.getresponse()

            resp = HTTPResponse.from_httplib(
                r,
                pool=conn,
                connection=low_conn,
                preload_content=False,
                decode_content=False
            )
        except:
            # If we hit any problems here, clean up the connection.
            # Then, reraise so that we can handle the actual exception.
            low_conn.close()
            raise

except (ProtocolError, socket.error) as err:
    raise ConnectionError(err, request=request)

except MaxRetryError as e:
    if isinstance(e.reason, ConnectTimeoutError):
        # TODO: Remove this in 3.0.0: see #2811
        if not isinstance(e.reason, NewConnectionError):
            raise ConnectTimeout(e, request=request)

    if isinstance(e.reason, ResponseError):
        raise RetryError(e, request=request)

    if isinstance(e.reason, _ProxyError):
        raise ProxyError(e, request=request)

    if isinstance(e.reason, _SSLError):
        # This branch is for urllib3 v1.22 and later.
        raise SSLError(e, request=request)

    raise ConnectionError(e, request=request)

except ClosedPoolError as e:
    raise ConnectionError(e, request=request)

except _ProxyError as e:
    raise ProxyError(e)

except (_SSLError, _HTTPError) as e:
    if isinstance(e, _SSLError):
        # This branch is for urllib3 versions earlier than v1.22
        raise SSLError(e, request=request)
    elif isinstance(e, ReadTimeoutError):
        raise ReadTimeout(e, request=request)
    else:
        raise
```

Figure 6.2: Error handling code - original code
Purple box - first try-except block
Turquoise box - second try-except block
Red box - third try-except block

6.4 Functions Length

As we can see in figure 5.1 most of the functions are of length 5-30 lines of code (80%), and 55% of the functions are of length 10-14. One of our goals was to check if there is an ideal length of function, and we actually asked the experimenters that question as it appears in figure 5.9. As we can see most of the answers were that there is no ideal length of function, but from those who did respond most of the responses were 20-30 lines of code. The result we see in figure 5.1 is even shorter than the ideal length the experimenters stated in the question they were asked in the experiment. So, there is a correlation between the responses and the results, and it also agrees with the no more than 20 lines suggested by Martin in Clean Code [C M08, p. 34].

6.5 Number of Functions

As we can see, every experimenter extracted at least 3 functions and 56% of them extracted more than 5 functions. The number of functions is corelated in some way to the functions length, since it's obvious that shorter functions increase the chance for more functions to be extracted.

6.6 Clean Code

As we can see in Martin's Clean Code book [C M08, p. 34], there is a quite clear statement about functions length - they should be as small as possible. That can affect the results as 60.9% of the experimenters answered that they are familiar with this discipline, so it might be the reason for the fact that 80% of the functions were 10-30 lines of code, and 55% were 10-14 lines of code. It fits to the Martin theory.

However, as we saw in the experimenters answers, "making function as short as possible" and "making each function close to the ideal length" were the two least important considerations for function extraction [Table 5.1], meaning the experimenters believe that small functions and ideal functions length are not as important as at least 6 other considerations we exposed them to, but in fact they do extract small functions. It's a quite interesting result, because it means the experimenters extracted small functions unintentionally, and we might carefully deduce that nowadays developers extract small function as part of their coding skills. Also possible that other considerations, like keeping the logic cohesive and doing just one thing, naturally lead to shorter functions. So short functions are not an end in themselves, but they happen to be the solution to other goals.

Chapter 7

Threats to validity

As we have introduced so far, it's quite hard to answer our research questions, let alone make an empirical experiment that will provide the answers. Thus, as in any experiment, we had some difficulties and threats to validity of our results. The following threats are the most significant ones.

7.1 Bugs

There might be scenarios where the scripts don't work as expected, for example: wrong line classification, wrong function detection, wrong lines comparison, etc. Obviously, all these cases have bad impact on the results and the results can be unreliable. To reduce these cases, we made several manual validation cycles. Since the number of results is not too big (23) we could manually validate all the results just by comparing the scripts' results to the actual code. Figure 5.6 contains the logic of all the scrips, that is it summarizes all the functions per experimenter - shows the number of functions per experimenter, the functions length, the start and the end of the functions, and functions location in relation to the flattened code. Once we validated this figure manually, we covered all the cases, and it means the scripts work as expected with very high probability.

7.2 Non-diverse Experimenters

We tried to find experimenters as diverse as we could. We reached students without experience, students with experience, experienced developers, different roles, different educations, different ages. The experiment is quite long and requires some time to participate, so finding experimenters and making sure they are very diverse was not so easy.

7.3 The Context of the Experiment

Refactoring and function extraction is the context of the experiment and not just writing code. It definitely affects the results, since the main goal of the experimenters is to extract functions and refactor. To overcome this threat, the experiment should be much longer, and the context should be to write a program from the beginning without any limitations. But in the scope of our research, it was not possible, since the experimenters were not paid for their effort, and it was not possible to recruit experimenters that will invest so much time and effort without being paid for that.

7.4 Lack of Generality

We did this experiment only on 1 function, so the question if it represents all the cases and if we can look on it as a general case is nontrivial. If we could do a much bigger experiment with much more experimenters, we could overcome this threat by using more than 1 function in the experiment.

7.5 The Number of Experimenters

We recruited only 33 experimenters of which only 23 did also the code assignment. It's not many results, and because of that the results may have a lot of bias, making conclusions is much more complicated and the conclusions themselves might not be accurate enough.

7.6 Experiment Environment

The experiment environment is different from the real developing environment. The goals are different, the lack of authority in the experiment environment, as well as the lack of incentives in the experiment environment and the anonymity effect can cause the experimenters to want to finish the experiment as fast as possible without investing enough time for quality. The conclusion is maybe to pay developers for such experiments.

Chapter 8

Conclusions

In this study we tried to answer some quite difficult questions like if there is an ideal length of functions, what are the consequences of too short or too long functions, what are the considerations to extract functions and if there is a difference between developers' statements and their actual actions. To answer these questions, we decided to make an empirical experiment with the hope that it will provide us some results that will help us to answer our questions.

Since the experiment involves people, obviously the results cannot be the same among all the experimenters, but we still can deduce conclusions if we will look for patterns and focus on the most common results. To prepare our experiment we first found a very common library, took from there one of the classes and flattened it to one big function. After that, we looked for diverse experimenters and asked them to refactor this one big function. In addition, we asked the experimenters to answer some questions related to their background and on the task itself.

We analysed the results in many different ways and from many angles. There are a lot of conclusions that can be deduced from our analysis, but we focused on the functions length and the considerations for function extraction. We found that most of the experimenters extracted small functions as it was described in the "Clean Code" book, and the main reason for the extraction was the code structure. It also fits the answers we have got from the experimenters' questions where most of them answered that they think functions should be small. We believe that we found the answers to our questions, and they met our expectations.

We also found out that in most of the results the code part that was in functions was bigger than in the original code. It probably was due to the fact we asked them explicitly to extract functions, but it can also be deduced from this that speaking with developers about functions length may spur them to extract small functions. This can be studied in future experiments.

Of course, there were some threats to validity as we have described, but we still believe that even if we had overcome these threats, the answers would have been the same. We believe that in addition to the results we have got, we produced some very good methods to analyse such complicated term as "software design" and written code. We developed a lot of scripts that analysed the written code in many ways such as functions length, the start and the end of the functions related to the original code, lines that start and end functions, order changes, lines in functions related to the whole code, and many more. Since this kind of experiments are almost non-existent, we hope our study and our methods will encourage additional experiments in this field.

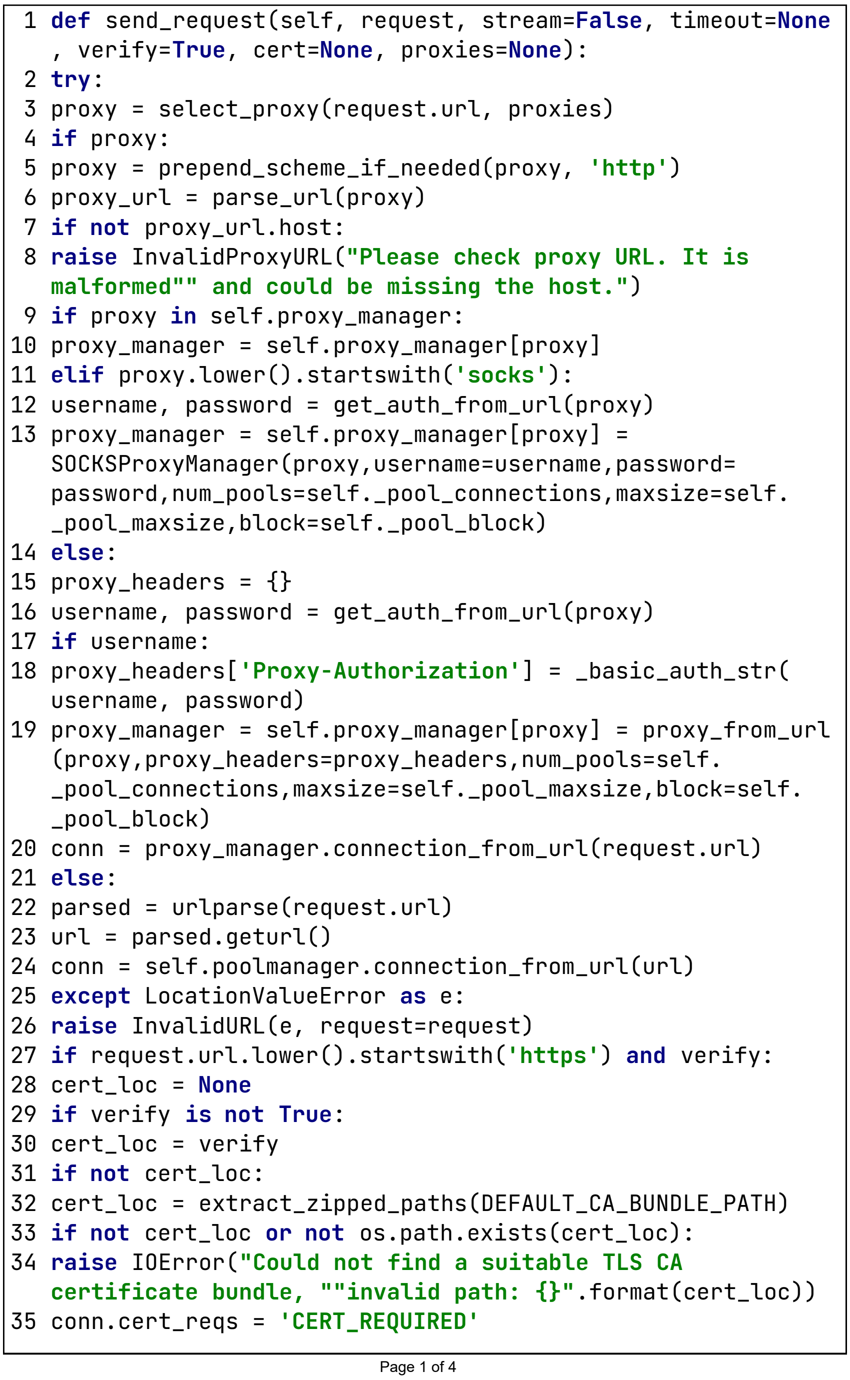
More complicated experiments can be done in order to overcome the threats to validity we have introduced. For example, more experimenters can be recruited to do the experiment, experimenters can be paid for the participation in the experiment, a different programming language can be chosen, different environment can be chosen to do the experiment, for example do the experiment as part of the regular development routine in existing product, and more.

Appendix A

The original flattened code

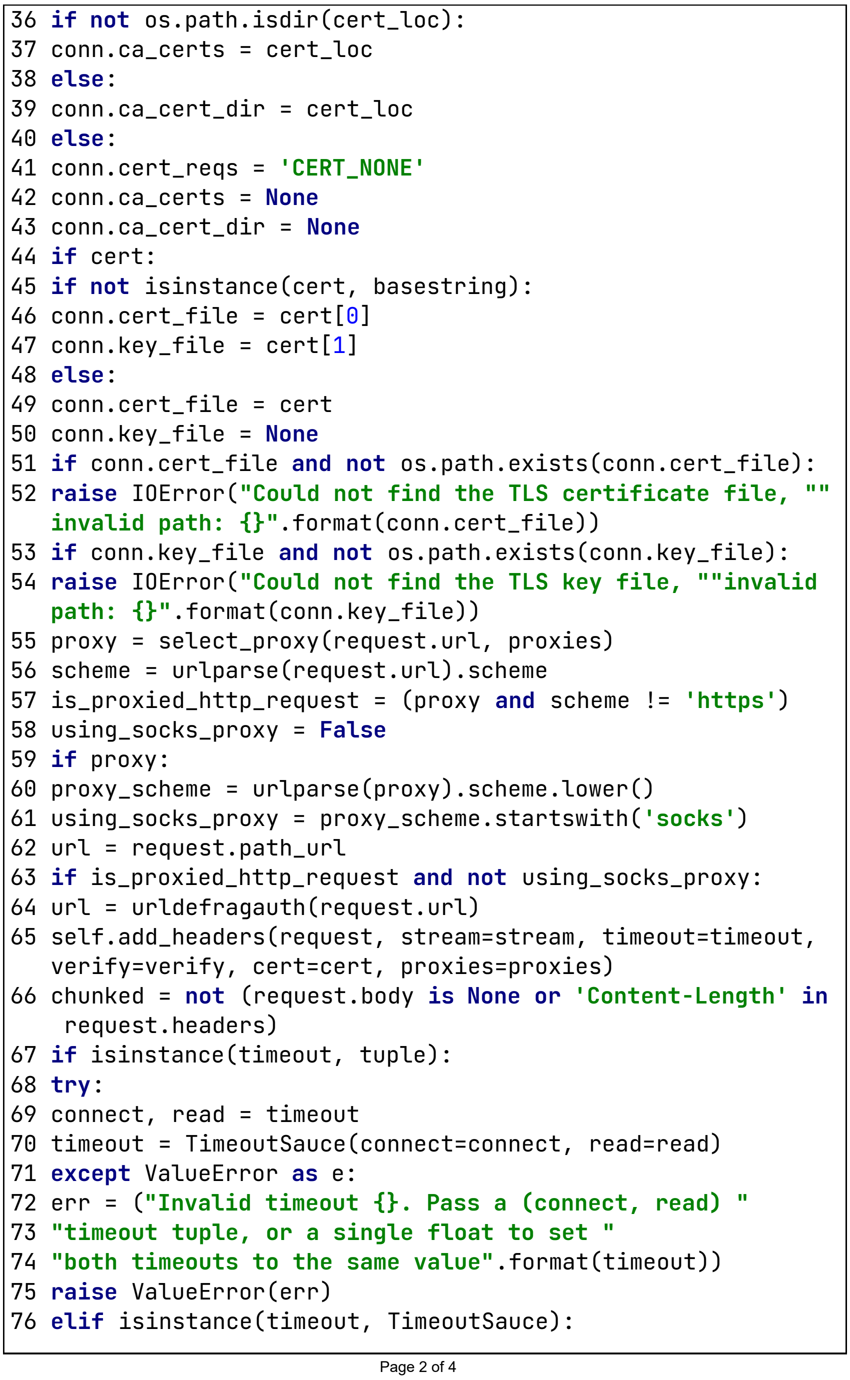
The following code is the flattened [4.2] stripped [4.5.1] code, that is the aligned code that all the submitted results were compared to.

Figure A.1: The original flattened code



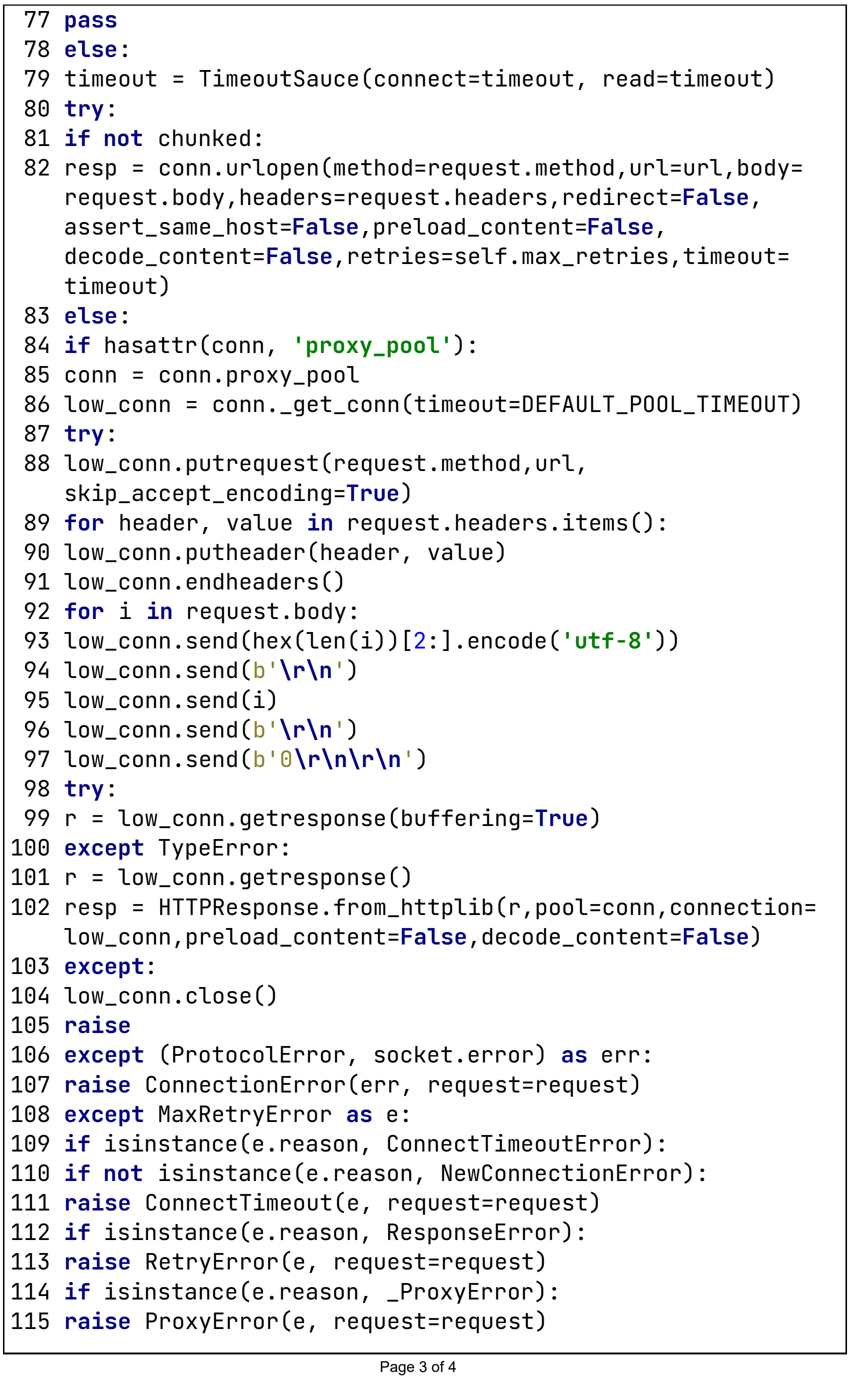
```python
def send_request(self, request, stream=False, timeout=None, verify=True, cert=None, proxies=None):
try:
proxy = select_proxy(request.url, proxies)
if proxy:
proxy = prepend_scheme_if_needed(proxy, 'http')
proxy_url = parse_url(proxy)
if not proxy_url.host:
raise InvalidProxyURL("Please check proxy URL. It is malformed"" and could be missing the host.")
if proxy in self.proxy_manager:
proxy_manager = self.proxy_manager[proxy]
elif proxy.lower().startswith('socks'):
username, password = get_auth_from_url(proxy)
proxy_manager = self.proxy_manager[proxy] = SOCKSProxyManager(proxy,username=username,password=password,num_pools=self._pool_connections,maxsize=self._pool_maxsize,block=self._pool_block)
else:
proxy_headers = {}
username, password = get_auth_from_url(proxy)
if username:
proxy_headers['Proxy-Authorization'] = _basic_auth_str(username, password)
proxy_manager = self.proxy_manager[proxy] = proxy_from_url(proxy,proxy_headers=proxy_headers,num_pools=self._pool_connections,maxsize=self._pool_maxsize,block=self._pool_block)
conn = proxy_manager.connection_from_url(request.url)
else:
parsed = urlparse(request.url)
url = parsed.geturl()
conn = self.poolmanager.connection_from_url(url)
except LocationValueError as e:
raise InvalidURL(e, request=request)
if request.url.lower().startswith('https') and verify:
cert_loc = None
if verify is not True:
cert_loc = verify
if not cert_loc:
cert_loc = extract_zipped_paths(DEFAULT_CA_BUNDLE_PATH)
if not cert_loc or not os.path.exists(cert_loc):
raise IOError("Could not find a suitable TLS CA certificate bundle, ""invalid path: {}".format(cert_loc))
conn.cert_reqs = 'CERT_REQUIRED'
```

Page 1 of 4



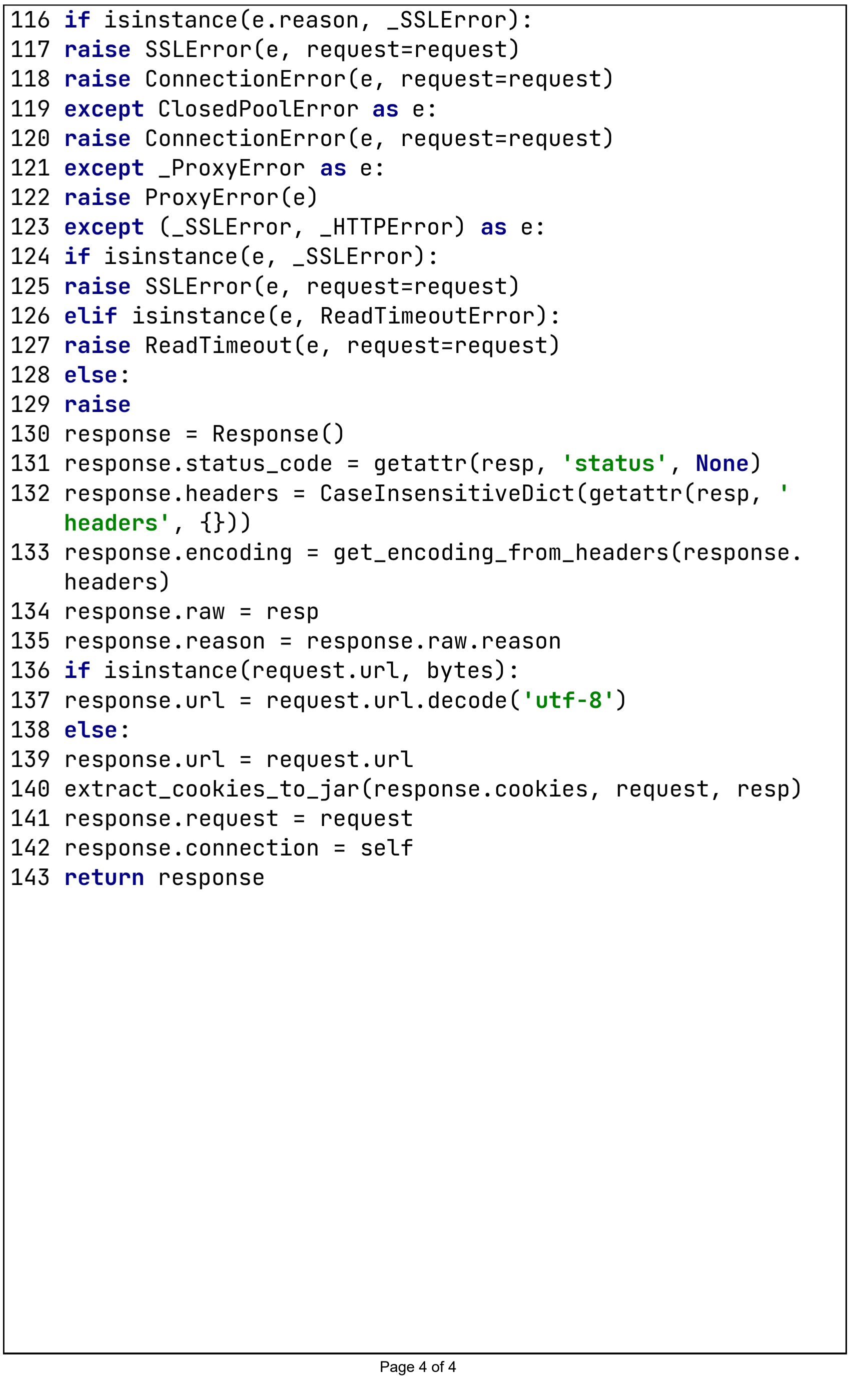
```
if not os.path.isdir(cert_loc):
conn.ca_certs = cert_loc
else:
conn.ca_cert_dir = cert_loc
else:
conn.cert_reqs = 'CERT_NONE'
conn.ca_certs = None
conn.ca_cert_dir = None
if cert:
if not isinstance(cert, basestring):
conn.cert_file = cert[0]
conn.key_file = cert[1]
else:
conn.cert_file = cert
conn.key_file = None
if conn.cert_file and not os.path.exists(conn.cert_file):
raise IOError("Could not find the TLS certificate file, ""invalid path: {}".format(conn.cert_file))
if conn.key_file and not os.path.exists(conn.key_file):
raise IOError("Could not find the TLS key file, ""invalid path: {}".format(conn.key_file))
proxy = select_proxy(request.url, proxies)
scheme = urlparse(request.url).scheme
is_proxied_http_request = (proxy and scheme != 'https')
using_socks_proxy = False
if proxy:
proxy_scheme = urlparse(proxy).scheme.lower()
using_socks_proxy = proxy_scheme.startswith('socks')
url = request.path_url
if is_proxied_http_request and not using_socks_proxy:
url = urldefragauth(request.url)
self.add_headers(request, stream=stream, timeout=timeout, verify=verify, cert=cert, proxies=proxies)
chunked = not (request.body is None or 'Content-Length' in request.headers)
if isinstance(timeout, tuple):
try:
connect, read = timeout
timeout = TimeoutSauce(connect=connect, read=read)
except ValueError as e:
err = ("Invalid timeout {}. Pass a (connect, read) "
"timeout tuple, or a single float to set "
"both timeouts to the same value".format(timeout))
raise ValueError(err)
elif isinstance(timeout, TimeoutSauce):
```

Page 2 of 4



```python
pass
else:
timeout = TimeoutSauce(connect=timeout, read=timeout)
try:
if not chunked:
resp = conn.urlopen(method=request.method,url=url,body=request.body,headers=request.headers,redirect=False,assert_same_host=False,preload_content=False,decode_content=False,retries=self.max_retries,timeout=timeout)
else:
if hasattr(conn, 'proxy_pool'):
conn = conn.proxy_pool
low_conn = conn._get_conn(timeout=DEFAULT_POOL_TIMEOUT)
try:
low_conn.putrequest(request.method,url,skip_accept_encoding=True)
for header, value in request.headers.items():
low_conn.putheader(header, value)
low_conn.endheaders()
for i in request.body:
low_conn.send(hex(len(i))[2:].encode('utf-8'))
low_conn.send(b'\r\n')
low_conn.send(i)
low_conn.send(b'\r\n')
low_conn.send(b'0\r\n\r\n')
try:
r = low_conn.getresponse(buffering=True)
except TypeError:
r = low_conn.getresponse()
resp = HTTPResponse.from_httplib(r,pool=conn,connection=low_conn,preload_content=False,decode_content=False)
except:
low_conn.close()
raise
except (ProtocolError, socket.error) as err:
raise ConnectionError(err, request=request)
except MaxRetryError as e:
if isinstance(e.reason, ConnectTimeoutError):
if not isinstance(e.reason, NewConnectionError):
raise ConnectTimeout(e, request=request)
if isinstance(e.reason, ResponseError):
raise RetryError(e, request=request)
if isinstance(e.reason, _ProxyError):
raise ProxyError(e, request=request)
```

Page 3 of 4



```
if isinstance(e.reason, _SSLError):
raise SSLError(e, request=request)
raise ConnectionError(e, request=request)
except ClosedPoolError as e:
raise ConnectionError(e, request=request)
except _ProxyError as e:
raise ProxyError(e)
except (_SSLError, _HTTPError) as e:
if isinstance(e, _SSLError):
raise SSLError(e, request=request)
elif isinstance(e, ReadTimeoutError):
raise ReadTimeout(e, request=request)
else:
raise
response = Response()
response.status_code = getattr(resp, 'status', None)
response.headers = CaseInsensitiveDict(getattr(resp, 'headers', {}))
response.encoding = get_encoding_from_headers(response.headers)
response.raw = resp
response.reason = response.raw.reason
if isinstance(request.url, bytes):
response.url = request.url.decode('utf-8')
else:
response.url = request.url
extract_cookies_to_jar(response.cookies, request, resp)
response.request = request
response.connection = self
return response
```

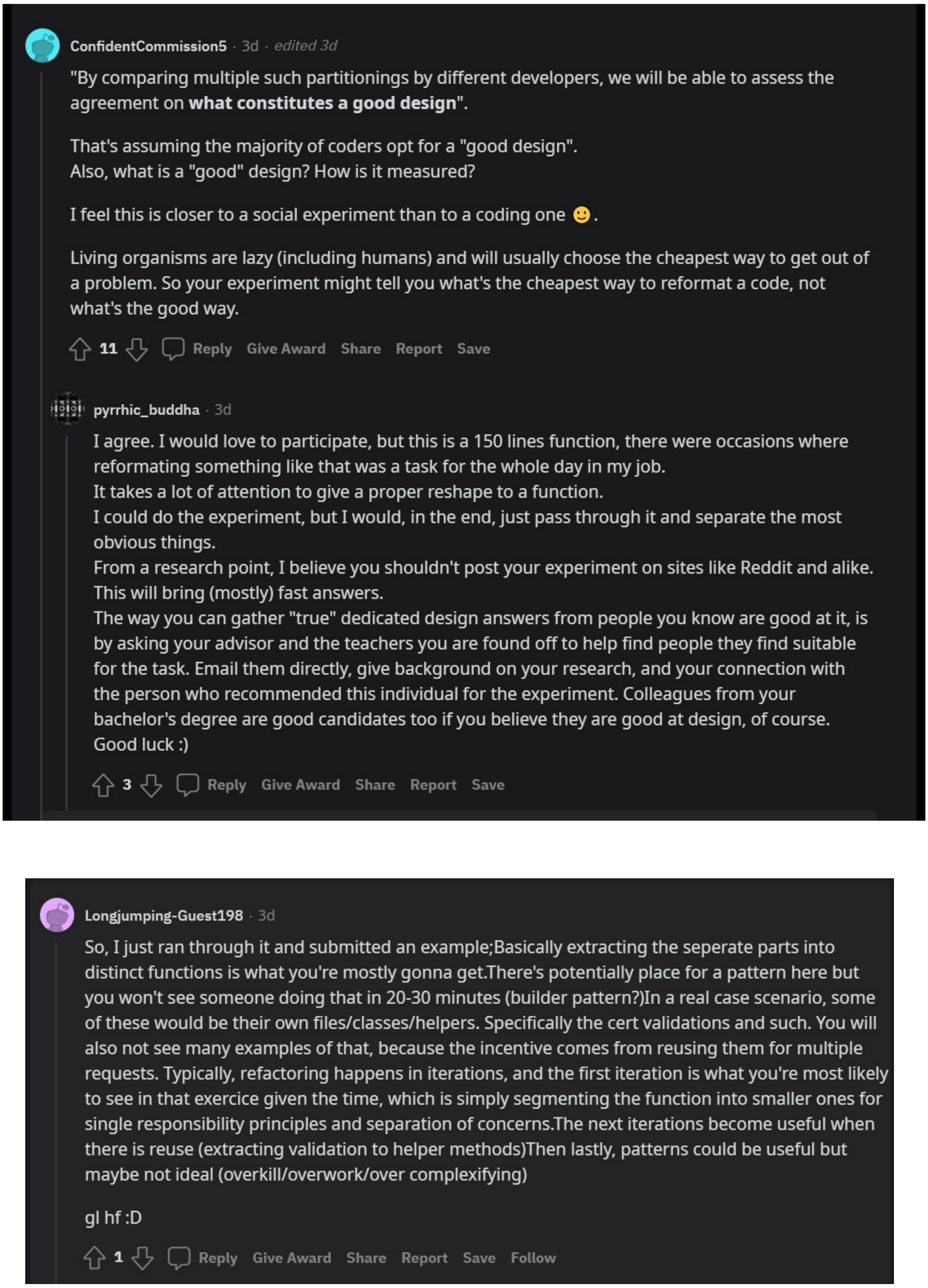
Page 4 of 4

Appendix B

Reddit comments

The following Reddit comments are the comments we have got after posting the experiment in Reddit.

Figure B.1: Reddit comments



ConfidentCommission5 · 3d · *edited 3d*

"By comparing multiple such partitionings by different developers, we will be able to assess the agreement on **what constitutes a good design**".

That's assuming the majority of coders opt for a "good design".
Also, what is a "good" design? How is it measured?

I feel this is closer to a social experiment than to a coding one 🙂.

Living organisms are lazy (including humans) and will usually choose the cheapest way to get out of a problem. So your experiment might tell you what's the cheapest way to reformat a code, not what's the good way.

pyrrhic_buddha · 3d

I agree. I would love to participate, but this is a 150 lines function, there were occasions where reformating something like that was a task for the whole day in my job.
It takes a lot of attention to give a proper reshape to a function.
I could do the experiment, but I would, in the end, just pass through it and separate the most obvious things.
From a research point, I believe you shouldn't post your experiment on sites like Reddit and alike. This will bring (mostly) fast answers.
The way you can gather "true" dedicated design answers from people you know are good at it, is by asking your advisor and the teachers you are found off to help find people they find suitable for the task. Email them directly, give background on your research, and your connection with the person who recommended this individual for the experiment. Colleagues from your bachelor's degree are good candidates too if you believe they are good at design, of course.
Good luck :)

Longjumping-Guest198 · 3d

So, I just ran through it and submitted an example;Basically extracting the seperate parts into distinct functions is what you're mostly gonna get.There's potentially place for a pattern here but you won't see someone doing that in 20-30 minutes (builder pattern?)In a real case scenario, some of these would be their own files/classes/helpers. Specifically the cert validations and such. You will also not see many examples of that, because the incentive comes from reusing them for multiple requests. Typically, refactoring happens in iterations, and the first iteration is what you're most likely to see in that exercice given the time, which is simply segmenting the function into smaller ones for single responsibility principles and separation of concerns.The next iterations become useful when there is reuse (extracting validation to helper methods)Then lastly, patterns could be useful but maybe not ideal (overkill/overwork/over complexifying)

gl hf :D

Bibliography

[C M08] Robert C. Martin.
Clean code.
1st ed.
Massachusetts: Pearson Education, Inc., 2008
(Cit. on pp. 6, 8, 10, 40).

[CAA15] Sofia Charalampidou, Apostolos Ampatzoglou, and Paris Avgeriou.
“Size and cohesion metrics as indicators of the long method bad smell: An empirical study”.
In: *PROMISE ’15: Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering.*
2015,
Pp. 1–10.
DOI: `https://doi.org/10.1145/2810146.2810155`
(Cit. on p. 9).

[Cha+18] Sofia Charalampidou, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Paris Avgeriou, Alexander Chatzigeorgiou, and Ioannis Stamelos.
“Structural Quality Metrics as Indicators of the Long Method Bad Smell: An Empirical Study”.
In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).*
2018.
DOI: `https://doi.org/10.1109/SEAA.2018.00046`
(Cit. on p. 9).

[D B+93] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig.
“Software complexity and maintenance costs”.
In: *Communications of the ACM* 36.11 (1993)
(Cit. on p. 9).

[FB18] Martin Fowler and Kent Beck.
Refactoring.
2nd ed.
Addison-Wesley Professional, 2018
(Cit. on p. 3).

[KF91] Geoffrey K. Gill and Chris F. Kemerer.
"Cyclomatic Complexity Density and Software Maintenance Productivity".
In: *IEEE Transactions on Software Engineering* 17.12 (1991), pp. 1284–1288.
DOI: `https://doi.org/10.1109/32.106988`
(Cit. on p. 9).

[Lan+15] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju.
"Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions".
In: *Journal of Software: Evolution and Process* 29.10 (2015).
DOI: `https://doi.org/10.1002/smr.1760`
(Cit. on p. 9).

[Leh96] M. M. Lehman.
"Laws of software evolution revisited".
In: *Software Process Technology.*
Ed. by Carlo Montangero.
Springer Berlin Heidelberg, 1996,
Pp. 108–124.
DOI: `https://doi.org/10.1007/BFb0017737`
(Cit. on p. 3).

[Mar01] Katsuhisa Maruyama.
"Automated method-extraction refactoring by using block-based slicing".
In: *ACM SIGSOFT Software Engineering Notes* 26.3 (2001), pp. 31–40.
DOI: `https://doi.org/10.1145/379377.375233`
(Cit. on p. 9).

[MB07] Timothy M. Meyers and David Binkley.
"An empirical study of slice-based cohesion and coupling metrics".
In: *ACM Transactions on Software Engineering and Methodology* 17.1 (2007), pp. 1–27.
DOI: `https://doi.org/10.1145/1314493.1314495`
(Cit. on p. 9).

[Par72] D.L. Parnas.
"On the Criteria to Be Used in Decomposing Systems into Modules".
In: *Pioneers and Their Contributions to Software Engineering.*
Springer Berlin Heidelberg, 1972.
DOI: `https://doi.org/10.1007/978-3-642-48354-7_20`
(Cit. on p. 2).

[TC11] Nikolaos Tsantalis and Alexander Chatzigeorgiou.
"Identification of extract method refactoring opportunities for the decomposition of methods".

In: *Journal of Systems and Software* 84.10 (2011), pp. 1757–1782.
DOI: https://doi.org/10.1016/j.jss.2011.05.016
(Cit. on p. 9).

[YKI12] Norihiro Yoshida, Masataka Kinoshita, and Hajimu Iida.
“A cohesion metric approach to dividing source code into functional segments to improve maintainability”.
In: *European Conference on Software Maintenance and Reengineering.*
2012.
DOI: https://doi.org/10.1109/CSMR.2012.45
(Cit. on p. 9).

Experimental Materials

[21] *The "Request" project that was used for the experiment.*
2021.
URL: `https://github.com/psf/requests/blob/master/requests/adapters.py`
(Cit. on p. 13).

[Bra21a] Alexey Braver.
The original flattened code that was given to the experimenters as the coding task.
2021.
URL: `https://github.com/AlexeyBraver/Program-Design-Experiment/blob/main/Flattened.py`.

[Bra21b] Alexey Braver.
The scripts we developed to analyse the results.
2021.
URL: `https://github.com/AlexeyBraver/Thesis-scripts`.