

How Developers Extract Functions: An Experiment

Alexey Braver

Dept. Computer Science
The Hebrew University
alexey.braver@mail.huji.ac.il

Dror G. Feitelson

Dept. Computer Science
The Hebrew University
feit@cs.huji.ac.il

Abstract

Creating functions is at the center of writing computer programs. But there has been little empirical research on how this is done and what are the considerations that developers use. We design an experiment in which we can compare the decisions made by multiple developers under exactly the same conditions. The experiment is based on taking existing production code, “flattening” it into a single monolithic function, and then charging developers with the task of refactoring it to achieve a better design by extracting functions. The results indicate that developers tend to extract functions based on structural cues, such as if or try blocks. The extracted functions tend to be short, but not necessarily due to an explicit desire to create short functions, but rather because of other considerations (like keeping logic cohesive or making functions easily testable). Finally, while there are significant correlations between the refactorings performed by different developers, there are also significant differences in the magnitude of refactoring done. For example, the number of functions that were extracted was between 3 and 10, and the amount of code extracted into functions ranged from 37% to 95%.

1. Introduction

It is generally accepted that programming should be done subject to a software design. The design can have a significant impact on the continued development and maintenance of the software. A good design can save a lot of effort for the developers, and as a result decrease the development time of the product and facilitate faster response to evolving needs. It is common to divide software design into two main parts:

- *High-Level Design*: The overall design of the product showing how the business requirements are satisfied. This is created by the software architect, and consists of dividing the functionality into modules and defining interfaces between them.
- *Low-Level Design*: The detailed design of the code based on refining the high-level design. Mostly, this is created on the fly by the system’s developers.

A core activity in low-level design is to define functions. Functions are the basic elements for structuring code. Creating functions is one of the first things that one learns when learning to program. Famous coding guides like Bob Martin’s *Clean Code* (Martin, 2009) and Steve McConnell’s *Code Complete* (McConnell, 2004) devote full chapters to writing functions. Martin, for example, opines that functions should be short, but also notes that this is based on experience and that there is no research to support it (Martin, 2009, pg. 34). And indeed, there has been relatively little research about functions and how they are defined. In particular, we have found no controlled experiments on this topic. A possible reason is that developers enjoy considerable freedom during low level design. Due to this freedom, it is impractical to conduct an experiment just by giving a coding task to the participants, and asking them to implement it, because it can be very difficult to compare the results.

Our solution to this problem is to base the experiment on refactoring a given code-base, rather than asking participants to produce new code. And while we do not answer Martin’s explicit question concerning the best length for functions, we do take a first step towards showing what sizes developers prefer and how function creation can be studied experimentally. Our main contributions in this work are:

- To devise a methodology for studying function creation by different developers, at least in the context of refactoring.
- To demonstrate that code structure provides cues for function extraction.
- To find that developers tend to use short functions even if this is not their explicit goal.

2. Background and Related Work

As noted above, Martin in *Clean Code* claims that functions should be as small as possible (Martin, 2009). A survey of practitioners indicates that there is wide acceptance of this suggestion (Ljung & Gonzalez-Huerta, 2022). Martin goes on to claim that the block of code within if statements, else statements, while statements, and so on should be one line long — and probably this line should be a function call with a descriptive name. This approach emphasizes explanation and documentation of the code as the reasons for creating functions. A similar stand is taken by Clausen, who uses the rule that functions should be no more than 5 lines of code as the title of his book on refactoring (Clausen, 2021).

Conversely, McConnell writes that functions which implement complex algorithms may be allowed to grow organically up to 100–200 lines, and that this is not expected to cause an increase in defects (McConnell, 2004).

The only study we know of which specifies a concrete optimal function size is Banker et al. (Banker, Datar, Kemerer, & Zweig, 1993). This 30 year old study analyzed the maintenance of large Cobol projects, and found the optimal procedure size to be 44 statements. McCabe suggested that functions with a cyclomatic complexity above 10 should be refactored, as otherwise they may be too complex to test effectively (McCabe, 1976). Fenton and Neil argue against the “Goldilocks conjecture”, that there is an ideal size that is not too small and not too large, but this was about module size, not individual function sizes (Fenton & Neil, 1999).

Several studies have found that code complexity is indeed related to scope (Banker et al., 1993), and some have claimed that this is such a strong correlation that length is the only metric that is needed (Herraiz & Hassan, 2011; Gil & Lalouche, 2017). Landman et al., in contradistinction, claim that the correlation between code complexity and lines of code is not as high as previously thought (Landman, Serebrenik, Bouwers, & Vinju, 2016). Their view is that high correlations are the result of aggregation, and if individual functions are considered then complexity metrics provide additional information to length. This echoes earlier work by Gill and Kemerer, who propose to normalize complexity metrics by length (Gill & Kemerer, 1991). Similar disagreements have been voiced about the size of modules and classes (Fenton & Neil, 1999). El Emam et al. also argue against the claim that small components are necessarily beneficial, and fail to find evidence for a threshold beyond which large classes cause more defects (El Emam et al., 2002).

Refactoring is used to improve the structure of code without changing its functionality (Fowler, 2019), and reflects a perception that the code is “dirty” and its quality can be improved (Allman, 2012; Tom, Aurum, & Vidgen, 2013; Zabardast, Gonzalez-Huerta, & Šmite, 2020). One of the most basic actions performed during refactoring is function extraction, that is turning a block of code into a separate function (Murphy-Hill, Parnin, & Black, 2012; Tsantalis & Chatzigeorgiou, 2011; Tsantalis, Chaikalas, & Chatzigeorgiou, 2018; Hora & Robbes, 2020; Liu & Liu, 2016). There has been extensive research on tools to perform function extraction semi-automatically (Maruyama, 2001; Komondoor & Horwitz, 2000, 2003; Ettinger, 2007; Haas & Hummel, 2016). These are usually based on the identification of *program slices* — subprograms responsible for the computation of the value of some variable (Weiser, 1984). Data collected on using refactoring tools indicates that function extraction is one of the most common types of refactoring. Half of all refactoring done with JDeodorant were function extractions (Tsantalis et al., 2018). An analysis of more than 400 thousand refactorings by Hora and Robbes found that 17% were method extraction (Hora & Robbes, 2020). But we know of no research about how developers actually perform function extraction, especially when not using tools. Our experiment is designed to explore this issue.

3. Research Questions

We are motivated by very basic questions in programming, and specifically by the question of function length. For example, how does function length interact with testing, with the propensity for having defects, and with understanding the whole system? These questions are interesting and important, but complicated, because myriad factors and considerations may affect the design of code. They are also hard to study experimentally, because if we give developers a non-trivial programming assignment, which includes enough logic to get different designs, they can come up with very different designs that are hard to compare (and in fact, multiple version are also possible with very simple assignments (van der Meulen & Revilla, 2008)). In addition, the answer may be ill-defined, as it may depend on the specific developers involved in writing and maintaining the code, who could be different from each other and from other developers.

To make some progress we therefore start with much more limited and concrete questions. We focus on function extraction rather than the whole issue of writing functions. Within this context we ask

1. How do developers extract functions?
2. Do developers agree regarding how to refactor code by extracting functions?
3. What are the considerations that developers apply when deciding about functions?
4. Do developers conform with the recommendation in *Clean Code* to use very short functions?

To answer these questions we start from a given code-base, and conduct an experiment in which developers are asked to refactor this code. As they all start from the same base, it is then straight-forward to compare their results. We then also ask them to answer a short survey about their considerations.

4. Methodology

Our experiment was based on real production code. This code was modified to make it monolithic by inlining all the functions. The experiment participants then refactored it, and we analyzed the resulting codes.

4.1. Code Selection

To make the experiment as realistic and relevant as possible we decided to use real production code as the basis. The main consideration was that it be a general utility that can be understood with reasonable effort. It also had to be long enough, and have enough logic, so that it would be reasonable to expect some variations in the results. In other words, we needed to avoid code that could be arranged in only one reasonable way.

The code we chose is from file `adapters.py` in the Requests http Python library¹. This module allows to send HTTP/1.1 requests easily. The code was originally 534 lines long including comments and blank lines. Excluding blank lines leaves 434 lines, and also excluding comments leaves only 286 lines of actual code. After we found a suitable code, we performed a pilot and sent the experiment to 4 developers. The results we received indicated that the experiment works.

4.2. Code Flattening

We call the procedure of converting the code from many functions to one function “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. It included the main logic and calls all the smaller functions.

¹<https://github.com/psf/requests/blob/master/requests/adapters.py>

2. Every function which was only defined, but not called internally, was removed. These were functions which were only called from other classes. We decided to remove them since it would be unreasonable to include unused code in our monolithic function.
3. The remaining functions were inlined by replacing calls to them with the function's code. The names of each function's parameters were replaced with the names of the arguments the function was called with. In one case we also needed to change a function name that was the same as that of a function from a different module (changed `send` to `send_the_request`).
4. All unrelated classes and imports 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 and cluttered.
5. All the comments in the original code which were added by its developers to describe some flow in the code, and the main function definition, were left as they are. We wanted the participants to experience the code as close to the original as possible. But the header comments of the inlined functions had to be removed, since there was no suitable place for them.
6. All calls to third-party functions remained as they are. We wanted the participants to refactor the code in a specific context, and not in such large context as all the Python language. So we inlined only the project's functions, and left calls to functions from other sources.

All these changes reduced the total length of the code to 298 lines including blank lines and comments, and 208 lines of actual code. This is the code the participants received to refactor.

4.3. Experiment Execution

The experiment was created with Google Forms. However, Google Forms does not allow to upload files anonymously (it requires to login to upload, and the email of the uploader is included in the results). As we wanted to avoid collecting any personally identifying information, we used the services of Formfacade, which integrates with Google Forms and allows to add various functionalities including anonymous file upload. The form contained an introduction and 3 sections as follows:

1. Initially we informed the participants that the experiment and the questions are not mandatory, they can quit the experiment whenever they want, and that no personal information is collected — the experiment is completely anonymous. The introduction page also informed them that by moving to the next page they agree to participate under these conditions.
2. After the introduction participants were asked questions related to their background, such as age, programming education, development experience, etc.
3. For the experiment itself they got our flattened code as a link to GitHub, and were requested to download it. They could work on extracting the functions in their preferred IDE and work environment, with no time limit. After they finished refactoring it, they needed to make a zip file of their code and upload it.
4. Finally, the participants were asked to complete a short survey with questions related to the code assignment they were requested to do. One question in the survey was about what they thought was the ideal length for functions. In addition we asked them to rate different considerations that may apply to function extraction refactoring.

The link to the website we created was sent to personal contacts with familiarity with programming. In addition, the experiment was published in two Reddit forums, [r/SoftwareEngineering](#) and [r/Code](#). We wanted the participants to be as varied as possible.

We received a total of 34 responses to the experiment. Of these only 23 submitted the refactored code, which is the most important part; the 11 others were removed from the analysis. The background of the

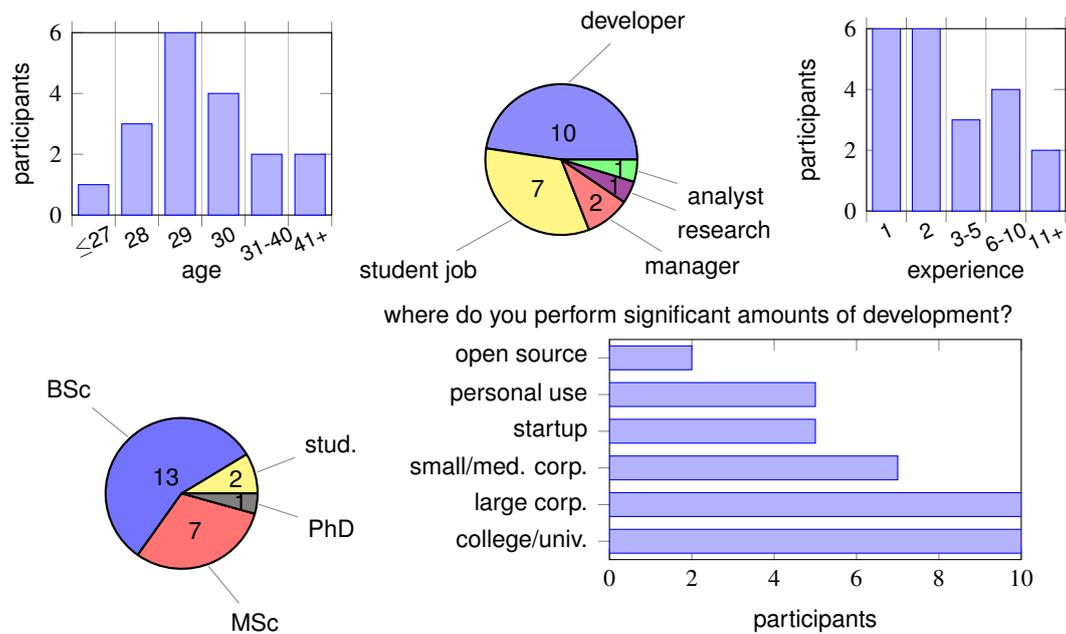


Figure 1 – Participants demographics.

participants was as follows (Figure 1). Their ages were rather focused, with 72% being between 28 and 30 years old; the oldest was 49. 51% had a BSc/BA in a computer-related field, and another 30% had an MSc; only 9% were students who had not completed their first degree yet. 91% reported that they were employed. The range of development experience in industry, that is excluding studies, was from 1 to 29 years, but for 57% it was up to 2 years. 48% were employed as developers, 33% had student jobs, and 10% held management positions. Most of the participants perform significant amounts of development in industry, in large corporations, small/ medium companies, or startups (in decreasing order). Many also code in academic institutions or for personal use. The most common programming language they felt comfortable with was Python (64%), followed by Java (50%) and C (36%). Finally, 44% reported that they use agile development practices.

4.4. Code Analysis

As noted above the code comprised 208 lines of actual code. After refactoring it was a bit longer, due to the added function headers and function calls. Analyzing this amount of code is hard and error prone. We therefore wrote a set of scripts to analyze it. Most of the analysis was done with regex matchers.

Code Stripping The first 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 presented, but not the content. The script works in the following way:

1. Remove all the import statements.
2. Remove all the comment lines, and comment blocks demarcated with `"""` on the first and last line.
3. Remove all blank lines.
4. If a statement was divided into multiple lines, they were joined into one line. Specifically, if there is an opening bracket, `(`, but no closing one, `)`, all the following lines were concatenated to the initial line until the line with the closing bracket.

5. Remove all the white space at the beginning and at the end of each line. Since Python is a language where indentation is part of the syntax, this step must be done last. Its purpose is to make the comparison between the lines independent of the exact indentation used.

Lines Identification The code submitted by the experiment participants needs to be compared to the original code. This comparison was based on comparing individual lines. For instance, to identify which part of the participant's code was originally in a function, we needed to compare each line in the participant's code to the original code. At first, we compared the lines just by using the Python "==" operator after we cleaned the white spaces, but it was not good enough. For example, where there were many false negatives where the participants changed a variable name, changed the argument name the function receives, or just added an underscore.

Because of these problems, we switched to comparing with SequenceMatcher from the difflib Python library. This provides a similarity ratio between 0 and 1. We could then choose a threshold to identify lines with small changes as equal, and lines with more changes as different. The threshold we used for identifying lines extracted into functions was 0.95. The threshold for noting lines that were in the original code but were not found in the results and were probably changed was 0.75 (these are the lines denoted by black dots in Figure 2 below).

Functions Identification To compare code refactoring by extracting functions we need to identify the functions in the results. This was done by a script which counts the indentation and looks for the def keyword which defines the start of a function. When a def is found, the string after it up to the character '(' is taken as the name of the function. The body of the function is all following lines with a larger indentation.

Once all the lines in a function had been identified, we retrieve from the original code the range of lines that the function was extracted from. First the line which starts the function is checked for its line number in the original code. If the line is not found 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 using the same process. If the line is not found in the original code, the previous line is checked until a line appears in the original code. The range is then determined by these lines, and we verified that it matches the length of the function that was extracted. Overall there were a total of 61 lines that were skipped in this way because they were not found in the original code. Most of these (57) were the return lines which ended the extracted 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 not identified due to extreme changes that were made by the participants, or were completely new lines they decided to add.

We also checked if the lines' order was changed relative to the original code. This is done by running over all the functions and for each function checking 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. In all the results we had only one case where the order of 2 lines was changed.

Manual Analysis A manual scanning of the results was also done. We did this 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 were the participants' main considerations for the refactoring.

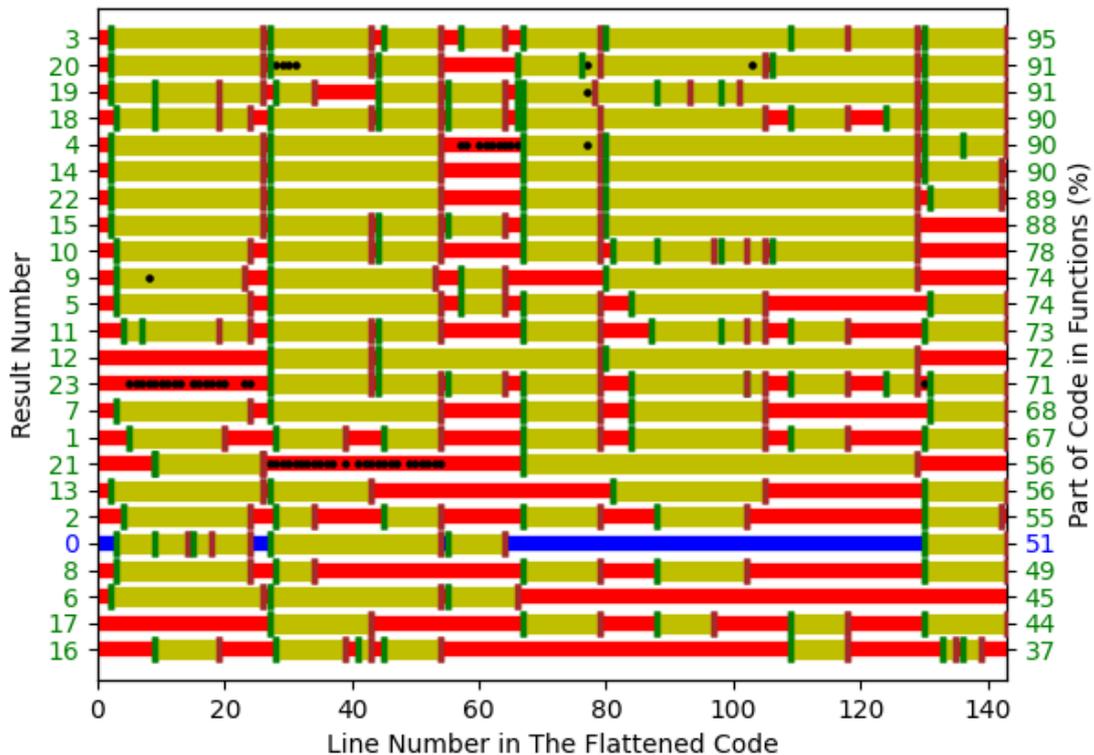


Figure 2 – Visualization of extracted functions locations relative to the flattened code. The horizontal dimension represents location in the flattened code, with the yellow-olive segments showing lines of code that were extracted to functions. Green and brown lines indicate function start and end, respectively. Each horizontal bar represents the refactoring performed by one participant. Result 0 (in blue) is the original code before flattening.

5. Results and Discussion

5.1. Function Extraction Results

The participants in the study were requested to provide us with two inputs: the refactored code, and answers to a set of questions. We begin with the code analysis.

We analyzed the coding task results both manually and automatically with scripts as described above. Figure 2 shows a bird’s-eye view of all the function extractions by all the participants, compared with the original code. The horizontal dimension represents the lines of code in the original flattened code, after excluding all the blank and comment lines. The range is from the first line at the left to the last line, line 143, at the right. This is less than the original 208 lines of actual code because in the original layout arguments to functions were spread over multiple lines, one each, for readability. As we noted above, in the analysis we unified such argument lists in a single line.

Each horizontal bar represents one experimental subject. The red segments are lines of code that were left in the main function. The yellow-olive parts are ranges that were extracted to functions. The numbers on the left are the participants’ serial numbers in the results data. They are sorted by the fraction of the code that they extracted into functions, as noted on the right-hand side. The blue line, with serial number 0, is the original code before it was flattened. The black dots represent lines in the original flattened code that did not appear in the results, meaning these lines were deleted, replaced by new lines, or changed too drastically to allow the comparator to find them.

Looking at the figure, we find that *no two participants produced exactly the same refactoring*. However, it is immediately obvious that function extraction is often correlated across many participants. Starting to analyze this data, we find that all the participants extracted between 3 and 10 functions, with just

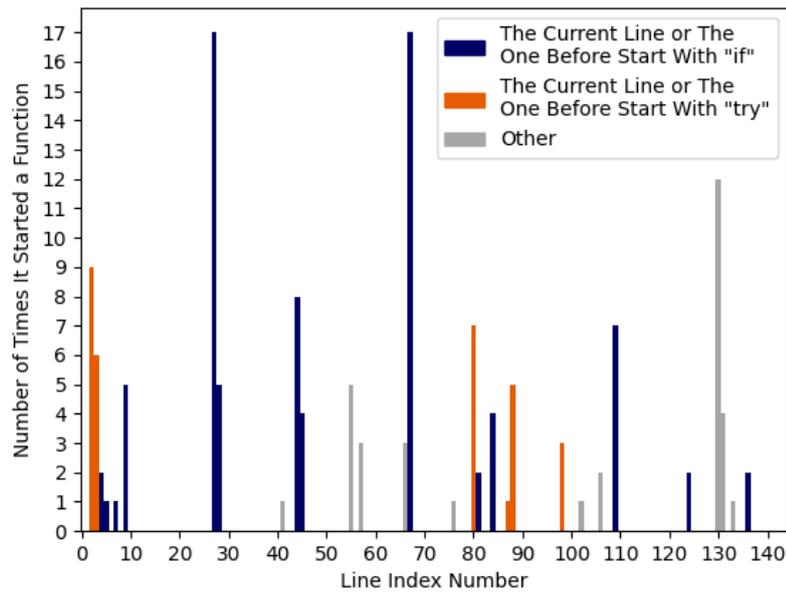


Figure 3 – Lines that start functions by type.

over half extracting 5–7 functions. (The original code had 6 functions.) In many cases the end of one function is closely followed by the beginning of another. But in many other cases functions are separated by blocks of code that remained in the main function. Notably the line at the beginning of the whole code was always left out, which is expected: it is the `def` of the main function.

The lines that start functions are shown in Figure 3, colored by the code construct which appears in that line (or adjacent to it): `if` in dark blue, `try` in orange, and other in gray. As we can see the vast majority of functions start on an `if` (77 of the 142 functions defined by the experiment participants) or `try` (31 functions). In addition, near the end of the code there is a block where the response object is created, and many of the participants extracted this block of code to a function. These are the functions that start on line 130 (including the creation of the object itself) or line 131 (only the population of the object’s data members).

Our conclusion is that the main triggers for extracting functions in our experiment are coding constructs. This could be related to the Python syntax, which requires such constructs to be indented. The start of a new block of indented code may then be taken as an indication to extract a function. Notably, our results resonate with the recommendations made by Martin in *Clean Code* (Martin, 2009): that the body of an `if` should be one line long and specifically a function call, and that `try-catch` blocks be extracted to separate functionality from error handling.

Figure 4 shows the lengths of the functions the participants chose to extract from the flattened code (all the functions extracted by all the participants, but not the main function). As we can see in the figure, 80% of the extracted functions had 5–30 lines of code, and 55% of the functions had a length of 10–14 lines. This result is discussed below in connection to the survey question about the ideal function length.

Returning to Figure 2, we can see the majority of the participants extracted at least 70% of the code to functions. The reason may be a general understanding that code should be divided into functions and not be structured as one big main function. But on the other hand there might be a bias due to the definition of the experiment. The experiment instructions were to extract functions, and this might cause the participants to perform more function extractions than they would otherwise. In addition, if we compare the results to the original code, the original code had only 51% of the code in functions, placing it fifth from the end in this parameter. This reinforces the conjecture that there may be a bias because of the experiment definition.

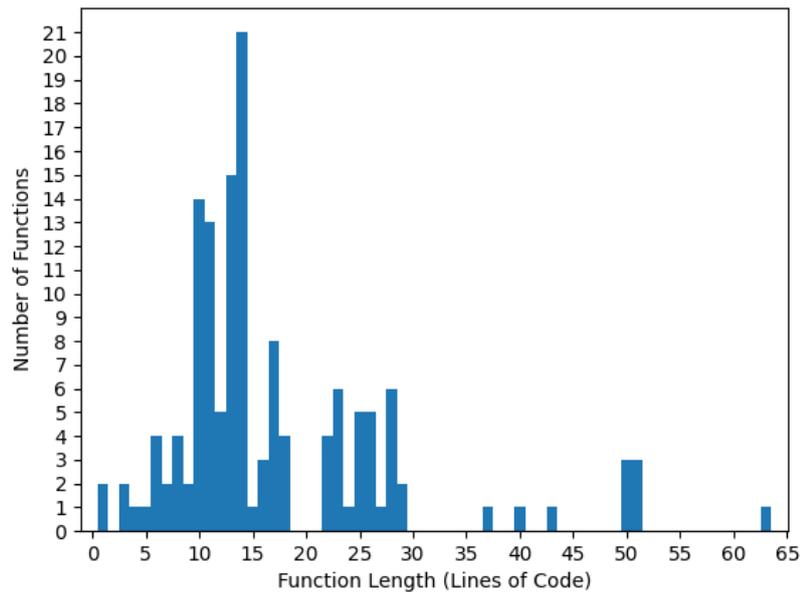


Figure 4 – Histogram of function lengths.

A very important observation is the diversity in the fraction of code extracted to functions. This ranges from a low value of extracting only 37% of the code lines, leaving nearly two thirds of the code in the main function, to a high mark of extracting no less than 95% of the code, leaving nearly nothing in the main function. And between these extremes we see a wide distribution of values. This testifies to differences of opinion between the experiment participants regarding what parts of the code should be extracted to functions.

To get a better picture of these differences, we identify 6 blocks of code that were the main candidates for extraction:

1. Lines 3 to 26, extracted (at least partially) by 20 participants and in the original code. This is a big try block surrounding the creation of the connection for sending the request.
2. Lines 27 to 54, extracted (at least partially) by 22 participants and in the original code. This verifies the SSL certificate, and contains two big if blocks, which were extracted together or separately.
3. Lines 55 to 64, extracted by only 9 participants, but also in the original code. This requests the URL and contains a smaller if.
4. Lines 65 to 79, extracted by 19 participants, but not in the original code. This is a large if block that checks whether a timeout is needed. It was nearly always extracted in exactly the same way.
5. Lines 80 to 129, extracted (at least partially) by 22 participants but not in the original code. This large block of code is where the actual sending is done and the response received. But these operations may fail for myriad reasons, so they are surrounded by three nested try-except blocks and error handling code. There was especially high variability concerning what parts of this to extract, reflecting different opinions regarding whether to extract the full largest try-except block as one large function, or to extract only some internal parts from it.
6. Lines 130 to 143, extracted by 17 participants and in the original code. This is code that creates the object to return, and it was nearly always extracted in exactly the same way.

Interestingly, three of these code blocks (numbers 1, 2, and 6) correspond to extraction types identified by Hora and Robbes as the most common, which are for creation, validation, and setup (Hora & Robbes, 2020).

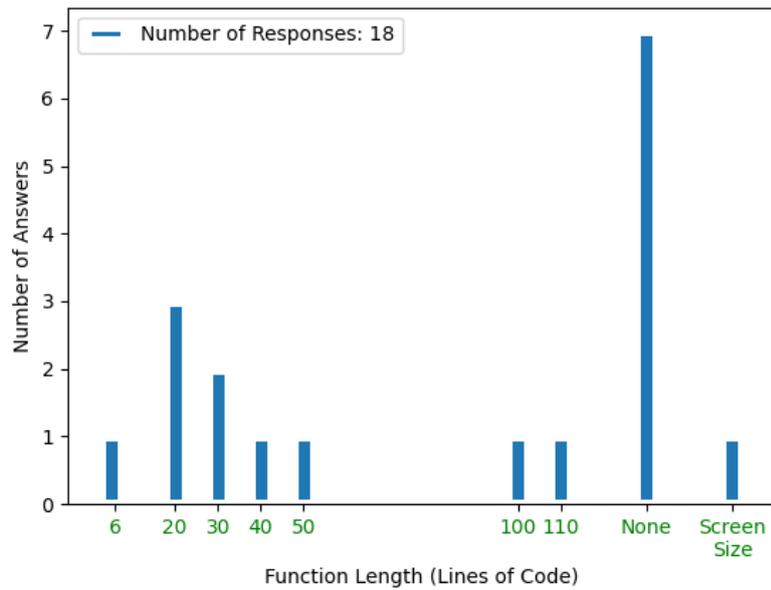


Figure 5 – Responses to the question concerning the ideal length of functions.

To summarize, the variability in the amount of code extracted to functions stems from different opinions on whether certain blocks should be extracted, whether to include or exclude nested constructs, and whether to include or exclude a surrounding construct (e.g. include the if itself or only its body).

Note that part of the diversity concerns try-except blocks and error handling code. This reflects the fact that indeed there are different ways and designs to handle errors. Some believe that error handling logic should be located in conjunction with the functions which suffered the errors, while others believe that it should be separated such that all errors are handled together. It is worthwhile to mention that in the original code this error handling code was not extracted to a dedicated function.

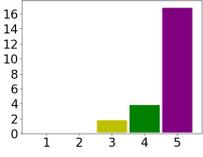
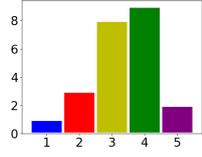
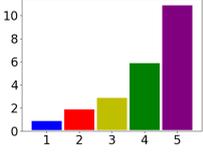
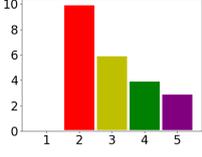
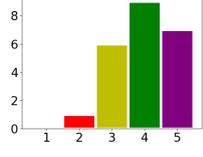
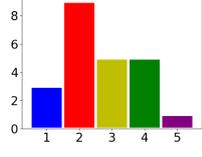
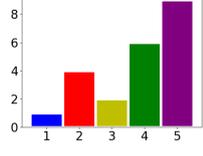
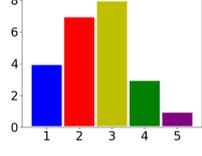
5.2. Survey Results

After the code assignment the participants were asked questions about the considerations involved in function extraction. We started with asking them about the “clean code” approach to code development. 61% of the participants indicated that they were familiar with this approach. And of those that were familiar, 93% said they agreed with the principles of clean code and strive to act on it (answers of 4 or 5 on a scale of 1 to 5).

The next question concerned the ideal function length. The responses we received are shown in Figure 5. As we can see the most common answer was that there is no ideal length of function, but from those who did provide a number most of the responses were 20–30 lines of code. Comparing this to the results of the actual functions they extracted, shown above in Figure 4, we find that the actual functions were even shorter than the ideal length the participants stated they believe in. Interestingly, the actual results agree with the recommendation that functions should usually have no more than 20 lines, suggested by Martin in *Clean Code* (Martin, 2009, p. 34).

We then asked about the participants’ main considerations when extracting new functions. The distribution of answers for the different considerations we suggested are shown in Table 1. The scale was from 1 = Not important to 5 = Very important. As we can see, the most popular consideration was “Making each function’s logic cohesive” with an average score of 4.62/5. The second most popular was “Making each function do only one thing”, with an average score of 4.10/5. The consideration that corresponds to our observation in Figure 3 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 considerations by looking at the function extraction results, because “Making each function’s logic

Table 1 – Main considerations for function extraction

Consideration	Mean	Histogram	Consideration	Mean	Histogram
Making each function’s logic cohesive	4.62		Following design patterns	3.41	
Making each function do only one thing	4.10		Not needing to pass too many arguments	3.10	
Separating code control blocks	3.93		Making functions as short as possible	2.72	
Making functions easily testable	3.89		Making each function close to the ideal length	2.65	

cohesive” and “Making each function do only one thing” are quite an abstract considerations, and incomparable to the “Separating code control blocks” consideration which can be measured by counting the lines that started a function on a new block of code. Interestingly, the two considerations concerning function length, “Making functions as short as possible” and “Making each function close to the ideal length”, were ranked as having lower importance than all other considerations.

As this questionnaire appeared at the end of the experiment, after the participants had submitted their refactored code, we also asked whether seeing the survey questions might have changed the way they refactored the code. 48% answered yes, possibly implying that they did not think about what they had done deeply enough. The main consideration cited in this context was to make the extracted functions more testable; this was mentioned by 4 participants. Several issues were mentioned by two each: using design patterns, the number of arguments, separation into classes or files, and the length of the functions.

5.3. Discussion

One of our main results is that participants in the experiment tend to extract functions based on the blocks of code defined by existing coding constructs, such as if and try. They also gave the consideration of “Separating code control blocks” relatively high scores in the survey. Using blocks of code as cues for function extraction is reasonable because such blocks indeed encapsulate separate sections of the computation. However, we note that extracting functions by looking at blocks of code is also the easiest way to extract. It doesn’t even require the participants to understand the code’s logic.

Another interesting observation is the difference between the approaches taken by tools and by human developers. As we noted above in the related work section, tools are often based on extracting program slices (e.g. (Maruyama, 2001; Tsantalis & Chatzigeorgiou, 2011)). Program slices are defined by the data flow in the program: they include instructions that may affect the value of a certain variable. This is useful for code analysis, and to ensure that the extracted code is self-contained and independent, so the program will retain exactly the same behavior after it is extracted. But for humans, data flow is not easy to follow in programs written in conventional procedural languages (C, Java, Python). Therefore

the participants in our experiment preferred to extract functions based on easily-observable coding constructs, which reflect control flow. It may be interesting to contrast these approaches in terms of the similarities and differences between the refactored codes they produce. We leave such a study to future work. In the meanwhile, we note that our results resonate with those of Pennington from 25 years ago, who found that professional programmers base their mental representations of programs on control flow, and not on functionality (Pennington, 1987).

An interesting issue is the question of ideal function length. As we noted, Martin famously makes the case that functions should be as short as possible (Martin, 2009, p. 34). Given that 61% of our participants acknowledged that they are familiar with the “clean code” discipline, this might be the reason that 80% of the functions were 10-30 lines long, and 55% had 10-14 lines. However, in the survey we saw that “Making functions as short as possible” and “Making each function close to the ideal length” were the two least important considerations for function extraction (Table 1). Thus the participants believe that small functions and ideal functions length are not as important as at least 6 other considerations we suggested to them. But in fact they do extract small functions. This is quite an interesting result, because it means the participants extracted small functions without explicitly thinking about them in this way. We might carefully conjecture that nowadays developers extract small function as part of their coding skills. Another possibility is 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.

6. Threats to Validity

As in any experiment, we had some difficulties and threats to the validity of our results. The following threats are the most significant remaining ones.

6.1. Methodological Difficulty

The problem of studying function creation is a difficult one. Ideally we would like to observe and compare developers creating functions when programming to the same specification. However, the diversity of programs that can be expected is huge, including in how exactly the problem is solved and how the solution is divided into functions. Nevertheless, analyzing such datasets (e.g. (van der Meulen & Revilla, 2008)) is an interesting direction for future work.

Another possible option is to start with “born-flat” code, that was written without being divided into functions, and ask that it be refactored. This may be extremely hard, because in such code the logic for different functionalities may be intermixed, and global variables may be used to communicate state. In such a situation splitting the code into functions is not representative of writing new functions. We used flattened code that was derived from production code including functions. This suffers from the drawback that the signature of the original functions may be observable in the code, and may have contributed to the correlation between the refactorings performed by different participants. However, we note that there was also significant agreement where the original code had *not* been divided into functions (lines 65–129), supporting our conclusion that structural cues may be at work.

6.2. Confounding Factors

Our experiment was explicitly about extracting functions. This may have affected the participants, who may have tried to “live up to expectations” and extracted more than they would have under real work conditions. Such behavior, if it exists, would lead to a bias in the results.

Another possible effect of an experiment environment is that it is perceived as less serious than “real” work, leading participants to invest less effort. This was noted in a few comments we received to the postings inviting participants on reddit. One wrote “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 code”. Another wrote “this is a 150 lines function, there were occasions where reformatting something like that was a task for the whole day in my job”. It is also supported

by the fact that nearly half the participants indicated they might have changed the way they refactored the code after seeing the survey questions. Finally, the three cases of stretches of lines that were not identified (sequences of black dots for participants 4, 21, and 23 in Figure 2) were actually lines that were deleted with the probable intention of turning them into a function, but this was then not done — which again might indicate lack of seriousness.

It is not clear how these problems could be avoided in an experiment. A possible approach is to conduct a much larger experiment, in both scope and cost, in which developers are actually hired to generally refactor a large body of code (Sjøberg et al., 2002). Such an experiment can be attempted based on our experiment’s demonstration that the methodology of using flattened production code looks promising.

6.3. Limited Generalizability

Being an initial experiment in a new direction, and given the need to develop a new methodology, our experiment was rather limited. We used only one function, which raises the question of how representative this is of other codes. For example, one of the main reasons for creating functions is to reuse code, so if the code contains clones they may be prime targets for extracting. However, our code did not contain any clones, so this was not checked. In addition, our results are limited to the context of refactoring, and may not generalize to function creation in the context of writing new code.

We also had only 23 participants who submitted the refactored code. Given the need to invest around half an hour in this non-trivial task this is not a bad response. However, it is not enough to enable an analysis of the behavior of different demographic groups, as each one would have too few members. Another issue is whether our participants are representative of developers in general. For example, in our experiment only 61% of the participants indicated they know clean code, while in a survey by Ljung et al. the vast majority of the participants had heard of clean code, and tended to agree with its principles (Ljung & Gonzalez-Huerta, 2022). Again, based on our initial experiment, a larger one can be attempted.

7. Conclusions

7.1. Summary of Results

In this study we tried to make initial observations of how developers create functions, including issues like how much they agree with each other and how long are the functions they create. Our approach was to design an experiment, where we took real production code, “flattened” it, and then asked the experiment participants to refactor it by extracting functions. In addition, we asked them to answer some questions related to their considerations when extracting the functions.

Analyzing the results we found that most of the participants extracted small functions, as suggested in the “Clean Code” approach, and that the main cues for extracting functions were related to the code structure. Interestingly the actual functions were somewhat shorter than what the participants said would be ideal. Also, they didn’t explicitly consider length to be a major factor, citing logical cohesion and doing only one thing as the most important considerations.

We also found that in most cases the fraction of the code that was extracted to functions was bigger than it had been in the original code. This was probably due to the fact we asked participants explicitly to extract functions. But it also implies that developers can be influenced in how they go about designing functions. This can be studied in future experiments.

We believe that in addition to these concrete results, we produced some promising methods to study such complicated issues like software design. Since this kind of experiments are almost non-existent, we hope our study and our methods will encourage additional experiments in this field. In particular, a desirable follow-up experiment is to replicate our experiment at a larger scale, and then study the quality of the refactored code produced by the experiment participants (in terms of readability, testability, etc.). It would be especially interesting to see if there is any correlation between the quality of the produced code and the considerations or approaches employed in the refactoring.

7.2. Implications

Our work has several possible implications for practitioners. The most immediate is to highlight a simple and accessible approach to function extraction. Extracting functions goes hand in hand with the teachings of procedural programming, which perhaps explains why it is one of the most commonly used types of refactoring. And by focusing on code blocks defined by constructs like loops, conditionals, or exception handling one can perform function extraction in a straightforward manner.

Extracting such blocks also provides a simple methodology to control the desired granularity, as one can continue to extract nested blocks down to the most basic blocks, or alternatively stop at a larger scope. For example, when extracting an if construct to a separate function, one can then decide whether to also extract the “then” and “else” blocks or not. If one desires to create very short functions, extracting blocks is an easy way to achieve this.

Using code blocks for function extraction is also related to program comprehension. When faced with unknown code, understanding what it does requires a bottom-up approach (Shneiderman & Mayer, 1979). This is greatly simplified if each function is short, and composed mainly of calls to other functions which have good descriptive names, as such a structure allows the readers to perceive the code at a higher level of abstraction rather than forcing them to contend with the basic constructs. Extracting blocks achieves exactly this effect.

8. Experimental Materials

Experimental materials are available at

<https://doi.org/10.5281/zenodo.7044101>

9. Acknowledgments

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).

10. References

- Allman, E. (2012, May). Managing technical debt. *Comm. ACM*, 55(5), 50–55. doi: 10.1145/2160718.2160733
- Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993, Nov). Software complexity and maintenance costs. *Comm. ACM*, 36(11), 81–94. doi: 10.1145/163359.163375
- Clausen, C. (2021). *Five lines of code*. Manning Publications.
- El Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., & Rai, S. N. (2002, May). The optimal class size for object-oriented software. *IEEE Trans. Softw. Eng.*, 28(5), 494–509. doi: 10.1109/TSE.2002.1000452
- Ettinger, R. (2007, Oct). Refactoring via program slicing and sliding. In *Intl. Conf. Softw. Maintenance* (pp. 505–506). doi: 10.1109/ICSM.2007.4362672
- Fenton, N. E., & Neil, M. (1999, Sep/Oct). A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5), 675–689. doi: 10.1109/32.815326
- Fowler, M. (2019). *Refactoring: Improving the design of existing code* (2nd ed.). Pearson Education, Inc.
- Gil, Y., & Lalouche, G. (2017, Oct). On the correlation between size and metric validity. *Empirical Softw. Eng.*, 22(5), 2585–2611. doi: 10.1007/s10664-017-9513-5
- Gill, G. K., & Kemerer, C. F. (1991, Dec). Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Softw. Eng.*, 17(12), 1284–1288. doi: 10.1109/32.106988
- Haas, R., & Hummel, B. (2016, Jan). Deriving extract method refactoring suggestions for long methods. In *Intl. conf. softw. quality* (pp. 144–155). doi: 10.1007/978-3-319-27033-3_10
- Herraiz, I., & Hassan, A. E. (2011). Beyond lines of code: Do we need more complexity metrics? In A. Oram & G. Wilson (Eds.), *Making software: What really works, and why we believe it* (pp. 125–141). O’Reilly Media Inc.

- Hora, A., & Robbes, R. (2020, May). Characteristics of method extraction in Java: A large scale empirical study. *Empirical Softw. Eng.*, 25(3), 1798–1833. doi: 10.1007/s10664-020-09809-8
- Komondoor, R., & Horwitz, S. (2000, Jan). Semantic-preserving procedure extraction. In *Ann. Symp. Principles of Programming Languages* (pp. 155–169). doi: 10.1145/325694.325713
- Komondoor, R., & Horwitz, S. (2003, May). Effective, automatic procedure extraction. In *IEEE Intl. Workshop Program Comprehension*. doi: 10.1109/WPC.2003.1199187
- Landman, D., Serebrenik, A., Bouwers, E., & Vinju, J. J. (2016, Jul). Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *J. Softw.: Evolution & Process*, 28(7), 589–618. doi: 10.1002/smr.1760
- Liu, W., & Liu, H. (2016, Aug). Major motivations for extract method refactorings: Analysis based on interviews and change histories. *Frontiers Comput. Sci.*, 10(4), 644–656. doi: 10.1007/s11704-016-5131-4
- Ljung, K., & Gonzalez-Huerta, J. (2022, Nov). “to clean-code or not to clean-code” a survey among practitioners. In *Intl. conf. product-focused softw. process improvement* (pp. 298–315). (Lect. Notes Comput. Sci. vol. 13709) doi: 10.1007/978-3-031-21388-5_21
- Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.
- Maruyama, K. (2001, May). Automated method-extraction refactoring by using block-based slicing. In *Symp. softw. reusability* (pp. 31–40). doi: 10.1145/375212.375233
- McCabe, T. (1976, Dec). A complexity measure. *IEEE Trans. Softw. Eng.*, SE-2(4), 308–320. doi: 10.1109/TSE.1976.233837
- McConnell, S. (2004). *Code complete* (2nd ed.). Microsoft Press.
- Murphy-Hill, E., Parnin, C., & Black, A. P. (2012, Jan/Feb). How we refactor, and how we know it. *IEEE Trans. Softw. Eng.*, 38(1), 5–18. doi: 10.1109/TSE.2011.41
- Pennington, N. (1987, Jul). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3), 295–341. doi: 10.1016/0010-0285(87)90007-7
- Shneiderman, B., & Mayer, R. (1979, Jun). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *Intl. J. Comput. & Inf. Syst.*, 8(3), 219–238. doi: 10.1007/BF00977789
- Sjøberg, D. I. K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., ... Vokác, M. (2002, Oct). Conducting realistic experiments in software engineering. In *Intl. symp. empirical softw. eng.* (pp. 17–26). doi: 10.1109/ISESE.2002.1166921
- Tom, E., Aurum, A., & Vidgen, R. (2013, Jun). An exploration of technical debt. *J. Syst. & Softw.*, 86(6), 1498–1516. doi: 10.1016/j.jss.2012.12.052
- Tsantalis, N., Chaikalis, T., & Chatzigeorgiou, A. (2018, Mar). Ten years of JDeodorant: Lessons learned from the hunt for smells. In *Intl. conf. softw. analysis, evolution, & reengineering* (pp. 4–14). doi: 10.1109/SANER.2018.8330192
- Tsantalis, N., & Chatzigeorgiou, A. (2011, Oct). Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. & Softw.*, 84(10), 1757–1782. doi: 10.1016/j.jss.2011.05.016
- van der Meulen, M. J. P., & Revilla, M. A. (2008, Nov/Dec). The effectiveness of software diversity in a large population of programs. *IEEE Trans. Softw. Eng.*, 34(6), 753–764. doi: 10.1109/TSE.2008.70
- Weiser, M. (1984, Jul). Program slicing. *IEEE Trans. Softw. Eng.*, SE-10(4), 352–357. doi: 10.1109/TSE.1984.5010248
- Zabardast, E., Gonzalez-Huerta, J., & Šmite, D. (2020, Aug). Refactoring, bug fixing, and new development effect on technical debt: An industrial case study. In *Euromicro conf. softw. eng. & advanced apps.* (pp. 376–384). doi: 10.1109/SEAA51224.2020.00068