“What’s in a chunk?”: A Recall Study of Quantitative Chunking in Source Code

Author: Nicole L. de Groot[a]

[a] Student number: 1564641

[a] Leiden Institute of Advanced Computer Science (LIACS), Leiden Universiteit, Leiden, The Netherlands

[a] n.l.de.groot.2@vuw.leidenuniv.nl

Thesis supervisor: Felienne F.J. Hermans[b]
Thesis second supervisor: Ross Towns[c]

[b,c] Leiden Institute of Advanced Computer Science (LIACS), Leiden Universiteit, Leiden, The Netherlands

"What's in a chunk?": A Recall Study of Quantitative Chunking in Source Code

Nicole L. de Groot

Abstract

Chunking is "the recoding of smaller units of information into larger, familiar chunks" (Thalmann et al., 2019). Several chunking studies have been done with chess (De Groot, 1946), language (Simon, 1974), and source code (McKeithen et al., 1981). These studies proved that experts have more, differently organized knowledge than beginners. McKeithen et al. conducted a recall study with code that found that beginners hold a broad variety of smaller chunks containing more natural-language elements than experts. However, as this setup and its findings are outdated, new knowledge can be gained from a conceptual replication study. In the present study, we broaden the theoretical framework, and recreate a short-term recall study inspired by McKeithen et al. in a modern setting. Using text analysis techniques, like comparative dictionaries and n-grams, we clarify the use of chunking in subjects' recalled Java code. Using such techniques we hope to broaden the knowledge on the use of text analysis techniques in the field of computer science. These can show differences between the different skill level groups within the sample. With this, we try to answer the following questions: How do recall and chunking of a Java snippet differ between skill levels (beginner, intermediate, expert), and between a normal and a scrambled version? And, How can text analysis techniques be implemented on recalled source code of subjects and what can be inferred from the results? An online questionnaire is used which includes the recollection of two versions of a Java snippet (normal versus scrambled). For recollection, we use an embedded Ace editor to simulate a natural coding environment. Analyses are done by doing quantitative analysis on several variables that summarize the recall of each subject (e.g. length of answer, amount of correct concepts used, and relative overlap between subject answer and solution). Statistical analyses proved that version or expertise have a marginally significant effect on recall of a Java snippet. More research is necessary to determine the use of text analysis techniques on code that is written under a time limit and thus may contain many typos and other errors that might not occur otherwise. However, text analysis techniques show potential when techniques can be combined in the future to create a more complete analysis pack for source code.

Keywords: chunking; expertise; cognitive load; replication; text analyses

1 Introduction

1.1 Crossing Borders Between Disciplines

The field of information processing (IP) stretches across multiple disciplines, all with different angles. However, within the discipline of computer science, IP is quite young, and therefore, the research on the subject is scarce. While IP within the discipline of cognitive psychology has its roots in the 1950s (e.g., De Groot, 1946; Miller, 1956; Atkinson & Shiffrin, 1968), the computer science version of IP, source code understanding and analysis, have only been granted more notice since the 2000s, especially in academic research. This is because the creation of the first and second computers in 1946 boosted interest in the development of the new technologies and their programming languages (PLs), but less so in the understanding and processing of those in the human brain.

However, around 30 years later, after an impressive range of PLs had already been created, scientists started researching the understanding of these programs and languages (e.g., Shneiderman & Mayer, 1979; Brooks, 1983; Pennington, 1987; Siegmund et al., 2020). Because this field of program comprehension was still so young, it had to borrow research techniques from other disciplines, namely cognitive psychology and natural language processing: two fields that were both starting to gain more ground and interest in the scientific world. It seemed logical to implement research practices from these fields: PLs are still designed and used by humans, so there is an overlap between natural languages and PLs, and so, also how the information is processed in the brain (Siegmund et al, 2014; Prat et al., 2020).

The concepts borrowed from IP from the cognitive science realm are chunking (De Groot, 1946; Miller, 1956), its (positive) effect on cognitive load

(Sweller, 1988), and the effect of skill level (De Groot, 1946; Chase & Simon, 1973), which were studied in fields ranging from chess, natural language, to physics concepts. Relationships between these concepts and the present study are described in further detail below.

Today, more and more natural language research is done quantitively with text analyses, specifically text mining, using natural language processing (NLP) techniques. While it could be argued the field of text mining is still in its infancy, these types of techniques are readily available to use for the field of mining software repositories (MSR), a field that has received particularly more attention since the 2010s. Gupta & Gupta (2019) define MSR as "the process of analysing and extracting the knowledge and patterns from the software artefacts" (p. 243). Traditionally, this type of research is done on open-source software repositories like GitHub, where people and companies can share software freely for others to use or build on. The present study will try to use NLP techniques in an exploratory manner to find out what types of patterns (and thus chunks) subjects use during a short-term memory recall of a snippet of Java code.

In the niche field in which this thesis exists, we hope to achieve a better grasp of source code understanding, and source code analysis, and how these fields can benefit from each other. That is why we aim to answer the following research questions:

RQ1*: How does* recall *of a Java snippet differ between skill levels, and between a normal and a scrambled version?*

RQ2*: How does* chunking *of a Java snippet differ between skill levels, and between a normal and a scrambled version?*

> **RQ3**: *How can text analyses techniques be implemented on recalled source code of subjects and what can be inferred from the results?*

In the remainder of the Introduction and the Related Works Section, the research questions will be put into context and explained further.

1.2 Building on the Existing Knowledge Base

To be able to add to the already existing base of source code understanding, but at the same time maintain freedom of exploration, we use a familiar type of experiment, namely a short-term recall study that measures the differences between three different skill levels. This is inspired by a paper from McKeithen et al. (1981), who studied recall, chunking, and the effect of skill level during a five-phase ALGOL W experiment. It was found that expertise (beginner, intermediate, or expert) and version (normal snippet, or scrambled snippet) positively influence the rate of recall during the five phases. More details on this study can be found in the Method Section. In the remainder of this paper, McKeithen et al.'s study will also be referred to as the *original study*. This study was one of the firsts to study human source code understanding and processing for the sake of programming language education and cognitive psychology.

We view their paper and their results as a thread that guides us in the process of creating a viable result. We feel it is imperative to build on previously done research to stay vigilant of changes during these modern times where techniques are being renewed and updated multiple times a year. This is why we chose the original study, as it seems it provided a solid base for studies that followed and thus seemed reliable to base a new study on. The theory McKeithen et al. used, the experiment that was performed, and their findings are described in further detail in further sections of the paper.

1.3 Chunking in the Past and Present

The particular form of information organization of "chunking" spreads across multiple domains but was first coined by Miller (1956) who described it as an action where "each chunk collects a number of pieces of information from the environment into a single unit" (p. 236, Gobet et al., 2001), like forming letters into words and words into sentences. Where Miller's research started with sequences of numbers, it was soon applied in other areas such as chess (De Groot, 1946), natural language (Simon, 1974) and even physics concepts (Cheng, 1999).

Still, even today, chunking has scarcely been studied in computer science fields, let alone with different PLs, like Java. However, it almost seems obvious that chunking would also appear in programming languages as they would in natural languages, as modern high-level PLs consist of natural language elements (Hermans & Aldewereld, 2017). However, many people find learning to program challenging, because, at first glance, PLs do not resemble anything familiar, except for maybe math equations. When someone is not familiar with programming yet, they have thus no already-existing schema that aid with understanding, which creates many opportunities to chunk.

McKeithen et al., the authors of the original study, for example, found that experts remember different chunks of ALGOL W programming concepts than beginners, when they used the Reitman-Rueter technique. This technique involves making subjects remember certain concepts (in this case 21 ALGOL W keywords) and then prompting the subjects with certain concepts to spark the recall process. Subjects are asked to recall all 21 concepts in an order they prefer. This technique proved that there are different recollection techniques, thus different recall orders, for different skill levels. Beginners, for example, recalled based on orthography, e.g., all commands together that have the same length, e.g., IF-IS-OF-OR, or all commands together that start with the same letter, e.g., SHORT-STEP-STRING; or storytelling (ordering based on some type of causality), e.g., TRUE-IS-REAL-THEN-FALSE. Experts recalled based on ALGOL W-meaning, e.g., WHILE-IS-DO-FOR-STEP. This effect could also be seen in the correctness of the recollection of the snippets during the five trials: experts recalled generally more, and more correct than beginners. However, McKeithen et al. (1981) do not ascribe this to the fact that experts have *more* knowledge, they just have it better organized in the brain.

De Groot (1946) uses the example of the "Sicilian opening" in chess. Experts recall this opening as one piece of information and go on to the next. Beginners recall this opening as all its separate steps that it contains, overloading their short-term memory, which causes them to forget the rest that they have seen. The expert, however, because she/he saved 'memory space' by chunking this opening as one step, can remember and thus recall what happened beyond this more easily.

We hope to uncover this using some metrics that McKeithen et al. also used, as well as performing text analysis techniques that could show what steps (a.k.a. chunks) are remembered during the recall process. These will be described in the following sections.

1.4 Source Code and Source Code Analysis

Source code analysis is a young field and is generally used for (software) developmental processes and understanding of applications. Cardoso, Coutinho & Diniz (2017) claim it can be viewed from two perspectives. The first perspective uses source code analysis with a specific goal in mind that determines the techniques that are going to be used. This usually happens in the field of development, for example, intending to reduce execution time. The second perspective, one that we are currently more interested in, has the goal of "discovering information about an application through an exploratory experimental process that aims at uncovering unforeseen properties" (p. 100). Code analyses that determine bugs, quality, or maintenance, for example. This is interesting for the present research, as we aim to find out if simple, manual text analysis techniques can be used as a type of source code analysis.

Multiple scientists in the field claim source code analysis will only become more important over the coming years (Harman, 2010; Binkley, 2007). Being reflective towards how people code, and how computers work will forever be of interest if we wish to strive forward. A sure proof of this is the existence of around 700 different programming languages.

The language ALGOL W is a good example of this phenomenon, where iterations of languages are made to improve the communication between humans and computers. The branch of Algol languages was created from a desire to create a universal programming language that was independent of the machine that it was used on. ALGOL W specifically was supposed to be a simplification of ALGOL 60, designed by Hoare and Wirth (Sebesta, 2012). However, the project to create a compiler-independent language became so big and complex, that it finally lost popularity among programmers as well. Nevertheless, the paradigm that was built 'around' Algol, namely structured programming (Dijkstra, 1970), remains and can be found in, for example, the Java language.

This paradigm is important for the present research because it builds on the premise that coding should be done and written in chronological order, bluntly spoken. Dijkstra (1968) made a case about the GO TO statement in the Algol languages (among others) that its excessive use was “considered harmful” because it goes against people’s natural instinct for chronology and causality. Although his statements were met with some resistance (Rubin, 1987), structured programming remained an important aspect in the development of following PLs, like Java. Because we want to find out which Java concepts are recalled after one another, we hope to be able to infer mental organisations that show a certain structure based on chronology or causality.

While McKeithen et al. used the Reitman-Rueter technique to find chunking habits, they did not study the chunks within the code that was recalled in the first part of their experiment (here, they only checked for correctness of recall). We believe it is precisely this aspect of information recall in source code that is so interesting: finding out, in real-time, how chunking happens *during* the recall of a full snippet of code, and not just its separate concepts. To give an example, we could expect that experts recall a single function like a for-loop more easily (and thus more correctly) than beginners, as they know the ‘form’ of it as one piece of information (like the Sicilian opening), rather than its individual steps.

We plan to do two separate analyses that will measure different matters. First, we want to make comparative use of a dictionary, which will contain all unique concepts that are used in the solution to the recall experiment. A similar list of concepts will be created from each subjects’ answer. Comparing these dictionaries will tell how many correct concepts are in the subjects’ answer (in the correct relative order), compared to the solution. This will be, among others, the measure of recall. More information on how we measure recall as a whole will follow in Section 3.5.

Second, we plan to infer mental organisations from key terms that are recalled most often together using modern text analyses techniques. One technique that falls under this category is called an n-gram technique. Such a technique measures the distance between two sequences and how many times a certain sequence is seen in a body of text, a well-known example being "San Francisco". When the model reads "San", it will automatically recognize this as part of the sequence of "San Francisco", as there are (almost) no instances where "San" is used in itself. In other words, the model chunks the two terms, "San" and "Francisco", together as one sequence, "San Francisco". To create these from the subjects' answers we use two-grams that clump every two concepts together, discarding all non-letter characters, like in the Reitman-Rueter technique. For example, take the following line of code:

```
List keys = dataset.getKeys();
```

This line will have the following list of two-grams:

```
[[List, keys],
[keys, dataset],
[dataset, getKeys]]
```

Using these lists of subjects' answers and comparing them to the two-grams from the solution snippet, we can get an overview of similarities and thus correctness between them. We feel the Reitman-Rueter technique, besides taking too much time from the participants, is not representative of how chunking happens during the process of recalling or coding in general, which is why we chose this non-invasive way of testing the phenomenon of chunking.

This study of using source code analyses is largely exploratory, as the answers from subjects will likely contain terms that do not literally relate to the solution of the snippets. It is then the question if such techniques could be used for such subject-focused studies, and what kinds of issues we ran into and what can be changed for future research. We hope to clarify small parts of such analyses in the present study.

1.5 The Goal and Expectations

There is a gap between cognitive psychology and the understanding of source code. In the last few years the field of cognitive representation and understanding of source code has gained more attention, but not in combination with a recall experiment and the use of text analyses like in the present study. We feel it is imperative to keep research on cognitive representation and understanding of source code up to date with developments in the field, to be able to create durable frameworks. Especially renowned studies like McKeithen et al. need to be verified in a modern context to not build on false pretences. We add to this study by redoing parts of their experiments and using modern techniques like text analysis. The importance of the present study is therefore twofold: a) results could give insight into chunking and recall of source code by (computer science) subjects, and b) we show if text analyses techniques can work in a free-recall experiment like the current. Using text analysis techniques to find these chunking effects would be a fascinating addition to the fields of text mining, cognitive psychology and source code understanding.

The paper and findings of McKeithen et al. are not widely known, and we are not aware of any conceptual replications as we propose here. There are therefore multiple justifications for this replication study. First, the modifications to the original study are proposed in a way that fits the current modern time frame but will not influence results negatively, and thus could provide nuanced results for programming education and development. Second, the addition of using text analysis techniques on recalled source code is relevant, because currently this has seldom been done and thus provides valuable knowledge on the usage and the results of such techniques.

The goal is to discover if McKeithen et al.'s results for chunking and the effect of skill level are a result of the research setting, including the choice of programming language ALGOL W and, for example, the fact that code was written down, or if these were a result of actual processes in the brain that help subjects understand incoming information such as source code. The expectation here, since this phenomenon of chunking is not only seen in areas like source code but across multiple disciplines, is that chunking will happen in a similar, if not the same way as in the original study. We expect to see significant main effects for version (normal or scrambled) and skill level (novice, intermediate or expert). Furthermore, we add text analyses to the recalled source code of subjects. As this is something that has not been done before, we have no specific expectations of these results.

The remainder of the paper is structured as follows. In the Related Works section previously done research will be explicated concerning the current concepts and experiments. In the Method section, an explanation can be found about the current procedure for the experiment, as well as comparisons to the previously done study by McKeithen et al. (1981). In the Results section, quantitative results can be found and, in the Discussion and Conclusion section, conclusions will be made about the results and what these entail for future research. The paper concludes with a list of limitations and risks, a data availability statement, an acknowledgement and finally, the references used.

2 Related Work

Source code analysis has interesting roots in different disciplines. In the paragraphs that follow we will explain different concepts from different disciplines and how they come together in the present research, and thus how the present research fills a gap between these disciplines. We feel knowledge about this interdisciplinary field between computer programming and PLs, cognitive psychology and (language) learning could greatly influence the field of programming and computer science education.

2.1 Information Processing

The whole journey of source code analysis started as (general) IP. Scientists theorized about long-term memory, short-term memory, and later the working memory, and how these blocks act when people learn new information (e.g., Baddely & Hitch, 1974; Rayner, 1998). First, Miller (1956) introduced the so-called ‘magic number’ of seven elements people can hold in their short-term memory with his perception and memory concepts. De Groot (1946) introduced the notion of an efficient mental information organization of chess setups to hold bigger elements in the short-term memory, which showed the difference between experts and beginners. Herein lies the key that experts not only have more knowledge to use when exposed to chess setups, they have also organized this knowledge more efficiently; new knowledge is subsequently organized in the same way (McKeithen et al., 1981).

Furthermore, Sweller (1988) introduced cognitive load theory, which theorizes that at any given time the working memory has a limited capacity and that overloading it will reduce the effectiveness of teaching. It claims that experts organize their knowledge in different ways than beginners to be more efficient and less load-bearing for the brain. To describe this process Sweller introduces three types of cognitive load: *intrinsic load*, which is the inherent complexity of a task; *extraneous load*, distractions from the task at hand that increase load; and finally, *germane load*, where new information is linked to information that is already stored in the long-term memory.

The ultimate goal of processing and understanding information, therefore, is to get information from the working memory, where information is processed (like simple calculations), to the long-term memory, where information is linked to already known information (like storing information about addition next to information about subtraction). And because the subject here is reusing what is already there (instead of encountering it anew), the cognitive load is pointedly lowered (Sweller, 1988). This effect has been shown in different fields like cognitive psychology (Rayner, 1998) and computer science (e.g., Fakhoury et al., 2018; Nakagawa et al., 2014; Hermans, 2021), where experts were found to experience less load than beginners during experiments. It is expected this is due to a different information organisation in the brain which aids in accessing necessary information.

Sweller argues that bundles of information about one subject in the long-term memory are called schema, and the more an individual practices to apply these, the easier it gets to retrieve this type of information. This is why experts have less trouble remembering code than beginners (Hansen et al., 2013), because they have pre-existing schema, and these schemas help them reduce load in the remembering process. Also, this could be why in McKeithen et al.'s experiment, experts perform better than beginners in the normal version recall, but perform the same as for beginners in the scrambled version recall, because in that latter case, neither group had schema that fit this type of scrambled code. Similarly, this is why beginners have generally a hard time recalling because most information is new to them and thus, they do not have appropriate schema. Then, the intrinsic or extraneous load takes over, leaving no room for any learning processes.

Lee (2012) also did a study with short-term memory, using the item-method directed forgetting technique. Results showed that it is easier to forget when there are fewer cognitive resources, like in beginners, available during encoding (the learning phase before recall). This would entail that beginners forget more and easier than experts because they do not have sufficient 'coat hangers' in the brain to hang new information on, meaning they have no appropriate schema available to them.

Christiansen & Chater (2015) even went so far as to claim there are certain phases to chunking and IP, one of which that includes the prediction of chunks using "forward models". First, subjects chunk 'eagerly', meaning that the first chunks that are processed are quite big. Presumably, this would be the snippet as a whole. Second, "computing multiple representational levels of the chunks" (p. 99), which is essentially the chunking of chunks, for example, chopping the while loop into the while statement, the condition, and the action if the condition is met. Third, the anticipation of chunks, where subjects remember the chunks they have seen and keep them in their memory for future chunks; this way subjects can predict what may come using “forward models” (p. 99), for example, expecting a closing bracket for a function because you have seen it also had an opening bracket. For this reason, it is expected experts can reduce cognitive load and think further than the code that is seen on the screen: while reading, the brain is trying to make connections with the knowledge that is already stored, and actively matching this to what is being read to predict what ‘is logical’ to come after another.

We believe these schemas from Sweller and these forward models are similar to each other in some ways. The forward models would not exist if the subject did not already have some pre-acquired knowledge, and thus schema, stored in the brain. However, these schemas have certain forms and certain styles. Our Java-schema for example looks different than our Python-schema. It is safe to say that these have an overlap of certain information, as the programming languages themselves also have quite some overlaps. This also generally means that the more programming languages you know (and thus how experienced you are), the better you are at remembering source code in memorization experiments (Siegmund et al., 2014) because many code pieces are already familiar to the reader. We expect to be able to visualise the schema and forward models with the help of the text analyses techniques, to view the differences between beginners and experts. With the following section, we hope to get closer to the ‘why’ of this phenomenon.

2.2 Forward Models in Multiple Disciplines

As explained above, the various ways of chunking can determine ones' success in understanding texts, source code, or any other medium that has a form of repetition and patterns in it. To underline the fact that this phenomenon happens, even though we might not understand it completely, we present multiple fields wherein these forward models or similar trends are present. We will try to approach the reasons for this to happen.

2.2.1 Storytelling, cohesion relations and understanding

The phenomenon of 'expecting what is logical' is something that crosses over multiple disciplines, and something that has been around for a long time. According to Gamble, Gowlett & Dunbar (2014) the ability to tell stories was probably formed from gossip as a form of grooming[1] among early primates (p. 54). Because these gossip stories contained a certain causality and higher-order intentionality (people talking about other people), it is theorized this is where we humans get our knack of storytelling from. Gamble, Gowlett & Dunbar claim, among others (as will be shown), that this is how we make sense of the world.

Clark (2007) builds upon this theory within the discipline of adult learning and how narrative learning could be an effective method. She especially asserts that personal narratives create coherence between events, which could help a learner to relate to the subject to be learned on a more personal level. According to other research that Clark mentions, coherence has two elements: continuity and causality (p. 87). These are similar to the concepts

[1] Grooming, especially social grooming in this context, is one of the "central mechanisms involved in social bonding" in primates, because it releases endorphins in the brain (p. 54).

from Gamble, Gowlett & Dunbar (2014), where continuity is understood as the self in relation to others during and after gossip, and where causality is understood as relational to time (person A did this in the past and now person B is angry in the present).

Even more so, Sanders, Land & Mulder (2007), from the discipline of text comprehension, performed studies with linguistic markers in functional contexts (e.g., reading a school book). They theorize that understanding discourse means constructing mental representations of the text. The best mental representations are coherent and thus contain coherence relations, for example, cause-consequence or problem-solution. In turn, these relations are “made explicit by linguistic markers” (p.220), connectives such as ‘because’ or ‘however’. Therefore, they give the advice to use the ‘maximize coherence strategy’ (p. 226) while writing texts, which instructs to include signals in the text to aid cohesive mental representations. Again, these signals could be related to Clark’s continuity and causality, and Gamble, Gowlett & Dunbar’s concepts of storytelling, where coherence relations create the meaning behind the story or the text. While the study of Sanders, Land & Mulder focused most on text understanding, we believe they make a valid point about these linguistic markings, one that can be transcribed into the discipline of source code understanding as well (for example, in the form of procedural units, as will be shown below).

For the field of source code understanding, it appears wise to use these kinds of markings in source code as well to make them more readable and understandable. In fact, we believe some concepts already convey these relationships inside code, like while- or for-loops. While it may be difficult and impractical to incorporate 'personal narratives' (Clark, 2007), or gossip (Gamble, Gowlett & Dunbar) into a piece of code, it is very well possible to include more coherence relations into code, albeit in the form of comments, a visual structure, or even specific types of commands. While there is research within this field, it is new and simplistic, which is more than enough reason to advance this research field, and thus also all the more reason for the present research. In the following section, we will share research that studied (effects of) certain concepts that are widely used in common PLs, and how they aid (or not) to understanding.

2.2.2 Signals in source code to aid understanding

Pennington (1987), for example, wrote an extensive report on how programming knowledge influences program understanding and how the mental representation of such a program would look. It was found that procedural units form the basis of experts' mental representations of computer programs. These procedural units are instructions on how to perform a certain task following certain steps and according to Pennington, these units are sequences, iterations, and conditionals (p. 7). Interestingly enough, the same results were found for similar experiments in the field of text understanding that show that text structure is strongly related to text understanding (Pennington, 1987; Spyridakis, 1991; Sanders, Land & Mulder, 2007; Siemund et al., 2014, May). Thus, these procedural units indicate a certain (sequential) structure within computer code, as well as natural language texts. They are detected by the reader because they use the same type of markings in both domains (McKeithen et al., 1981). For example, 'while' exist both in PLs and in natural language and the reader knows this means something is going to happen for the duration of a certain condition.

Pennington describes these as phrase structures and the segmentation of these reflect the control structure of the program. It is expected that because these segmentations are familiar – we argue these could also be called chunks – they lower cognitive load and thus help the reader to understand and recall more quickly, especially experts. The question remains if this can be observed in the results of the text analysis on the subjects' answers. What concepts will be grouped together in these recalled answer sheets and will there be differences between skill levels?

In addition, Ichinco & Kelleher (2017) mention the original study in their research on chunking, skill level and source code, more specifically, source code elements. They focus on the types of concepts programmers concentrate most on. It was found that occasional and everyday programmers primarily focus their attention on core structural tokens that indicate the overarching control flow. Thus, again the focus lies on markings that indicate chronological and sequential information. Meaning detail tokens (tokens that specify details, like iteration in for loops) were mostly only remembered by everyday programmers and not by the other two groups, occasional and non-programmers. This means that the everyday programmers own the appropriate schema to remember these small details, and others do not.

Similarly, Fakhoury et al. (2018) researched cognitive load and lexicon quality, which they studied in forms of linguistic antipatterns and readability metrics of source code[2] using a technique with fNIRS[3]. These linguistic antipatterns are described as "recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of program entities" (p. 287, Fakhoury et al., 2018). It was found that linguistic antipatterns increase participants' cognitive load, however, the same was not found for the used readability metrics. Moreover, no significance was found for the treatment containing both the linguistic antipatterns and the readability metrics. This means that unclear identifications of entities had a more negative effect on cognitive load than for example the number of spaces or the lines of code. Interestingly enough, from the aspect of chunking, it was assumed that the readability metrics would also yield significant results for a higher cognitive load, because of a lack of structure (comparable to the random chess setup from De Groot, 1946, for example, or absence of linguistic markers in texts from Sanders, Land & Mulder, 2007) would make it more difficult to read and understand the source code. However, as stated by Fakhoury et al., for the participants who *completed* the treatment containing both the linguistic antipatterns and the readability metrics, an increase in cognitive load *was* found, unlike for the 60% of participants who did not complete the treatment. Still, this research and the outcomes show that cognitive load can be heightened by poor identifier-name choices and subpar structures in source code. This last part could also be related to the negative effect of scrambled source code on recall.

[2] Readability metrics that Fakhoury et al. used during the study: cyclomatic complexity, number of arguments, lines of code, max depth of nesting, variable of declarations, and number of: loops; expressions; statements; comments; comment lines; and spaces (p. 288).
[3] Functional Near-Infrared Spectroscopy is used for optical imaging in the brain that measures haemoglobin-concentration changes as indication for brain activity.

One other signal that could aid in understanding and could help with the chunking process is a beacon. Fan (2010) explains this in her work with the help of several other studies. Brooks (1963), being one of the first to define beacons, describes them as "sets of features that typically indicate the occurrence of certain structures or operations within the code". Other research in this field proved that experienced programmers in particular benefit from the use and presence of beacons; more than beginners. To further this, Von Mayrhauser and Vans (1995) claim that beacons can trigger the chunking process to reach a high-level understanding of a program. This happens because the functional structure of the program is recognized with the help of these beacons, and then hypotheses about the program structure can be accepted or rejected. This is important for the present research because it gives reason to why experts would recall more and more correctly than beginners, as experts recognize important features that indicate the structure of the program and thus, it helps in understanding and subsequently during the recall process.

2.2.3 Ways to understand and read source code

Taking this back to cognitive load and source code, more recent research has shown that beginners read source code less linearly than natural language text; and experts read even less linear than beginners (although non-significantly). Busjahn et al. (2015) studied cognitive load in terms of reading order using an eye movement measuring technique. They introduce a new term for novice reading behaviour in code: "story reading" (p. 256), which indicates that code is read from the beginning until the end and from left to right, like a story in natural language. This is similar to Christiansen & Chater's "forward models" and Pennington's procedural concepts because both relate to a certain chronological structure in chunking, reading, and understanding source code. Consequently, their finding for beginners entails that they find source code harder to read than experts. This can be found in eye fixations, which are positively correlated to cognitive effort – and thus cognitive load (Busjahn et al.). Even more, beginners have more eye fixations of longer duration than experts, meaning that beginners experience more cognitive load.

In the study of McKeithen et al. (1981), it is argued that chunks are hierarchical in essence. Using the Reitman & Rueter technique (1980), subjects produced a “hierarchical representation of information from regularities in the orders in which items are recalled over many trials” (McKeithen et al., p. 308). Some subjects of the first experiment from McKeithen et al. (although not all) were asked to learn 21 ALGOL W concepts and then recall each of the concepts 25 times with or without cues. As for the hierarchy, it was found that skill level does not have an effect on ‘nestedness’[4] within the ordered chunk trees (p. 319). However, there were differences between beginners and experts: experts mostly had ALGOL W-specific chunks, like WHILE-DO or FOR-STEP, while beginners mostly (and only) had chunks that were associated with real-life discourse, such as LONG-SHORT, TRUE-FALSE or END. We believe this is where the discipline of natural text understanding crossed over into the paradigm of source code understanding. Source code, like natural language, is easier to understand and remember when it contains markings of chronology. Both skill level groups recognize these chronology markings in the code, but

[4] A high number of nodes in ones’ ordered tree structure would indicate a high degree of nesting. Although, it was not significant between groups.

only experts recognize the non-chronology markings also, just like more advanced readers understand texts better than poor readers.

What we have seen in the literature above is that the three concepts together – chunking, cognitive load and skill level – studied in the present research are underrepresented in the scientific field of programming language understanding. For the sake of education and the development of PLs, we must take into account these concepts together, as a complete foundation cannot be produced without these. Without understanding how programmers read, learn and understand programming languages, it is of no use to create new ones. Seeing to it computer science is one of the fastest-growing fields in the corporate world, research like this can show how to proceed further and without having to make unsubstantiated assumptions about how programming languages should be designed, taught and understood.

2.3 Aptitudes for Programming

Alongside this stream of discoveries, it was thought that an aptitude for logic (like mathematics) and problem-solving meant an aptitude for programming, because PLs contain so many logical elements, and generally also problems to solve (e.g., Gazdar & Pullum, 1985; Sauter, 1986; Siegmund et al., 2020). While in essence, this might not be completely false, it turns out the relation between learning a new PL and certain aptitudes someone might have is much more nuanced (Siegmund et al., 2020).

For this reason, more research emerged how learning a new PL, might very well be much like learning a new (second) language (e.g., Shute, 1991; Connolly, 2001; Hermans & Aldewereld, 2017; Portnoff, 2018; Siegmund et al., 2020), because natural languages share many similarities to PLs. This sparked an interest in the possibility to use natural language research methods (from cognitive psychology) for programming understanding research, like text analyses techniques.

2.4 Source Code Analysis

As noted before, young computer science fields like source code analysis borrow techniques that are well-known in neighbouring fields like text analysis. However, as source code understanding still falls under the umbrella of computer science studies, such methods were quickly adapted to view source code understanding as a computer science problem. Our curiosity is founded on this juncture: could techniques used for text analysis also be used for non-working, bug-filled source code like from a short-term memory recall experiment?

Gupta & Gupta (2019) wrote a review paper about natural language processing techniques for the mining of unstructured data. They describe this as Mining Software Repositories as more and more (open-)source code repositories are becoming available, and much can be learnt from these code repositories. Their focus lies on unstructured data which includes text data, image data, and video data. They theorize that insights and patterns can be derived from such source code projects, with the help of “automatic summarization, automatic sentiment analysis, traceability analysis, mining, and preprocessing”, among others (p. 244).

As stated in the Introduction, we will execute text analysis techniques to figure out the nature of recalled source code on a deeper, more varied level than could be done qualitatively. We hope to be able to explain what kinds of techniques we employed and what resulted from these. The field of text analysis on source code is in its infancy, especially when we are doing research on code that is recalled on a time limit and thus faulty. We explore possibilities of using such techniques and will infer future uses for these.

3 Method

3.1 Introduction

The present study is inspired by McKeithen et al.’s study (1981) and set in a more modern, and immersive setting. However, we believe this will not interfere negatively with the results. We believe our research setup is similar to the original study, except for the fact that the testing setting is more intuitive and immersive than in the original, where subjects had to ‘code’ on paper. It is expected this will only enhance the results previously found. However, if in any case, results demonstrate to go in the opposite direction, this could also give insight into possible changes to multiple factors surrounding the experiment, as well as the current environment. Also, we expect that the concepts of chunking, cognitive load, and skill level have not changed internally (in the brain) or externally (in the coding output) in participants over the past years.

In the following sections, we will go over the original study and its research setup, and the changes that we made that deemed fit. Then we specify proceedings of the experiment and some preliminary procedures before analyses could start.

3.2 The Original McKeithen et al. Study and Changes Made

McKeithen et al. (1981) were one of the first to study human source code understanding and processing for the sake of programming language education and cognitive psychology. With the knowledge of chunking (e.g., De Groot, 1946; Miller, 1956; Zeschel, 2008; Thalmann et al., 2019; Hermans, 2021), and the effect of skill level on recall in mind, McKeithen et al. sought to understand the mental organizations of source code using two experiments: a recall experiment; and an experiment using the Reitman-Rueter technique (Reitman & Rueter, 1980).

To confirm the abovementioned effect of skill level on IP and recollection on source code, McKeithen et al. completed experiment 1 on their sample using ALGOL W source code: “the classic expert-novice difference in short-term recall” (p. 309) study comparing recall results between a normal coherent computer program and a scrambled version across three skill level groups. This experiment contained five phases of each two minutes studying the code snippet and three minutes of recalling. The differences between the skill level groups and the versions were significant, also throughout the different phases. As these results proved only an *external* effect of (skill level on) short-term recall, they completed experiment 2 to clarify *internal* effects: inferring subjects’ mental, and possible hierarchical, organisations of 21 programming concepts from ALGOL W using ordered tree structures (from Reitman & Rueter, 1980). What followed was an analysis of the differences between skill levels and these ordered trees. It was found that experts chunk the concepts programming language-specific, thus based on ALGOL W, whereas the beginners chunk the concepts natural language-specific, thus on English. However, this difference was not significant. They did find, though, when they compared the types of chunks being recalled, that experts are more cohesive as a group than either of the other two groups. They claim these results could have stemmed from the distribution across the skill levels, and from the assumption that chunks only occur when the code is being understood, and not per se when it is only being read.

At the time of the original study, chunking had not yet been tested in-depth in the field of computer science, which had few established measures to perform studies. The original study was therefore one of the first to study ALGOL W, or any PLs for that matter, in this light. The original study was set up as follows (p. 309):

> *31 Subjects of three skill levels were shown a 31-line ALGOL W computer program in either normal or scrambled version for five 2-min study trials. Groups of two or three subjects saw the program projects onto a screen by an overhead projector. In the 3-min recall period that followed each trial, subjects wrote as much of the program as they could remember on a blank recall sheet, putting each recalled line on the sheet as close to its presented position as possible. Subjects could not look at previous attempts of trials and were asked to recall everything anew on each trial.*

In the current environment, due to COVID, it seems unwise to bring subjects together in a classroom, as well as the fact that projecting code onto a screen seems very unintuitive for the modern coding environment. For these reasons there are some practical changes in terms of modernity (and therefore physical constraints, which will follow) we made to the research setup:

1. We transferred the experiment to an online questionnaire, so that subjects can perform the experiment in the comport of their own familiar homes, which is in line with COVID regulations, and which will lower any occurring experiment effects.
2. The snippets to be shown and the editor where the recollection of the snippet was done is in a source code text editor from Ace (Ace, n.d.). This includes automatic indentation and highlighting. Highlighting,

indentation and so-called whitespace (horizontal and vertical white spaces in code) are a big part of source code understanding (Bauer et al., 2019; Fakhoury et al, 2018; Hansen et al., 2013), so we expect the inclusion of these aspects inside the recollection phase of the experiment could positively impact recall. More on this in Section 2.2.2.

3. We found there are multiple reasons to use Java instead of ALGOL W as our experiment programming language:
    a. As ALGOL W is an old PL with a now-small following, it would be unwise and unpractical to perform the study in ALGOL W, as it would be challenging to find beginners, let alone experts;
    b. Java is a more modern, widely-used PL, as it is still being taught across universities, and also being used across multiple developers' jobs in daily working life;
    c. As Java is based on the Algol-language group, it is based on the same paradigm, namely structured programming, and thus contains similar features. Further, as we believe structured programming plays an important role in the processing of PLs (in a fashion that is chronological and sequential), a language from this paradigm seemed logical.
    d. We could reuse a snippet that was previously used by Fakhoury et al. (2015), which provided us with the assurance that the snippet would be useful in a research setting made for remembering and recollection.

This online-based research setup, however, brought a big constraint that had to be changed accordingly. The original study took at least 35 minutes (five trials of each 5 minutes, appropriate time between these, and introduction and conclusion to the study). We found this too long to ask of subjects because we cannot assure the efficacy of the subjects of such an experiment without supervision. Therefore, we found this was the biggest constraint that followed some smaller practical changes to the setup to enhance efficacy in terms of time constraints:

1. We found changing the number of trials performed would be the fastest way to shorten a questionnaire session. This is why we changed the number of trials from the original five to the current one-trial recall experiment.
2. We considered a one-time trial, however, a 2-min study and 3-min recall would be very short so we lengthened the study and recall periods to 160 seconds, and 240 seconds, respectively, so we kept the 2:3 ratio.
3. The length of the original ALGOL W snippet was 31 lines. The currently used snippet length was slightly increased, but only following strict guidelines to enhance possible recall and to make up for the loss of the number of trials. Thus, the Java-snippet that was used contains 40 lines, but a) it includes an 8-line comment, which supposedly helps for chunking and understanding (Fan, 2010; Busjahn et al., 2015), and b) it includes some white lines which help in understanding source code (Hansen et al., 2013). We trust Fakhoury et al. (2018) did the necessary studies to confirm the usefulness of the snippet.
4. The original study used the Reitman-Rueter technique (1980) to determine chunking within different skill level groups. However, we find this method too passive, as it is not done in a natural coding

experiment (like the recall study), but rather in a learn and prompt session of separate concepts. Also, our interest in text analyses techniques brought us to another explorative technique to use for the present research, namely text mining techniques. As this is done after the collection of the data, it will take no (extra) time from the participants.

Using the abovementioned experimental setting changes, we used the original study as our base, but the goal was not to replicate per se, but mostly to build on the original study, which is why some decisions and changes have been made. We believed, just like McKeithen et al. at the time, that these changes would provide us with strong information about chunking and understanding of source code. Finally, we have tried to keep the essentials from the McKeithen et al. study the same. This included, but was not limited to, the time participants got for learning the snippet, the time they got for recalling the snippet, the size of the snippet, and, in the best way possible, the contents of the snippet. In further sections, we will describe the participants of the study, the procedure and materials, and finally the preliminary procedures.

3.3 Participants

McKeithen et al. focused the experiment on computer science students from a specific education program. We planned for a more varied group of subjects than this original plan. We hoped to find subjects from all kinds of disciplines, with all kinds of skill levels.

We copied the classification of McKeithen et al.’s skill levels: beginners, intermediates and experts. This is a self-classification scale, which was proven by Siegmund et al. (2014) to be a good indication of programming experience. Formerly, the original study classified skill level as follows:

A) Beginner: Students that just started their first ALGOL W course. Some of these had previous experience in either BASIC or FORTRAN (two similar structured programming languages).
B) Intermediate: Students that just finished their first ALGOL W course. Some of these had previous experience in either BASIC or FORTRAN.
C) Expert: Subjects that teach ALGOL W, have over 2000hr of general programming experience, and have an average of over 400hr of experience in ALGOL W.

We altered this classification slightly to fit our needs:

A) Beginner: just starting a Java course or similar programming language.
B) Intermediate: just finished a Java course or similar programming language.
C) Expert: Over 2000hrs (250 working days) of general programming experience and over 400hrs (50 working days of Java experience).

For all participants, it was not specifically necessary to learn or use Java daily, but we were looking for groups (especially in the beginner category) who are starting to learn programming in a language that at least resembles or is based on Java (like C++, C#, Groovy, Scala, Processing, Yeti). Also, we explained some knowledge in programming is advised. We have actively pursued (learning) groups of these languages within our home university in Leiden, in which we have personal contacts.

Half of each skill level group have seen the normal version, half the scrambled. With three skill level groups and one trial, this resulted in a 2 x 3 factorial design. An a priori power analysis indicated that a total calculated sample size of 31 subjects would be sufficient to detect a significant interaction effect (*F*(2, 25) = 0.6) at version, skill level and trial with a given power of 0.8 and an alpha of 0.05. Moreover, as the pilot study resulted in more than 30 participants, we expected to find around 50 participants total (given the experimental setting for the present study is more time consuming). Also, we have asked subjects their age and gender to determine the distribution of the sample and to see if it is representative of the field in which we operate the study.

Possibilities for distribution among students and experts are the computer science faculties at our home Universiteit Leiden, The Netherlands, or the Vrije Universiteit in Amsterdam. Though, Java is usually not a language that is taught right away to students entering a bachelor computer science program. That is why we expected not to find a lot of experts among this group, however, the opposite turned out to be true: it was much harder to find beginners in the field willing to participate. However, the PERL research group[5] that this research is a part of has a wide range of interested people in computer science education and interested followers on social media, as well as people who follow the mailing list of the PERL research group. Because we are not limited to classroom students like McKeithen et al., we can use social media and personal contacts to broadcast the questionnaire among interested people even

[5] The Programming Education Research Lab can be found via the following link: https://perlliacs.wordpress.com/

outside of our own circle. For example, by using fora and the PERL Twitter account managed by the second author. As we want a diverse group of people with different skill levels, we will also distribute the experiment website among master of science- and PhD students working in STEM fields and Computer Science fields. As for the experts, we will ask specific teachers or professors that are very familiar with Java, most of which we know from our home university in Leiden. As for the beginners, we will ask specific students from the master program Media Technology, because it is known first-year students get a crash course in Processing, which is closely related to Java.

More information on the actual sample that was used to perform statistical analysis on can be found in the Results Section of the paper.

3.4 Procedure and Materials

The procedure of the experiment is as follows. A questionnaire will be made for subjects to participate in the study. This way, participants can enter completely voluntarily and it is expected there will be less of an experiment effect because of this. Following a link[6], participants enter the webpage that starts with an introduction, followed by an explanation of the main question with a recall period with a timer, a page for descriptives and finally a short thank you note. The questionnaire will take around max. 15 minutes to fill in.

The introduction explains, in short, the goal of the study, who the researchers are, what the contents are of the questionnaire, and finally, it includes a privacy statement. When subjects accept the conditions of the study and choose to participate, they will be shown a short explanation for the main

[6] A preliminary version of the questionnaire can be found via the following link: http://liacs.leidenuniv.nl/~hermansffj/questionnaire.html .

question. There are two varieties to the main question, as was the case with McKeithen et al. The test group will be shown a normal snippet, and the control group will be shown a scrambled version. These will be chosen at random at the time of the questionnaire via a script (Javascript). The main question contains either of the snippet versions and is as follows:

1. A short explanation of the question, in which it is explained that subjects are to recall a snippet once under a time limit.
2. Once a button is clicked that assures the participant is ready, a snippet is shown (as an image to make sure subjects do not copy the code) for a time frame of 2 minutes and 40 seconds (160 seconds) to learn the snippet as well as possible. It is not possible to click on 'Further' until the time limit has expired to make sure participants don't race through the questionnaire.
3. After the time limit has expired, it is expected of subjects to click the 'Further' button to continue to the next phase. A short explanation of the recall period is shown. Once a button is clicked that assures the participant is ready for recall, an online, inline, empty text editor is shown for a time frame of 4 minutes (240 seconds). This text editor field is an embedded Ace editor which "matches features and performance of native editors such as Sublime, Vim and TextMate" (Ace, n.d.). During this phase, it *is* possible to click on 'Further' before the time limit has expired.
4. A summary of what participants have just done will be shown, and it will be explained that they are already one-third of the way there.

After clicking once again on 'Further', participants move on to the descriptives.

After the trial, we ask participants to answer some descriptive questions. Here, very basic and limited personal information is asked: age, gender, skill level in programming (as by McKeithen et al.), approximate years of programming experience, programming languages the subjects know, and highest degree pursuing or completed. After the participants have filled in all questions (all required) participants are asked to go to the last page. Here it is possible to fill in an email address if people wish to know the results of the study after completion.

Age and gender are asked to study the distribution and representativeness of the data. The skill level question asks the subjects to self-classify. To check the validity of this, we compare these answers to the approximate years of programming experience, and the (amount of) programming languages that subjects are familiar with. For this last question, we decided that it is not necessary to have full mastery over a language to tick it on the list. The list we used was from the Tiobe index website, a well-known Dutch website that tracks the popularity of languages in terms of most used. The languages listed are:

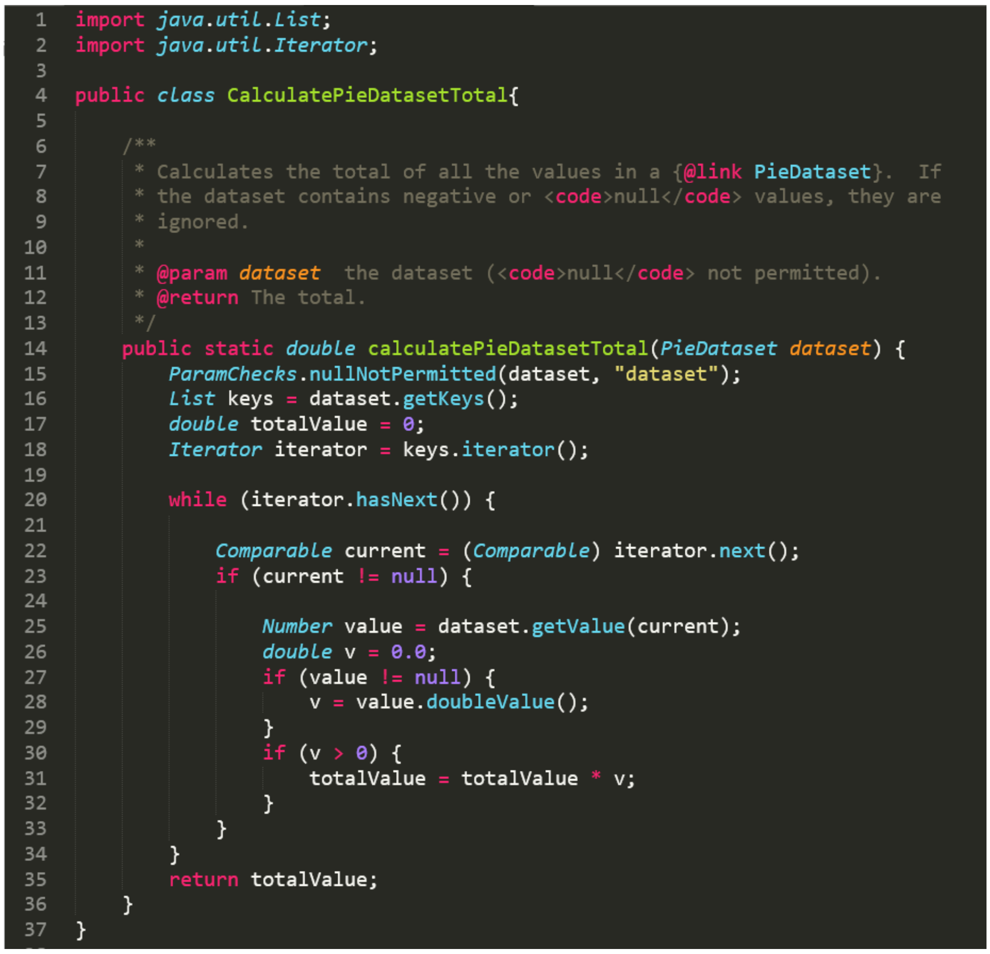
- ❑ Assembly
- ❑ C
- ❑ C++
- ❑ C#
- ❑ Dart
- ❑ Go
- ❑ Groovy
- ❑ Java
- ❑ JavaScript
- ❑ MATLAB
- ❑ Objective-C
- ❑ Perl

- ❑ PHP
- ❑ Python
- ❑ R
- ❑ Ruby
- ❑ Rust
- ❑ SQL
- ❑ Swift
- ❑ Visual Basic

We also added tick boxes for Other and None, in case this arises, although we expect few respondents to check these, thanks to the nature of our anticipated sample.

Originally, McKeithen et al. let participants do this same exercise five times in a row (thus repeat the main question another five times). However, as we are doing the questionnaire online without supervision, we found it unwise to subject participants to a questionnaire duration of at least 40 minutes (including introduction, the trials, the descriptives and the final word).

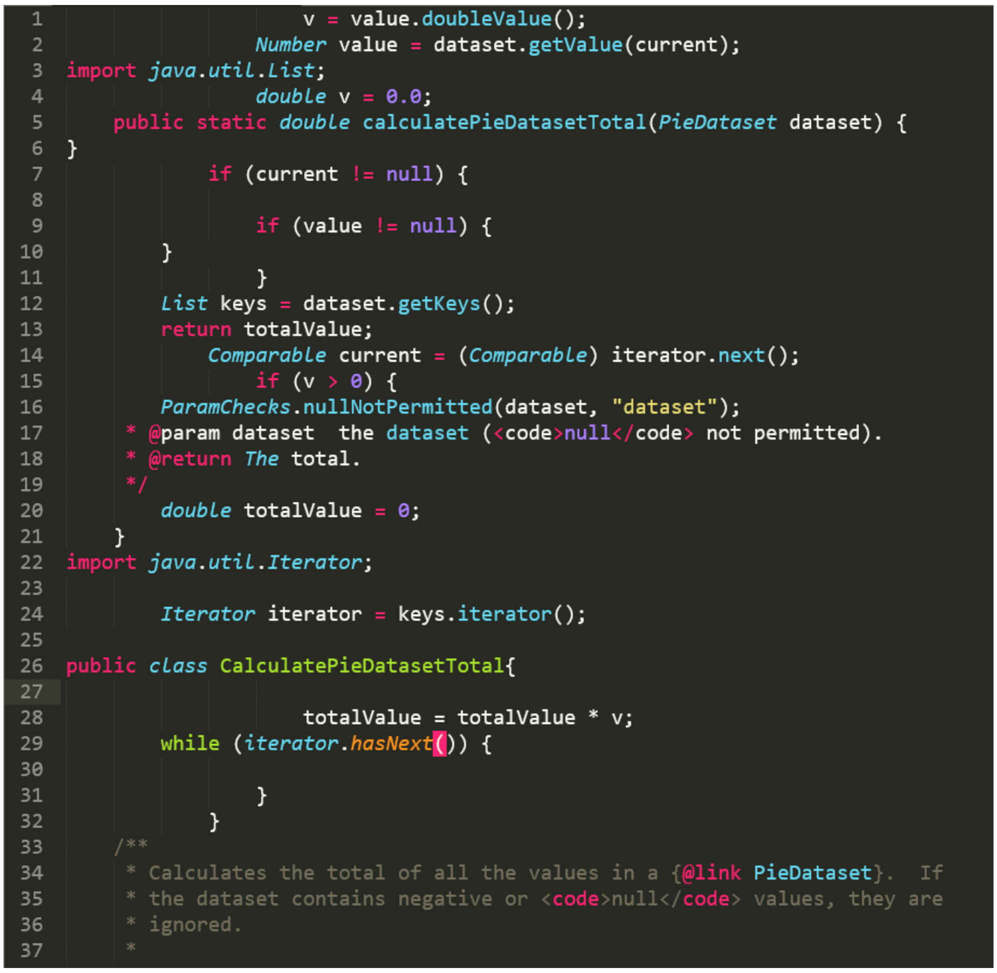
Figure 1. Source code snippet used for the normal condition.



```java
import java.util.List;
import java.util.Iterator;

public class CalculatePieDatasetTotal{

    /**
     * Calculates the total of all the values in a {@link PieDataset}.  If
     * the dataset contains negative or <code>null</code> values, they are
     * ignored.
     *
     * @param dataset  the dataset (<code>null</code> not permitted).
     * @return The total.
     */
    public static double calculatePieDatasetTotal(PieDataset dataset) {
        ParamChecks.nullNotPermitted(dataset, "dataset");
        List keys = dataset.getKeys();
        double totalValue = 0;
        Iterator iterator = keys.iterator();

        while (iterator.hasNext()) {

            Comparable current = (Comparable) iterator.next();
            if (current != null) {

                Number value = dataset.getValue(current);
                double v = 0.0;
                if (value != null) {
                    v = value.doubleValue();
                }
                if (v > 0) {
                    totalValue = totalValue * v;
                }
            }
        }
        return totalValue;
    }
}
```

As was stated in the literature Section above, we are reusing a snippet from Fakhoury et al. (2018). The snippets used for both conditions can be seen in Figures 1 and 2 on the previous and current page. The chosen snippet for the present study is from a project called JFree-Chart with a method called calculatePieDatasetTotal (from DatasetUtilities.java).

Figure 2. Source code snippet used for the scrambled condition.



```java
                    v = value.doubleValue();
                Number value = dataset.getValue(current);
import java.util.List;
                double v = 0.0;
    public static double calculatePieDatasetTotal(PieDataset dataset) {
}
            if (current != null) {

                if (value != null) {
        }
                }
        List keys = dataset.getKeys();
        return totalValue;
            Comparable current = (Comparable) iterator.next();
                if (v > 0) {
        ParamChecks.nullNotPermitted(dataset, "dataset");
     * @param dataset  the dataset (<code>null</code> not permitted).
     * @return The total.
     */
        double totalValue = 0;
    }
import java.util.Iterator;

        Iterator iterator = keys.iterator();

public class CalculatePieDatasetTotal{

                    totalValue = totalValue * v;
        while (iterator.hasNext()) {

                }
            }
    /**
     * Calculates the total of all the values in a {@link PieDataset}.  If
     * the dataset contains negative or <code>null</code> values, they are
     * ignored.
     *
```

As can be seen, the snippet is 37 lines long of which 32 actual text lines (including an 8-line comment and 6 lines with only brackets). We chose to keep the length of the snippet approximately the same as McKeithen et al. (1981) because we gave subjects more time, and our snippet contains a comment to give away the general structure of the snippet, to reduce cognitive load within subjects. Comments are proven to help with chunking (Busjahn et al., 2015).

We made sure that the snippet is still understandable in 30 seconds or less, without losing the complexity of, for example, loops within loops. What's more, it does not contain references to external methods, and the snippets contain easy to understand English identifiers, which are usual guidelines for snippet design by Fakhoury et al. (2018). Besides the fact that appropriate pre-runs have already been done with the snippets, Java seemed like an obvious choice as there are not many other structured programming languages that have the same consistent usage over the past few years (from Tiobe.com).

In summary, the following changes and/or improvements have been made:

1) The biggest change to the recall experiment is the number of trials used for the experiment compared to McKeithen et al. Whereas the original study had five trials, the present study has only one. We believe this is enough to still demonstrate the effect of skill level on recall. Also, we found it too extreme to let subjects perform a questionnaire that would take up to 40 minutes, and we expect subjects might cancel midway if they find the questionnaire too boring or repetitive. Moreover, we have sufficient justification for this choice without losing the essence of the original study.
2) Instead of the experiment taking place in a classroom where subjects had to view the code snippet on a projection screen, subjects can now do the questionnaire from the comfort of their own home or workspace. This will ensure a more intuitive and natural working environment, which will provide a presumed reduction of any possible experiment effects.

3) Instead of having to write down the recall answer on a sheet of paper with a pen, subjects can now code and edit in an immersive, imbedded text editor on their own familiar computers, which will aid in recalling due to features like automatic indentations, automatic highlighting, and a form of spell-check. As these are all features that are all developed and proven for better readability of code, we expect this will provide us with less noisy results.
4) As mentioned before, the snippet is now a Java snippet. Choosing to reuse the original study's snippet would not have proven the effect of skill level on chunking, but rather that ALGOL W is now quite unknown. Plus, we are trying to replicate the modern (programming) learning environment, and thus a more modern PL was the proper choice.
    a. The length of the snippet is 37 lines long, of which only 18 effective text lines, instead of the original that had 31 lines of ALGOL W. However, we did the necessary research to provide us with the certainty that this snippet length does not interfere with cognitive load or chunking effects. Also, we kept the length of the snippet relative to the time frames set (for reading and recall) by McKeithen et al.
    b. The timing for the present study is relative to the timing of the original study, meaning: McKeithen et al. used a timeframe for the reading of the snippet of two minutes and a recall timeframe of three minutes. If this is relative to 31 lines of ALGOL W, then 18 effective text lines of Java are

approximately relative to the current 160 and 240 seconds (2 minutes, 40 seconds, and 4 minutes, respectively).

5) We used the same skill level scale as McKeithen et al., meaning subjects have to choose between being a beginner, an intermediate, or an expert at coding. However, we believe to be able to get a broad, more varied sample of subjects, as explained under the Participants Section.

3.5 Preliminary Procedures and Variables

In the questionnaire, responses from each participant are gathered in a JSON package and stored in a password-protected bin on a safe server. Most of the single answers for one participant will be values like integers (e.g., age) or strings (e.g., the recall answers per trial), and some of these will be lists (e.g., the programming languages a participant is familiar with). To export all responses from the server, a JSON script is written to save all data in an excel datasheet. Further analyses will be done using R Studio and python, with appropriate packages.

There are multiple variables to analyse. We planned to use text analysis techniques to infer relations of recall and chunking in the data, however, for this, it is necessary to pre-process it. For each final answer from the subjects, this consisted of removing punctuation (special characters) and white space and adding all unique concepts a subject used to a dictionary. This is a list of concepts that the subject used in the order the subject typed, which will be called *subjectDict*. Uppercases were left in the subject's answers, as there are differences between using an uppercase or a lowercase for some concepts (like

'Iterate' versus 'iterate'). An explanation of variables follows below; for a summary view table 1 below.

From *subjectDict*, *subjectDictCount* could be inferred, which is the amount of unique concepts within *subjectDict* (in other words, the length of the list). Because a similar list was created from the solution snippet (*solutionDict*), the overlap between each *subjectDict* and *solutionDict* could be calculated. Two variables arose from this: *overlapDict*, which contained the words that occur in both lists, and *overlapCount*, the length of *overlapDict*. Together, these create *relativeOverlap*. Essentially, this means *overlapCount* divided by *subjectDictCount*. Finally, the relative size of the subjectDictCount compared to the length of solutionDict is *relativeSize*, however, because these are so strongly related, no statistical analysis will be run with *relativeSize*.

These variables all concern the recall of the subjects of the solution snippet, but using two-grams we also create variables that determine the chunking within subject answers. Thus, from the solution concept dictionary (*solutionDict*) we also created two-grams, and because the length of *solutionDict* is 65, there will automatically be only 64 two-grams. For example, in the dictionary ['Anna', 'really', 'likes', 'cheese'], the two-grams will be: [['Anna', 'really'], ['really', 'likes'], ['likes', cheese']], which has a length of three two-grams. We use the terms two-grams and chunks interchangeably in the following.

The variable *twoGramsDict* is created, which is a list of two-grams of a subject's answer. This is compared to the list of two-grams from the solution snippet, which results in the variable *twoGramsOverlap*. Also, the length of *twoGramsDict* will become *twoGramsCount,* however, because it relates strongly to *subjectDictCount*, no further statistical analysis will be done on *twoGramsCount.*

Table 1 Types of variables used to determine recall within subject's answers

Variable	*Type*	*Explanation*
solutionDict	List of strings	All concepts in the solution snippet
solutionDictCount	Integer (65, fixed)	The length of solutionDict
subjectDict	List of strings	All concepts in a subject’s answer snippet
subjectDictCount	Integer (3 – 55)	The length of subjectDict
overlapDict	List of strings	All concepts that occur in both the solution snippet, and the subject’s answer snippet
overlapCount	Integer (2 – 42)	The length of overlapDict
relativeOverlap	Integer (percentage)	(overlapCount / subjectDictCount)
twoGramsDict	List	All two-grams from a subject’s answer snippet
twoGramsCount	Integer	The length of twoGramsCount
twoGramsOverlap	Integer	All two-grams that occur in both the solution snippet, and the subject’s answer snippet

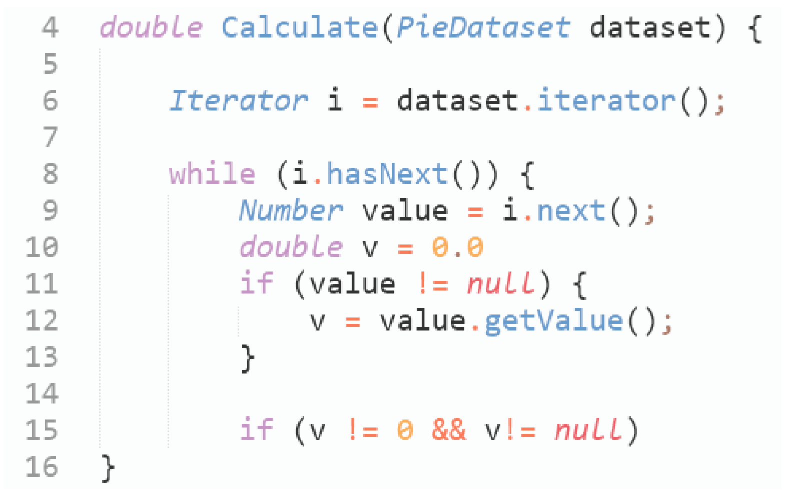
In the following Results Section, analyses will be done on all variables in order of the previously mentioned research questions.

4 Results

In this section, all analyses and results will be shown that were performed on the two independent variables (version, expertise), and the four dependent variables (subjectDictCount, overlapCount, relativeOverlap, and twoGramsOverlap). First, a summary of the general data is provided. Second, correlation effects between the dependent variables are shown. Then, the first and second research questions are answered in succession. The third research question, being reflective and conceptual in essence, is answered in Section 5.

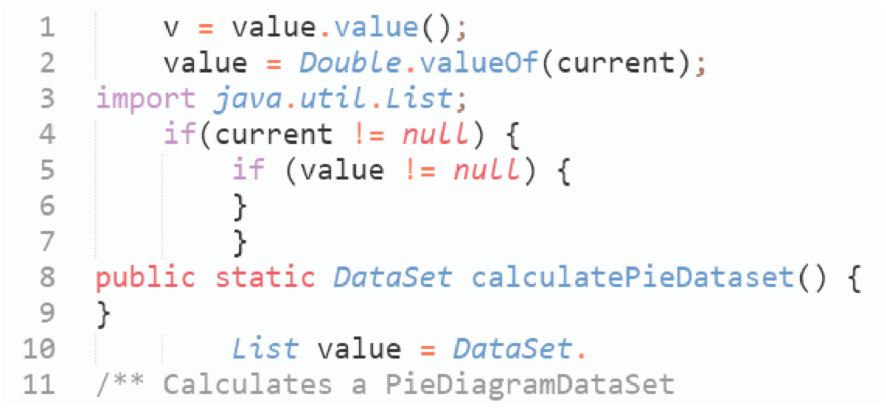
As examples, below two answer snippets from two participants are shown in Figures 3 and 4.

Figure 3. Example answer from subject that recalled the normal version.



```
double Calculate(PieDataset dataset) {

    Iterator i = dataset.iterator();

    while (i.hasNext()) {
        Number value = i.next();
        double v = 0.0
        if (value != null) {
            v = value.getValue();
        }

        if (v != 0 && v!= null)
}
```

Figure 4. Example answer from subject that recalled the scrambled version.



```
    v = value.value();
    value = Double.valueOf(current);
import java.util.List;
    if(current != null) {
        if (value != null) {
        }
        }
public static DataSet calculatePieDataset() {
}
        List value = DataSet.
/** Calculates a PieDiagramDataSet
```

The dataset was cleared of insufficient answers. These fell into two categories. The first category contained answers with either fewer than five lines of code or where pseudo-code was used exclusively. Second, due to a bug in the code, 13 answers did not include a tag for the version which was shown to the subject. From these answers, the version had thus to be inferred. An extra variable was created to keep track of these in case of abnormality in the data. When a version could not be inferred for sure, the subject and its answers were deleted from the dataset.

The final dataset contains 67 subjects of which 33 are in the normal code group (version A), and 34 in the scrambled code group (version B). The average age of the participants is 35, with outliers of 20 and 78 years old. Of all subjects, there were 11 females, 51 males, 3 subjects that preferred not to say, and 2 subjects that identified as other (genderqueer and non-binary). Considering the expert level distribution in the dataset, there were 40 in the expert group, 23 in the intermediate group, and 4 in the novices group. Subjects had a wide range of native languages, the largest group being English (20 participants), and Dutch (21 participants). Of all participants, 27 have finished or are currently pursuing a master degree, 23 have finished or are currently pursuing a bachelor degree, 13 have finished or are currently pursuing a PhD and four have other occupations or degrees. These are representative of the field of computer science.

Contrary to what was previously expected, an imbalance in the skill levels presented itself during the acquisition of the data. For this reason, two sets of statistical analyses have been done to find out if the effects are due to the imbalanced sample or not. These can be found in Sections 4.1, and 4.2, respectively.

Between all continuous variables, a correlation analysis was done using Pearson's r. A summary of these results can be found in the table below.

Table 2 Variable pairs that showed a medium to large correlation

Variable pairs	*r-value*	*p-value*	*Correlation*
overlapCount - twoGramsOverlap	0.85	< 0.05	Large
subjectDictCount - overlapCount	0.71	< 0.05	Medium - large
overlapCount - relativeSize	0.71	< 0.05	Medium - large
overlapCount - twoGramsCount	0.71	< 0.05	Medium - large
overlapCount - relativeOverlap	0.58	< 0.05	Medium - large
relativeOverlap - twoGramsOverlap	0.53	< 0.05	Medium
subjectDictCount - twoGramsOverlap	0.52	< 0.05	Medium
relativeSize - twoGramsOverlap	0.52	< 0.05	Medium
twoGramsCount - twoGramsOverlap	0.52	< 0.05	Medium

What can be seen in the table above is that the largest correlation occurred between overlapCount and twoGramsOverlap (*r*(65) = 0.85, p < 0.05). This can be explained as follows: as subjects use more correct concepts in their answer snippet, they are apparently more likely to have included the correct order of these concepts. This in turn creates correct twoGrams that also occur in the solution snippet. The next medium to large correlation occurred between subjectDictCount and overlapCount (*r*(65) = 0.71, p < 0.05), which shows that when subjects have a large dictionary, they usually have more overlap with the solution snippet than when they have a small dictionary. The following two medium to large correlations between overlapCount – relativeSize and overlapCount – twoGramsCount were to be expected, because both relativeSize and twoGramsCount are strongly related to subjectDictCount ($r(65) = 0.71$, $p < 0.05$, and $r(65) = 0.71$, $p < 0.05$, respectively). The last medium to large correlation effect is between overlapCount and relativeOverlap ($r(65) = 0.58$, $p < 0.05$). This means that when a subject has a large number of concepts that overlap with the solution snippet, the relative overlap is also bigger. This is also related to the fact that relativeOverlap is a function of subjectDictCount and overlapCount. Other medium correlation effects can be found in the table.

4.1 RQ1: Recall

In this section, statical analyses will be performed to answer the first research question: How does *recall* of a Java snippet differ between skill levels, and between a normal and a scrambled version? This will first be done with the independent variables version (versionA, versionB) and expertise (novices, intermediates, experts), and the dependent variables subjectDictCount, overlapCount, and relativeOverlap. After this, to correct for the disbalance in samples within the expertise levels, the analysis will be done with version and expertise (intermediates, experts), and the dependent variables. Using a MANOVA the following hypothesis and null-hypothesis are tested:

> *H_1: For subjectDictCount, overlapCount, and relativeOverlap the means of all groups are unequal.*
> *H_0: For subjectDictCount, overlapCount, and relativeOverlap the means of all groups are equal.*

A two-way MANOVA was performed to test differences between group means for version and expertise across the three different dependent variables. First, there was no significant interaction effect between version and expertise on the combined dependent variables, $F(6, 120) = 0.38$, $p = 0.89$; $V = 0.04$. This means that the effect of the version on the dependent variables is the same for all skill level groups. Thus, there were no main effects found for either version, $F(3, 59) = 0.46$, $p = 0.71$; $V = 0.02$, and expertise, $F(6, 120) = 0.71$, $p = 0.64$; $V = 0.07$. Consequently, the null hypothesis is accepted. This means that the answer to **research question 1** is as follows: Recall of a Java snippet does not differ between skill levels and versions.

To determine if the imbalance in group sizes decided the result of the two-way MANOVA, multivariate multiple regression was performed to test differences between group means for version and expertise (intermediates and experts only) across three dependent variables. There were no effects found for version, $F(3, 58) = 0.33$, $p = 0.81$; $V = 0.02$, or expertise, $F(3, 58) = 1.05$, $p = 0.38$; $V = 0.05$. Even though the p-values are different (higher for version, but lower for expertise), the outcome is still not significant, and thus, the null hypothesis is still accepted.

4.2 RQ2: Chunking

In this section, statical analyses will be performed to answer the second research question: How does *chunking* of a Java snippet differ between skill levels, and between a normal and a scrambled version? This will first be done with the independent variables version (versionA, versionB) and expertise (novices, intermediates, experts), and the dependent variable twoGramsOverlap. After this, to correct for the disbalance in samples within the expertise levels, an analysis will be done with version and expertise (intermediates, experts), and the dependent variable. Using a two-way ANOVA, the following hypotheses and null-hypotheses are tested:

Null hypotheses:	Alternative hypotheses:
(a) There is no difference in average twoGramsOverlap for any version.	*(a) There is a difference in average twoGramsOverlap by version.*
(b) There is no difference in average twoGramsOverlap for any skill level.	*(b) There is a difference in average twoGramsOverlap by skill level.*
(c) The effect of one independent variable on twoGramsOverlap does not depend on the effect of the other independent variable.	*(c) There is an interaction effect between version and skill level on average twoGramsOverlap.*

A two-way ANOVA was performed to test if there are main effects and interaction effects of version and expertise on twoGramsOverlap. We did not find a significant difference in average twoGramsOverlap by both version, $F(1, 61) = 1.33$, $p = 0.25$, and expertise, $F(2, 61) = 1.17$, $p = 0.32$. Also, there was no significant interaction effect between version and expertise on the dependent variable, $F(2, 61) = 0.16$, $p = 0.85$. Consequently, the null hypotheses are accepted. This means that the answer to **research question 2** is as follows: Chunking of a Java snippet does not differ between skill levels and versions.

To determine if the disbalance in group sizes decided the result of the two-way ANOVA, another two-way ANOVA was performed to test differences between group means for version and expertise (intermediates and experts only) for the dependent variable, twoGramsOverlap. We did not find a significant difference in average twoGramsOverlap by both version, $F(1, 58) = 1.50$, $p =$

0.23, and expertise, $F(1, 58) = 0.66$, $p = 0.42$. Also, there was no significant interaction effect between version and expertise on the dependent variable, $F(1, 58) = 0.08$, $p = 0.78$. Thus, the outcome is still not significant, and thus, the null hypotheses are still accepted.

5 Discussion and Conclusion

This section is constructed as follows: first, the third research question will be discussed and reflected upon, with suggestions for future research. Second, quantitative results of the analysis (the first and second research questions) are discussed and contextualised with the use of related works. Finally, a summary of all results is presented.

5.1 RQ3, Results, Implementations, and Suggestions

In this first section the research question 'How can text analyses techniques be implemented on recalled source code of subjects and what can be inferred from the results?' will be reflected upon and answered. As this is more of a conceptual question, we aim to reproduce how we came to our results and how they could be improved, using some qualitative examples from subjects.

Looking back at the data and the analysis, there is a realisation that the program that produces the data needs to be highly detailed and goal-oriented. The dictionary from the answers was created by removing punctuation and non-letter characters, except for number values. This is done in python using functions and for-loops to iterate over subjects' answers. As it is not completely automated, but also not annotated by hand, concepts disappear from the concept list or get added to the list wrongfully. Even though there is more control on what gets filtered out and whatnot, the program used needs to be very specific to extract the right kind of information, which can be a very time-consuming job to complete. Due to time constraints for the present research, it was decided to focus on (the dictionary of) concepts and the two-grams. These exclude indentations, white space, and newlines, which give the program overall structure. What is lost is then the order in which concepts and special characters are used, especially across multiple lines. This could potentially be detrimental to the precision and recall of the text analysis. The questions arise then, first, how many of the selected items are relevant for the current research task, and secondly, how many relevant items are actually selected?

Furthermore, we use two-grams as a measure of chunking, or to be able to make conclusions, to some extent, about the order of subjects' answer concepts. However, because the two-grams are created based on the concept dictionaries – which may or may not include mistakes – the two-grams are also to be taken with a grain of salt, which may be why we found that the overlap between two-grams by the subject and two-grams from the solution snippet was meagre.

This brings us to a suggestion for possible future works, as this study proved was there is still a lot to learn from subjects of different skill levels and recall studies, but also from the way we analyse these. One thing that was deliberately not chosen for the present study due to a shortage of time and knowledge in that specific field, was to use existing text overlap metrics, like ROUGE. It stands for Recall-Oriented Understudy for Gisting Evaluation. This is a type of machine learning software that evaluates automated summarization techniques within the field of natural language processing. However, it could be of great use in the field of source code analysis also. Particularly ROUGE-n could be used for the present research as it is "an n-gram recall between a candidate summary and a set of reference summaries" (Lin, 2004). The candidate summary would then be the subjects' answers, and the reference summary would be the solution snippet. The result is the co-occurrence of n-grams within the subject answer and the solution snippet. Future works using this technique could expand on the present study.

Looking back at the code, some things went incorrectly and need consideration. This was found by looking more closely at the data and going through the answers of the subjects by hand. Some issues were noticed that our program did not catch:

1. Mistakes in uppercase and lower cases were not ignored and thus counted as wrong. Initially, these were kept in the solution snippet on purpose, as there are distinctions in meaning for, for example, the identifier name `iterator` and the class name `Iterator`. This is similar to the following point.
2. General typos were counted by the program as wrong. For example, we saw that one subject made the mistake of writing `iterator` as `itterator`. Now each time this subject used this word, it was not recognized by the program as this item was not in the list of correct concepts from the solution snippet, even though the rest of the code might have been very well written and recalled. The question is then how strict we should be when measuring for recall. As the study was inspired by McKeithen et al. (1981), we felt a certain obligation to stay true to their rules, however, as the goal of the recall was not to write a completely functional program, but

rather to recall the meaning of the program including all its intricacies, these types of problems arose during analysis.

3. Some subjects used different PLs to recall the snippet. Theoretically, this was not the concept of the experiment, however, what if a subject were to use a completely correct program that would execute the same idea as our Java snippet? One subject did so in C#, however, as this is not in our skillset, we had no means to check if the program was correct. This was mostly seen as wrong by our analysis (despite the overlap of Java and C#), however, a future analysis might be able to filter out different PLs, and make meaning of this.

These are just some of the problems that might arise when research is done on source code that could potentially be very buggy. However, following the analysis and looking at the data gave inspiration for potential changes to the experiment design that could help clarify results in the future.

5.2 Discussion of Research Questions 1 & 2

Unfortunately, neither of the independent variables turned out to affect the dependent variables. Multiple conclusions could be derived from these results.

First, experiment wise, there were some concerns about the snippet being too long to be able to recall. However, for this reason, we included a comment, kept the same amount of actual text lines in the snippet, and gave subjects more time than was rationally necessary. A pilot study proved that recall of a snippet was inherently difficult, meaning that intrinsic load for all participants would be high. We assumed that a higher skill level would be able to surpass this disadvantage, and thus score higher for the measurement of recall. This was not the case. This leads us to believe that there are external or internal problems within the experiment that made this study very difficult to perform. It could have been the fact that there was no supervision and people were not concentrating, or the fact that the time limit was still too short.

However, one factor that stands out is the way that we measured skill level. After data acquisition had already been done, a peer made us aware of the fact that this scale for skill level was in fact very outdated and even unrepresentative of the field. Indeed, this might even be why the novice skill level group turned out to be so small, and also maybe why the intermediates and experts were so closely related in the results. Beforehand, we chose to copy this classification from McKeithen et al. (1981) for several reasons: a) we feel this was a limitation of the original study and not per se a feat for the current research to solve, b) using more skill level groups would require more subjects which we feel is not desirable under current conditions, c) the current power analysis is based on three skill levels, so adding more levels would result in a lower effect of our independent variables on our dependent variables, and d) we have chosen to keep the number of levels, but change the label descriptions slightly to a more modern view of programming experience.

However, in future studies, it would be advisable to use a more modern scale such as the one from Dreyfus & Dreyfus (1989) or Mead et al. (2006), which is less about the literal experience and more about programming experience in the conceptual sense. Moreover, more recent research has shown that a self-classification technique of programming experience with five scales is better than one with only three (Siegmund et al., 2014).

Additionally, a point that follows the previous point, the present study did not prove the familiar skill level effect like De Groot (1946) proved with his chess studies. However, this could be due to the small sample of novices, or it is not an effect as traditional as previously thought. Especially for the current research field, this warrants more studies in the field of source code understanding.

Second, a closer look at some of the answers showed that even some of the subjects that were presented with version B (the scrambled version) sometimes put the elements of the solution snippet in somewhat the right order. Linking this back to the related works, this could potentially mean that even though subjects have seen a piece of code that makes no sense, their brain helps them remember in a way that seems logical, or more familiar to them, because this reduces load. Clark’s (2007) concept of continuity helps with understanding this phenomenon because coherence aids understanding, and continuity is one of the pillars of coherence. More research would be necessary on the order of recall to determine this possible effect, as it was not within the scope of the current research.

Third, source code analysis on code that is filled with mistakes due to a time limit and experiment effects is going to be difficult and the process to analyse this needs to be detailed and streamlined. This is something that was experienced during the analysis of the current results. We believe it is not in our interest to make sure subjects make as few mistakes as possible, but rather, to create analyses techniques that see through these mistakes and thus still be able to create viable results. However, to make it a little bit easier for subjects to make sure their code actually works is to make sure that subjects can run their code to check for bugs. This is a common practice among subjects, more so in beginners than in experts, but it can help subjects conceptualise the code better, and thus create data that could be of more use to the researcher. We also believe this would help in creating a more immersive environment, that mirrors the environment of a real programming integrated development environment, like JupyterLab, or RStudio.

5.3 Summary

The present research studied the effect of version and expertise on recall and chunking of a Java snippet, inspired by a study previously done by McKeithen et al. (1981). Findings show that there are no significant effects of the independent variables on the dependent variables, however, the study proved fruitful in multiple ways. First, source code analysis using text analysis techniques need to be refined for future use but show a wide range of possibilities. Second, the study raised questions about the order of recall of subjects, which would indicate subjects recall in a manner that is more logical to them, something that can also be found in Christiansen & Chater’s (2016) forward models of chunking. Third, a framework for recall and n-gram analysis was created. Future research should focus more on the quality and accuracy of techniques to determine recall and chunking in source code, however, we can build on the field of text mining that already uses machine learning techniques for natural language purposes.

6 Limitations and Risks of the Study

Like any study, there are some risks and limitations. First, it could prove difficult to find enough subjects for each skill level. We do not believe it will be difficult to find participants in general, because we have broad faculty- and social media relations that prove helpful in these situations. The only slight concern is to find enough beginners. Our pilot study and experience proved that finding beginners using fora and twitter is sometimes not fruitful. However, we have taken appropriate measures to be sure to reach a varied group of subjects, like approaching specific programming courses in our faculty.

Second, following the first point: may it be that we find a limited number of subjects, this would also be a risk to the study. However, according to the power analyses, an amount of 31 subjects is sufficient for a viable effect.

Third, as we test the hypotheses using only one language, Java, the results might not be generalisable across other programming languages. However, we find this a good reason for further research with a wider range of PLs. Also, it was found later that the snippet that was used contained a mistake, which was part of the original study it previously was a part of (Fakhoury et al., 2018). This did not come to our attention until after the study had been performed, because Java is unfortunately not in our PL-skill set. However, as the mistake was not very big, and it was included in all versions of the snippet, we do not see this as a big limitation to the study. However, it would be interesting to repeat the same research in another study without the error.

Fourth, the original study used five trials instead of the current one. However, in the current environment considering COVID, we found it unwise to perform an experiment in person, as well as unwise to make the questionnaire too long for subjects. This last might cause a boredom effect, which would limit the validity of the current experiment. Of course, this means that the present study is less comparable to the original and that effects might have changed, that might not have happened, had we kept the five trials. This is a limit of the present study, but in the future, we might be able to repeat the same experiment in real life once more.

Fifth, as the analyses used to determine the variables for recall and chunking are used in a rudimentary and exploratory manner, they are not perfect. However, human-created code is also just that. More advanced techniques can be used to find patterns in the code that include non-roman characters like brackets and commas, which were excluded from the present study. We think this is a field where much can be learnt from available techniques, as well as human-written code.7 Data Availability Statement

For reproduction or otherwise experiments, we will keep the dataset available upon request.

8 Acknowledgements

I would like to thank Felienne Hermans and Ross Towns for supervising this thesis from such different perspectives. Both have given me renewed interest and energy to create the thesis like it is now. I would also like to thank the peer reviewers for their attendance and insight. Furthermore, I would like to thank Suzan Verberne for her insights on the text analyses part of the study. Finally, I would like to thank Tom Kouwenhoven and Fábio Costa during my struggles with programming that concerned the text analyses of the experiment, and the intricacies of the online questionnaire, respectively.

9 References

Ace. (n.d.). Built for Code. https://ace.c9.io/ Recovered on November 13th 2020.

Atkinson, R. C., & Shiffrin, R. M. (1968). Human memory: A proposed system and its control processes. In *Psychology of learning and motivation* (Vol. 2, pp. 89-195). Academic Press.

Baddeley, A. D., & Hitch, G. (1974). Working memory. In *Psychology of learning and motivation* (Vol. 8, pp. 47-89). Academic press.

Bauer, J., Siegmund, J., Peitek, N., Hofmeister, J. C., & Apel, S. (2019, May). Indentation: simply a matter of style or support for program comprehension?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (pp. 154-164). IEEE.

Binkley, D. (2007, May). Source code analysis: A road map. In *Future of Software Engineering (FOSE'07)* (pp. 104-119). IEEE.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, *18*(6), 543-554.

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... & Tamm, S. (2015, May). Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension* (pp. 255-265). IEEE.

Cardoso, J.M., Coutinho, J., & Diniz, P. (2017). Source code analysis and instrumentation. In Simpson, J. (Ed.). *Embedded Computing for High Performance: Efficient Mapping of Computations Using Customization, Code Transformations and Compilation.* Elsevier Inc.

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive psychology*, *4*(1), 55-81.

Chase, W. G., & Simon, H. A. (1973). The mind's eye in chess. In *Visual information processing* (pp. 215-281). Academic Press.

Christiansen, M. H., & Chater, N. (2016). The now-or-never bottleneck: A fundamental constraint on language. *Behavioral and brain sciences*, *39*.

Clark, M. C. (2001). Off the beaten path: Some creative approaches to adult learning. *New directions for adult and continuing education*, *2001*(89), 83-92.

Connolly, J. H. (2001, July). Context in the study of human languages and computer programming languages: A comparison. In *International and*

Interdisciplinary Conference on Modeling and using Context (pp. 116-128). Springer, Berlin, Heidelberg.

Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Communications of the ACM*, *11*(3), 147-148.

Dijkstra, E. W. (1970). Notes on structured programming.

Dreyfus, H.L. & Dreyfus, S.E. (1989). *Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer*. Oxford: Basil Blackwell.

Ebbinghaus, H. (1913). On memory: A contribution to experimental psychology. *New York: Teachers College*.

Fakhoury, S., Ma, Y., Arnaoudova, V., & Adesope, O. (2018, May). The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (pp. 286-28610). IEEE.

Fan, Q. (2010). *The effects of beacons, comments, and tasks on program comprehension process in software maintenance*. University of Maryland, Baltimore County.

Gamble, C., Gowlett, J., & Dunbar, R. (2014). *Thinking big : How the evolution of social life shaped the human mind*. London: Thames & Hudson.

GeeksforGeeks (2020, September 25th). *Luhn Algorithm.* https://www.geeksforgeeks.org/luhn-algorithm/ Recovered on February 3rd 2021.

Gobet, F., Lane, P. C., Croker, S., Cheng, P. C., Jones, G., Oliver, I., & Pine, J. M. (2001). Chunking mechanisms in human learning. *Trends in cognitive sciences*, *5*(6), 236-243.

De Groot, A. D. (1946). Het denken van den schaker [Thought and choice in chess]. *Amsterdam: Noord Hollandsche*.

Hansen, M., Lumsdaine, A., & Goldstone, R. L. (2013). An experiment on the cognitive complexity of code. In *Proceedings of the Thirty-Fifth Annual Conference of the Cognitive Science Society, Berlin, Germany*.

Harman, M. (2010, September). Why source code analysis and manipulation will always be important. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation* (pp. 7-19). IEEE.

Hermans, F. (2021). The Programmer’s Brain, Manning 2012.

Hermans, F., & Aldewereld, M. (2017, April). Programming is writing is programming. In *Companion to the first International Conference on the Art, Science and Engineering of Programming* (pp. 1-8).

Ichinco, M., & Kelleher, C. (2017, October). Towards better code snippets: Exploring how code snippet recall differs with programming experience. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 37-41). IEEE.

Lee, Y. S. (2012). Cognitive load hypothesis of item-method directed forgetting. *Quarterly journal of experimental psychology*, *65*(6), 1110-1122.

Lin, C. Y. (2004, July). Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (pp. 74-81).

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, *13*(3), 307-325.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., & Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *ACM SIGCSE Bulletin*, *38*(4), 182-194.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, *63*, 81-97.

Nakagawa, T., Kamei, Y., Uwano, H., Monden, A., Matsumoto, K., & German, D.M. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment. In *Proceedings of the International Conference on Software Engineering (ICSE).* 448-451.

Pane, J. F., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, *54*(2), 237-264.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, *19*(3), 295-341.

Portnoff, S. R. (2018). The introductory computer programming course is first and foremost a language course. *ACM Inroads*, *9*(2), 34-52.

Processing Foundation (n.d.). https://discourse.processing.org/ Recovered on February 25th 2021.

Rayner, K. (1998) Eye movements in reading and information processing: 20 Years of research. *Psychological Bulletin 124*, 3. 372-422.

Reitman, J. S., & Rueter, H. H. (1980). Organization revealed by recall orders and confirmed by pauses. *Cognitive Psychology*, *12*(4), 554-581.

Rubin, F. (1987). GOTO considered harmful considered harmful. *Communications of the ACM*, *30*(3), 195-196.

Sanders, T., Land, J., & Mulder, G. (2007). Linguistics markers of coherence improve text comprehension in functional contexts. *Information Design Journal*, *15*(3), 219-235.

Sebesta, R.W. (2012). Some Early Descendants of the ALGOLs. In *Concepts of Programming Languages (12th edition)* (pp. 140-154). New York, New York: Pearson.

Shneiderman, B., & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, *8*(3), 219-238.

Siegmund, J., Kästner, C., Liebig, J., Apel, S., & Hanenberg, S. (2014). Measuring and modeling programming experience. *Empirical Software Engineering*. *19*(5), 1299-1334.

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., ... & Brechmann, A. (2014, May). Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th international conference on software engineering* (pp. 378-389).

Siegmund, J., Peitek, N., Brechmann, A., Parnin, C., & Apel, S. (2020). Studying programming in the neuroage: just a crazy idea?. *Communications of the ACM*, *63*(6), 30-34.

Simon, H., & Chase, W. (1988). Skill in chess. In *Computer chess compendium* (pp. 175-188). Springer, New York, NY.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive science*, *12*(2), 257-285.

Thalmann, M., Souza, A. S., & Oberauer, K. (2019). How does chunking help working memory?. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *45*(1), 37.

Von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, *28*(8), 44-55.