2019

Code readability: A proposal on the effects of psychology and comprehension in software development and maintenance

Ethan Brian Sankey
University of Northern Iowa

CODE READABILITY: A PROPOSAL ON THE EFFECTS OF PSYCHOLOGY AND COMPREHENSION IN SOFTWARE DEVELOPMENT AND MAINTENANCE

A Thesis Submitted

in Partial Fulfillment

of the Requirements for the Designation

University Honors

Ethan Brian Sankey

University of Northern Iowa

December 2019

This Study by: Ethan Brian Sankey

Entitled: Code Readability: A Proposal on the Effects of Psychology and Comprehension in Software Development and Maintenance

has been approved as meeting the thesis or project requirement for the Designation University Honors

_____	_____
Date	Dr Andrew Berns, Honors Thesis Advisor, Computer Science
_____	_____
Date	Dr. Jessica Moon, Director, University Honors Program

There are many stereotypes about the world of software development and those who inhabit it. Most peoples' general proficiency with technology has steadily risen over the past few decades, mainly in part to the more common use of household computers, smartphones, and other devices that have become more technologically integrated. However, not many people really understand the underlying pieces and frameworks that combine to form their devices. On top of that claim, I would venture to say that many people are almost willfully ignorant to the more complicated parts of the tech world, either based on an underlying feeling of ineptitude when it comes to the inner workings of their devices or a misplaced conception that such knowledge is reserved for a more elitist group of people who base their career in technology. Whenever I tell people that I'm a Computer Science major or that I work with computers, a decently high percentage of people will immediately joke about asking me to check out their laptop for viruses or ask if I can be their personal Geek Squad when the need arises. You would think based on these social conceptions that someone in a tech field is a master of all technology, or at least a considerable portion of it.

However, something I've learned over the past few years as a Computer Science student is that there is no such thing as a real expert in the field of computers and technology. The field itself is vast enough, covering topics such as software development, construction of devices, innovation and improvement on existing technology, data science, cyber-security, and other seemingly small portions that combine to create the field as a whole. Going even further into the rabbit hole, I would say that very few people are even masters of their "sub-field" as it were. Instead, many developers, programmers, and other tech-savvy people are extremely specialized in their knowledge, either because their career path required it or they just never needed to expand their

knowledge past the immediate scope of their job. Many will have a basic knowledge of most things relating to their field of choice, but very few will ever truly master multiple aspects.

Another matter to take into account is that the general field of technology is ever expanding. Machine learning, automation, virtual reality, and dozens of other subfields were at most in their infancy several years ago but are now becoming hot-button issues in the tech world. Many people will have no interest in the new applications for such technology because they feel it won't apply to them in any meaningful capacity.

One of the most crucial things to come out of these ever-branching paths of technology are the languages and systems that were built to run these new machines. As more and more people became familiar with programming, more and more programming languages began to appear. With all these newly-specialized machines and devices being invented, developers would design their own language that was best suited to the tasks they required. Using a general purpose, commonplace language like C could be done, sure, but when you could improve efficiency and optimize program size by getting rid of all the unnecessary bells and whistles for something much more barebones or specialized, C became less appealing. Over time, this has led to a staggering number of different languages and language families. Everyone had their own syntax, their own way of running functions, their own way of compiling and decompiling. There are currently hundreds of unique high-level programming languages out there, each with their own purpose(s). While the sum of all these languages is definitely a staggering feat of human engineering, it would be near impossible to become proficient in all of them. They are just too diverse and specialized to be considered functionally similar.

Yet all these programming languages (or most of them, I should say for the sake of correctness) are based on one common concept: human communication. They all had to spawn from a human brain, thinking human thoughts and codifying them into language that was deemed acceptable for the programming needs at hand. They are not random words strewn together that make fancy computer magic happen. All the different functions, constructors, variable instantiations of every language have meaning to those who are proficient in it, similar to how words in the English language have meanings assigned to them. Every piece of syntax in a programming language has a designated purpose, to the point where programs will not run if you include items or symbols they do not have an assigned meaning to.

So while all these languages were based on human language and communication, with each part of a codebase having specific intent, there are still two problems that nearly every programmer or software developer will encounter: **Reading code is hard, and reading code you are unfamiliar with is even harder.** You would think that people who spend thousands of hours writing tens of thousands of lines of code would have a firm grasp on the entire genre that is "coding". To an extent this is true; knowing how one programming language works does make it easier to learn other languages, similar to how knowing English and Spanish could help you to decipher other Latin-based languages. But there is so much nuance and diversity within the user base of a single programming language that, when asked to review a fellow programmer's code in a language you are both proficient in, you could be so unfamiliar with their personal coding style that you may not even know what they are attempting to do with their code. The spoken language analogy works also works well here if you think of it like a comparison between

accents: A person who has never left their small hometown in Montana may have some trouble fully comprehending what a person from New Zealand is saying, even though they are both speaking English. From a coding standpoint variable names, indentation, spacing, comment style and depth, and many other small nuances that coders pick up over the years can develop into a sort of personal coding "accent" that may confuse other developers that look over your code.

Some people even have trouble reading their own code if they have not actively worked on the project recently, possibly because the "game plan" they had in their head while working on it isn't as fresh or they may be confused by their old coding style habits that they have not employed as of late.

This problem has always fascinated me. Because of the diversity and complexity of the hundreds of coding languages out there, code readability has become more and more of an issue as the years have passed and as the popularity of technology that required built-in computers has increased. With so many different formats, styles, and restrictions on each language, even a developer with experience in only a few common languages may have trouble remembering which language allows for certain indentation, which language requires variable instantiation, which language requires return statements at the end of functions. While I can appreciate the diversity and efficiency that having many different languages provides, the pure amount of rules and restrictions provided by the thousands of people that constructed these languages to their preferences can make for a chaotic, confusing minefield of rights and wrongs in the coding world.

With this problem in mind, I would like to dig deep into research on the topic of code readability, hoping to gain some insight into what makes code “readable” or what makes the difference between “good code” and “bad code”. I believe that there is a strong psychological element at play, and because the world of coding is so widespread and free-form at times, it may be hard for people to truly score code on it’s readability. A lot of it will come down to personal opinion, which is unlike most modern spoken languages have strict rules of grammar. While there are strict rules to compiling code and most languages, there are usually many ways to get a program to run “correctly”, albeit potentially inefficiently. However, most experienced coders can look at a block of code and give an approximation as to how “correct” the code may be. That is the aspect that I want to investigate. I would like to see if I can nail down what choices have to be made in the programming process to make code that is considering good by the general public (which in this case would be the programming community.)

While I cannot commit the time or resources to investigating many languages, I will be pulling research from similar academic studies on a few of the more common languages in the hope of gaining some insight on potential reasons for the amount of duress that reading code can cause it’s user base. I will then be compiling the fruits of that research with the results of tests that Dr. Berns and myself have conducted to figure out just what it takes to make “good code”, or if there is any reason to believe there are better ways to establish coding languages. Perhaps the results of our study will be useful in the future for helping in the creation of languages that are easier to learn and comprehend at a base level, opening up the field of programming to a wider audience.

Since I am at least partially approaching this topic from a psychological perspective, I devoted some of my research hours into the more cerebral parts of coding, as opposed to the cut-and-dry syntax related issues that many developers struggle with. Dr. Berns, my thesis advisor, found an article specifically on that topic and recommended it to me, which provided some interesting potential background on the mental behind-the-scenes of programming. In his article *Psychology of Code Readability,* Egon Elbre of *Medium* speaks on the topics of memory and attention span in relation to programming. Programming can be complex at times, meaning you have to focus on several tasks at once such as: interpreting the requirements of a programming project into coding logic, typing that code out, comparing that code to previous code to make sure it will mesh well with other functions, and many other potential sidebars.

Because people have a limited amount of things they can focus on at once, this can create problems when trying to write code, with the difficulty to focus and write effective code rising as the tasks you are focusing on increase in number or become more complex. To quote Elbre (2018), “As any programmer knows we have a limited capacity to think about things. This is our working memory limit...For all intents and purposes we can assume we have a small number of ideas we can process in our head at a given time.”.

Luckily, the human brain has ways to adapt to the many tasks it wants to focus on at once. A key point that comes into play when talking about memory and focus are what Elbre calls the concept of “chunking”.

Chunking refers to the brain’s ability to “chunk” related memories and ideas together, such as the digits of a phone number. By associating specific pieces of information, the brain can build up

long term memories and it can also pick up on stimuli that you have stronger knowledge on more easily based on those memories. From a coding standpoint, this relates especially to the idea naming variables. It is almost easier to comprehend what code is supposed to do when your variable names have a strong meaning that cannot be easily confused with other data. For example, when you name a list of phone numbers *phone_nums*, you are more likely to remember what the 7-digit numbers in the list are supposed to mean than if you had just named it *num_list* or *pn*. This concept can be applied to many aspects of programming, but quick relation and communication of information is one of the most obvious.

So while we as people are capable of focusing on multiple tasks and stimulating things at once, it helps when we can relate those stimuli to each other mentally. This concept actually applies heavily to examples of object-oriented programming, where you create a class that represents an object with various distinct variables. For example, a class representing a person would most likely have an age, a height, a weight, and a name. If we know all of these variables and their values, we can assemble a mental picture of the type of person it could represent (e.g. if the height was a large value like 2 meters, we could potentially find it more likely that this person also weighs more than the average person, given that most people of increased height are larger all around.).

This example of mental picturing actually relates to the next psychological topic I researched: the concept of the mental model. Stephen Young, an author for *Medium On Coding*, talks at length about the ways to decrease the complexity and confusion of your code in his article, *Why your code is so hard to understand*. To quote Young (2014), “Almost all the code you write is trying

to solve a real world problem...In order to solve any real world problem we first need to form a mental model of that problem.”.

Young has a very good point when he stresses the importance of creating a good mental model. Rushing into a coding project can lead to sloppily constructed, inefficient, or just plain unusable code. Assembling a mental model, a sort of mental outline of what you need to the code to do and how you can relate that to your knowledge of the coding language, can help reduce the complexity (and potentially increase the efficiency) of the program as a whole. Essentially, taking the time to construct a well thought out model with meaningful structures and efficient code can drastically reduce the complexity and stress of programming. To quote Young (2014) again, “...you need to form a model of a solution that will achieve your programs intent. Lets call this the semantic model...The journey from the program intent to semantic model to code needs to be as smooth as possible...As you go from abstract program intent to concrete code the choices you make should be driven by the clarity with which you’re able to represent the more abstract model below it”.

To relate the above information to the overall topic of this thesis, making code that is clear, concise, and effective is in part related to the amount of preliminary groundwork the developer does before even touching the keyboard. Code can easily become too complex if the developer does not have a strong grasp on what the end result should look like or if they get too overwhelmed by the amount of bells and whistles they want to include in the final product. Keeping a clear mind that is following a clear plan will help in the creation of simple, effective

code that can be understood by anybody.

I believe it is apparent now that there is a heavy psychological element at play in the minds of developers. There is a general consensus that code that is well-formatted, lacks unnecessary repetition of functionality, and is overall comprehensible upon reading is considered “good code”. Based on the information hypothesized by Elbre and Young, taking the time to make sure you code makes sense mentally can do a great deal to increase the readability of the final product code. As I claimed earlier, all programming language and syntax is at least somewhat based in human communication and spoken language. Someone (or multiple someones) who created a language from scratch had to make specific choices when designing their programming languages, choosing keywords or callsigns that have significant meaning in their area of context . Just like taking the time to carefully word an essay or a letter, taking the time to simplify and codify the intent of your program into easily explainable notions (e.g. “I want this function to accept a calendar date and tell me what day of the week that date is tied to.”) can drastically reduce the complexity of code, and by using coding languages that make use of succinct, easily identifiable syntax you can create code that not only gets the job done efficiently but is also easily readable to developers of varying levels of expertise. While experience with a language will have obvious positive effects on a developer’s ability to read and comprehend code, relying on the innate human ability to connect coding syntax to it’s related spoken language meaning provides an underlying increase in comprehension even in developers who are unfamiliar with the language at hand.

While I was doing more research into the psychological aspects of programming, I found out that

there is actually a sort of sub-genre of psychology, called Psychology of Programming, that relates heavily to these concepts of communication and comprehension. Jorma Sajaniemi of the Department of Computer Science and Statistics at the University of Joensuu in Finland wrote for *Human Technology* on the topic in their article *Psychology of Programming: Looking Into Programmers' Heads*. As defined by Sajaniemi (2008), "Psychology of programming (PoP) is an interdisciplinary area that covers research into computer programmers' cognition; tools and methods for programming related activities; and programming education. The origins of PoP date back to the late 1970s and early 1980s, when researchers realized that programming tools and technologies should be evaluated based on their computational power only, but also on their usability from the human point of view, that is, based on their cognitive effects." (p. 4).

While Psychology of Programming (herein referred to as PoP) may not be an officially recognized branch of psychology, the concepts related to it are of great interest. Sajaniemi goes on to explain that the main motivation of PoP is to improve existing programming tools as well as help develop new ones. By analyzing the use of programming tools such as Integrated Development Environments (IDEs for short) and how users feel about their experiences with the tools and languages from a psychological perspective, PoP researchers can attempt to determine what it is about modern programming that people find understandable, confusing, enjoyable, and frustrating. They can then compile their findings and present them for peer review and presentation, trying to spread awareness for psychological aspects involved in programming and attempting to improve the overall experience.

While the rest of their article in *Human Technology* does not go on to describe any significant

movements or achievements that PoP researchers have made, Sajaniemi brings up an interesting concept in the idea of “cognitive dimensions”, also known as CDs. According to Sajanimei (2008), “...consider cognitive dimensions (CDs), which were introduced by Green (1989) to describe, compare and control how programming language features affect program design strategies. The dimension role-expressiveness, for example, relates to how well a piece of program code (e.g., “+”) reveals its meaning without a need to study the context of the piece (addition, string catenation, etc.).” (p. 4-5).

This is exactly the type of thing I consider crucial when thinking about code readability and how to improve the foundations of coding. The plus sign (+) can have several meanings within one coding language, like how in Python it can be used for the addition of integers and other numeric variables, or for the concatenation of strings or character elements, or even used as a part of the incrementation sign (x += 1). While it has several versatile uses, the context in which the sign is used makes all the difference in how we understand it to be used. This also relates once again to my claim that programming languages are based in human communication. Even though the plus sign has different operations associated with it in Python, there is still always a loose association with conjoining or combining of multiple pieces associated with it. Concepts like this may seem a little obvious or too sensical to even question them as intended behavior in a human-designed language, but the creators of Python really could have assigned any number of other symbols to fulfill the operations that the plus sign occupies now. They could have used a dollar sign ($) or a question mark (?) or some other symbol and achieved the same results. But they did not, and instead went with a symbol that already has an association with summing or connecting items together when used outside of programming circumstances. As I said, this may seem like reading

too deep into a relatively simple operation, but making those seemingly obvious choices while developing a programming language that create an intuitive, comprehensible programming environment can be what separates an enjoyable language from one that programmers dread using.

Now that we've established some psychological elements that could potentially define the separation between good code and bad code, we can start to talk about code readability. Code readability can have several different definitions and dimensions depending on who you ask, but the general idea is that the easier your code is to follow and understand from a logical standpoint as well as a stylistic standpoint, the more readable the code is. These standpoints can be hard to define and can vary based on the level of experience of the developer, especially when considering things like standard style guides or style recommendations set in place when using a specific language. Regardless, there is still at least a somewhat informal level of observation and critique involved when reading your code and the code of other developers, resulting in an overall definition of the code as somewhere between an exemplary level of clarity and something that needs a lot of work before it can be considered usable.

But how do we define code readability? Most developers can perform a cursory glance at a page of code in a language they are familiar with and give at least a loose rating to how readable they would consider it to be. Many developers with experience in one language could probably even evaluate code in a language they have little to no experience in. But there is no true formal structure to making code run (besides the limitations set in place by the language's syntax, but even messy-looking code can still run effectively). What is it about readability in code that is so

hard to objectively define? There are hard rules set in place for the grammar of a written language, which is probably the closest analogy to following syntactic standards in code.

Considering each individual coding language has its own levels of complexity and abstraction and its own style or format it would be almost foolhardy to compare code in one language to another in terms of readability. Perfectly readable and concise code in one language could look grotesque and messy in comparison to code in another language that accomplishes the same task. There could be an argument made that an entire language in general could be considered more readable in comparison to another language, based solely on characteristics such as line length, indentation, and the complexity of variable interaction and operators. So while readability is definitely a concept that exists, you must consider readability relative to some standard, possibly one set by the style guide of a language or by the standards you were taught to code by.

Regardless of the presence or lack of a systematic grading scale for the readability of a section of code, there are still certain aspects or concepts to look for in good code. Many of these concepts are present, regardless of language. Code that respects guidelines concerning the maximum length of one line of code, for example, is usually considered good code.

Take Python for example, which usually recommends a maximum of 80 characters per line of code (or any number up to 120 characters, depending on who you ask) and at most 72 characters per line if that line consists solely of comments. The reasoning given by Guido van Rossum, Barry Warshaw, and Nick Coghlan in the *PEP 8 Style Guide for Python Code (2013)*, the official style guide for Python, is as follows: “Limiting the required editor window width makes it

possible to have several files open side-by-side, and works well when using code review tools that present two versions in adjacent columns." While this guideline may not necessarily relate to the readability of a singular window of code, it does help to keep the general complexity of individual lines of code to a minimum while also providing the benefits described in the quote above.

Many language development teams provide official style guides similar to the PEP 8 guide for Python, and over time the communities surrounding a language can begin to develop best practices of their own. So while there are language-specific characteristics for good code, they are also many characteristics that are considered good or optimal in most modern languages.

Erik Dietrich of *Submain*, a software quality blog lists some qualities that he believes to be the keys to "good code" in his article *How Do You Evaluate Code Readability?*. One of the foremost techniques to increase comprehension in code is to make sure your variables are well-named so that visualization of the effects and changes to the program make sense. According to Dietrich (2018), "Use descriptive member names to make it clear, at a glance, what's happening. As a rule of thumb, you should also use more descriptive names for larger scopes. For instance, naming a loop counter variable "i" is fine when the scope of the loop is a single line. But don't name some class field or method parameter "i" if you want people to understand its meaning.". Following Dietrich's example, naming simple counter variables after very basic ideas like "i", "x", "count", or "num" is extremely common in many languages. It is not an idea that is attached to one specific language, but rather is a more universal technique to show that whatever

variable is being counted or incremented is not necessarily important by itself; the variable is most likely part of a list or set of variable values that are being cycled through in search of specific results. Because such logic is present in nearly every programming scenario or language, these simple counter variables are encountered everywhere. While there is some diversity in the actual format (usually depending on language), the concept is well known to most developers.

On the opposite end of the variable-naming spectrum, again following Dietrich's example, using naming schemes that highlight the importance of crucial variables can also help to increase the readability of code. While excessive name length or inconsistent casing (such as camelCase vs pascal_case, representing each case respectively with their style) can create some confusion (names that are extremely long can push the seams of the "chunks" you put variables in), finding the middle ground where your variable names are descriptive enough to get the idea across without taking too long to comprehend is ideal.

Dietrich also makes a good point arguing against making code as readable as possible: sometimes the use of complex functionalities that are only known to some of the more advanced users of a language can be required to truly accomplish a task, or at least to accomplish the task with much less effort than struggling to make a huge chunk of code that is somewhat readable in pieces but is a huge mess of lines that lack the use of advanced functions. According to Dietrich (2018), "Code readability is generally preferable, but not in all cases (e.g. you might have need to optimize a critical path where advanced language usage concerns trump readability).". While it may seem antagonistic to the ideas behind this thesis, Dietrich's stance has some merit. Readability is, of course, a strong goal to strive for. But sometimes you must sacrifice a portion

of that readability for the sake of the program as a whole, sort of like an investment in making the rest of the program more bearable from a comprehension standpoint versus adding an unnecessarily large piece of code that only consists of "the easily readables".

Dietrich also adds another layer to the definition of code readability that I believe holds some merit. According to Dietrich (2018) on the topic of code readability, "Thematically, the goal is to make it so that maintenance programmers (you or others) have to spend less time understanding your code." Coming at this from a slightly psychological perspective, coding with others in mind is greatly appreciated by those others who will have to look over your code at some point. This is why best practices are established or begin to pop up among coding communities: There is a good chance that someone else will have to look over your code, and if your code is filled with too many personal style choices and seemingly counterintuitive practices you will only cause strife to befall your fellow programmer. This is essentially the Golden Rule ("Do unto others what you would have done to you.") of the programming world.

While Dietrich's insights were definitely helpful at establish a stronger concept of code readability as a goal, we still have not established any kind of quantifiable metric or system to truly compare code. What if two programmers were given the same coding assignment, used the same language and development environment, spent similar amounts of time coding, created a working program that accomplished the task assigned to them, but they both used practices that were somewhat "against the grain" from a readability perspective? Maybe Programmer A used a strange indentation style that requires lots of jumping around on the page, and Programmer B only used single-character variable names like "a", "b", and "c" where having actual descriptors

attached to the variables would have been heavily appreciated? Who would be declared the "winner", if this was to be a competition? They both reduced the readability of their respective code, but how do you decide what mistakes were more egregious?

It would be complicated (if not impossible) to establish a quantifiable metric of something like code readability. Consider all the different aspects we've discussed so far (different languages, styles, naming schemes, etc.). As I've said before, calling code in one language more readable than code in another language would be a hard case to argue. However, given enough time and resources (such as a large set of feedback data on the use of common programming language) it could be possible to establish a metric for code readability within a singular programming language.

While I am personally lacking in those resources, I managed to find a research report from the Department of Computer Science of the University of Virginia where the researchers, Raymond P.L. Buse and Westley R. Weimer, were attempting to find a correlation between code readability and software quality. In *A Metric for Software Readability*, Buse and Weimer are extremely focused on the concept of readability and how to improve it from a coding standpoint, quoting several other researchers and experts in the field to show the importance of the concept. According to Buse and Weimer (2008), "Readability is so significant, in fact, that Elshoff and Marcotty proposed adding a development phase in which the program is made more readable… Knight and Myers suggested that one phase of software inspection should be a check of the source code for readability...Dijkstra, for example, claimed that the readability of a program depends largely upon the simplicity of its sequencing control…" (p. 121). The fact that they

agree with the proposal of a specific phase in the development of code that is dedicated solely to making sure the code is viable and readable speaks volumes to me.

Clearly Buse and Weimer are interested in defining readability and applying it to code, and they also bring up notions similar to my constant analogies of programming languages compared to written/spoken languages. According to Buse and Weimer (2008), “The Flesch-Kincaid Grade Level, the Gunning-Fog Index, the SMOG Index, and the Automated Readability Index are just a few examples of readability metrics that were developed for ordinary text. These metrics are all based on simple factors such as average syllables per word and average sentence length.” (p. 121). Objective measurement systems to determine how readable a written sentence is clearly exist. It would stand to reason that a similar system could be created for programming languages, as long as this new system had a set of standards to be applied. I cannot think of a more easily applicable set of standards than the official style guides presented by the developers of the languages themselves, such as the PEP 8 style guide for Python. If we could establish a rating system to apply to code, such as a score based on the number of characters in a variable name, a score for the number of nested layers in a function, and other factors that could potentially increase or decrease the readability of the code, we would hypothetically have a working system that defines just how readable a piece of code is.

On continued examination of the works of Buse and Weimer, their team constructed a study to help determine what characteristics help define code as readable (which I have been referring to interchangeably as “good” code.). The professors recruited a team of 120 annotators of code to inspect 100 code snippets of varying length each, for a total of 12,000 judgements on the

snippets of code. All annotators that took part in the study were computer science students of varying levels of experience, with a little over half the participants taking part in 200-level computer science classes, or second year classes. The code snippets were generated from a set of open source projects written in the Java programming language of varying levels of complexity with varying characteristics that could define them as readable or unreadable, such as line length, the presence of documentation or comments, and others. While these may be considered a hefty amount of variables to consider in a study, I believe they help to represent the spread of knowledge and experience that could be seen in a development environment, if not in experience level then at least in variety of experience level. Given that Java is a relatively common language these days, I also find the given language snippets to be within reasonable expectations of comprehension from a language standpoint.

The participants were given the code snippets in random sets of five and asked to order them from most readable to most unreadable. The participants were not given a strict way to decide what snippets were more readable than others to promote individual scorings that were unique and without bias based on the current circumstance. Since the whole point of the study was to determine what code was most readable, they did not want to skew the results.

The participants would then order the snippets in the given set from most readable to least readable, and the final results were compared ordinally, not nominally. That is to say, it was more important to the research team to discover what snippets were considering more readable than others within the same set than it was to see what snippets were the most readable out of the entire 100. According to Buse and Weimer (2008), "...absolute score differences are not as

important as relative ones. If two annotators both gave snippet *X* a high score than snippet *Y*, then we consider them to be in agreement with respect to those two snippets, even if the actual numerical score values differ.". I appreciate the use of this style of comparison as it helps to cross-reference the choices of many programmers on what snippets were deemed most readable and then allowed for inspection of those cross-referenced snippets to figure out what about them made readable, while also allowing the participants to focus on small innate comparisons between a handful of code snippets. Asking the participants to examine and rank all 100 code snippets in order would be a near considering the number of interlocking levels of complexity that would have to be compared. Breaking it down into twenty sets of five allows for similar styles of comparison without the difficulty brought about by sheer numbers.

Once the participants have completed their assessments of all snippets, the results were run through a machine learning algorithm to determine what code snippets were the most readable and to help determine which singleton characteristics in those most readable snippets were most influential on the readability. There were 25 different characteristics that the research team were accounting for as potential influencers of code readability, and among those they found that the most influential characteristics (the average number of unique identifiers in a snippet, the average length of a line of code in the snippet, and the average number of parentheses or brackets per snippet) and all negative predictive power on the readability of the code. To put it into layman's terms: the more new variables you identify in a section of code, the longer your lines of code are, or the more parenthetical clauses of code you include, the harder you code gets to read.

While these may seem like relatively obvious statements to make, I want to hypothesize what

about these characteristics makes the lines of code harder to understand. There were several other characteristics tested for in this study, but considering these had the most predictive power in determining the readability of the code, I will just focus on these top three.

The characteristic with the most predictive power was the number of identifiers in a snippet of code. There was a strong negative impact on the readability of all snippets with a higher than average number of identifiers. I believe there is a strong connection between this result and the idea of the Miller's Law, or The Magic Number, which was brought up in Egon Elbre's article *Psychology of Code Readability*, referenced earlier in this paper. Miller's Law is essentially the idea that the average person can only truly focus on about seven different objects at a time, give or take about 2 objects in either direction. There is only so much energy that our brains can commit to active observation of our surroundings and our thoughts. So much of what our brains do is subconscious or instinctual (blinking, telling you to scratch that itch on your arm, etc.) that I think people do not always realize how little they can truly focus on at once. Even the combined of working on an essay while also occasionally checking out your bedroom window to see if your friend has arrived takes a lot of active effort, and if you've ever been in charge of a major event like a wedding or a surprise party than you know how many different facets you have to pay attention to and keep in line to make things go smoothly.

The same logic applies here to how many variables your brain can focus on at once, especially considering the participants in this study were unfamiliar with the code at best and may not have even know what the code was supposed to do if the snippet was given to them without much documentation attached. A proper naming scheme that follows its own rules and restrictions

could alleviate some stress, but even then it does not guarantee comprehension in the mind of the reader. As the number of new identifiers increases (which in the case of Java code could include variables, classes, packages and other structures), the amount of "brain power" available for focusing on all these identifiers get spread thinner and thinner. If things get spread too thin, you can begin to lose track of what each identifier means or what it relates to and interacts with. As a result, the likelihood of comprehension of the code decreases.

The second most influential characteristic was the average length of a line of code, also with a negative impact on the readability as the size of the line increased. This could likely be given a similar explanation to the average number of identifiers. Longer lines of code will presumably have more complex functionalities, such as nested functions that use their return values as parameters in other functions. This can also lead to lower levels of readability, considering the fact that you can only focus on and keep track of so many concepts and values at the same time. Each line of code can be as long or as short as the developer pleases (in most languages), though many style guides will give a recommended maximum character limit to help in limiting the amount of confusion. Similar to how functions are usually supposed to have a main purpose or interaction with a program, it can be beneficial from a readability standpoint for the developer to create lines of code that only accomplish one major change at a time. For instance, instantiating a variable and changing that same variable by using it as a parameter in a function within the same line could create some confusion for the reader as to the actual value or content of that variable. Again, much of this is up to personal preference and who will be reading the code. If the program can be made to run faster by utilizing some of the more advanced features of a language at the cost of readability, there can almost always be an argument made for one stylistic choice

over the other.

The third most influential characteristic in the study was the presence of parentheses (()) and brackets ({}), with a negative correlation with readability. I feel that this relates heavily to the points I made on the previous characteristic. In most languages, Java included, the presence of parentheses or brackets indicates the wrapping of a function, it's parameters, or it's logic.

Curled brackets are used to denote the beginning and end of a function's logic, while parentheses can be used to encompass parameters or other values to be denoted as separate from the rest of a line, perhaps in the instance of keeping order of operations clear. This is actually a rare case in which I would consider parentheses to actually be a useful feature, opposing the data brought forth by the Buse and Weimer study. Using parentheses to denote that values are correlated and will interact, such as in the line $x = (y+z) * 4$, can help clarify any potential confusion on a cursory reading of the line of code when it comes to the intended order of operations. If a developer were coding hastily, they could potentially gloss over the correct implementation of the math, resulting in unintended behavior within their code. However, given my experience with Java, you are more likely to see parentheses used to denote the parameters of a function or object than you are to see them being used as a mathematics clarifier.

Back to the main point, according to Buse and Weimer's study, the presence of brackets and parentheses has a negative impact on readability, which increases as the number of brackets and parentheses increases. This is once again similar to the concept of the Magic Number and how we can only focus on so many concepts at once. Keeping track of a variety of variables at once

can be confusing, but considering parentheses usually come into play when functions are being called and interacted with, this only increases the complexity of the situation at hand. Keeping track of what function has which effect on which variables can easily get out of hand.

So it would seem that many of the most influential features and characteristics of code (in the case of this study Java code, but many of these concepts apply to most common languages) have at least somewhat of a connection to concepts of the ability to focus and comprehend multiple concepts or lengthy structures. While I do not consider the study done by Buse and Weimer to be comprehensive or indicative of the problems of code readability as a whole, I think they bring to light some interesting conclusions based on a well put together research study.

As an interesting side note, Buse and Weimer included the entire scale of the twenty-five code features they were gathering data on, including the features that had the least influence on the readability of the code at hand. There was only one feature that they found had little to no bearing on readability, and that was the average length of the identifiers in the code. It didn't matter if the variables or function definitions were 1 character long or 30. While there are a lot of variables to consider here (the participants were active computer science students, they had never seen the code snippets before the study and were therefore unfamiliar with the design choices of the developers, etc.), I do find it interesting that there were no variable or function names that were considered too lengthy and distracting. I was always under the impression that keeping your variable names down to a short length while also making them descriptive was the goal, and considering the idea of a "short variable name length" is a subjective idea, I would have figured there would be some predictive power in the code snippets with identifiers of varying length.

After spending several months researching the concepts of code readability and what made for good code or bad code, I decided that I wanted to construct a small study of my own in a language I was more familiar with. In a similar fashion to the study done by Buse and Weimer, I wanted to see what aspects of the Python coding language people found more readable than other aspects that provided the same level of code functionality. As defined earlier in the PEP-8 style guide, there are obviously many well-known stylistic choices that are recommended to the Python development community, such as keeping your line length to 80 characters or less.

However, I would consider Python 3 (the most recent version of Python) to be one of the more flexible programming languages when it comes to syntax and stylistic choice. There are often multiple ways to get the same result out of a piece of code, be it from old functionalities left in from old versions of Python or just built-in quality-of-life functions, such as not needing to instantiate your variables as a specific data type since the type is apparent based on the content related to the variable (e.g. *x = 4* clearly refers to x being equal to the integer 4, while *y = '4'* defines y as a character or string with the ASCII representation of the number four.). Python is also commonly known as one of the more easily readable languages out there because of its use of coding structures that are nearly pseudo-code, requiring fewer levels of abstraction to comprehend.

Because of this innate flexibility, I wanted to see what members of the Python community considered to be the best implementations of the functionalities at hand. My advisor, Dr Andrew Berns, helped me construct a short survey consisting of small portions of relatively simple

Python code. Each of the nine survey questions consisted of two snippets of code that had slight differences in structure but ultimately resulted in the same output when run. Participants in the survey were then asked to choose which of the two code snippets in each question was most readable in their personal opinion. The question was left simple and non-descript to allow the participants to create their own definitions of what "most readable" meant. After each choice, survey participants were also encouraged (though not required) to explain why they chose the code snippet they did in a comment box made available to them.

The survey was administered on the Amazon Mechanical Turk (often shortened to MTurk) virtual marketplace site. Posters on MTurk are allowed to post surveys and give financial compensation for participation in activities like academic studies, code reviews, and other technological scenarios that require human interaction or observation. Dr. Berns and I agreed that allowing anonymous online users would allow for a more diverse pool of answers that would avoid any potential interference or skewing of the results that could arise through interaction with other students at UNI.

It should be noted that we also anticipated there to be at least some dishonest or untrustworthy participants in the MTurk survey, and therefore some survey participants results were removed from the final results for various reasons, such as if they chose not to answer a specific question, their comments were nonsensical or seemed otherwise untrustworthy, resulting in the questions having varying numbers of confirmed answers. The goal was to have up to 100 participants take the survey, and as of the time of examining and analysing the survey results 97 people had taken part. Condensing ninety or more comments into a concise decision would be near impossible for

this type of survey, so instead any attempt to consolidate and interpret the comments left by the survey participants will be a best attempt at presenting the most general description of the comments and the intentions behind them.

Question 1 provided a list of the colors of the rainbow and asked participants if they preferred to print each color in the list by making a for-loop of the list and printing each color as it was iterated over within the for-loop (print(color)), or if the user preferred to created a for-loop based on the length of the list of colors and printed each color based on the index within the color list (print(rainbow[x]). Of 90 confirmed answers, 62 participants (68.9%) deemed print(color) to be the preferred style for implementation and 28 participants (31.1.%) deemed print(rainbow[x]) to be better suited. The first choice's comments usually contained messages about the chosen syntax being shorter, easier to read, or involving less punctuation/parentheses, thus making it the better choice.

Question 2 gave the user a scenario in which they could calculate the total cost of buying a number of hamburgers at a restaurant. Choice A gave proper names to all variables introduced (burger_cost = 5, three_burger_discount = 1, tax = 0.07) and Choice B gave replaced all variable names with non-descriptive characters (x = 5, y = 1, z = 0.07). Out of 91 responses, 47 chose Choice A (51.6%) and 44 chose Choice B (48.4%). Comments left by those who chose the more descriptive Choice A said the code made more sense to them when the variables had easily relatable names, while participants who chose Choice B said the more algebraic method was more intuitive to them.

Question 3 proposed a simple scenario where, if the variable *yesterday* equaled the phrase "Thursday", the program would print out "It's Friday!". The only difference in the choices given to the participants was that Choice A had the print statement on a separate line (common practice in Python), while Choice B had all code on one line. Out of 91 responses, 47 chose Choice A (51.6%) and 44 chose Choice B (48.4%). Several comments left by those who chose Choice A stated that they believed the separated code was better because it was how they were trained to read Python or because split code was easier to read. However, there were also many comments on the Choice B side that stated keeping code as concise as possible was the better stylistic choice.

Question 4 asked the reader how they preferred to read an if-statement containing the "not" negation syntax, with a slight change in grammar between the choices. Out of 96 responses, Choice A (if x is not None) got 62 votes (64.6%) and Choice B (if not x is None) got 34 votes (35.4%). Comments on Choice A said that it was the closest to "normal English" or that it made more sense to the reader.

Question 5 was originally intended to be an example of unnecessary indentation, where the code between Choice A and Choice B was identical except for a print statement that was indented more times than it needed to be by Python standards. However, the formatting of the question was accidentally put into the MTurk system incorrectly and the intended purpose of the question was rendered unidentifiable. Therefore, the data was deemed irrelevant to the study.

Question 6 provided a list called my_list that contained the letters A, B, C, D and E and provided code that would reverse the order of the letters (resulting in E, D, C, B, A). Choice A (my_list[::-1]) made use of the "backwards step" functionality of Python lists, indicated by the -1 value, while Choice B (reversed(my_list)) used the built-in reversed() method. Out of 95 responses, Choice A got 45 votes (47.4%) and Choice B got 50 votes (52.6%).

Question 7 provided a string ("ABCDE") and two ways to remove the "C" character from the middle of the string. Choice A used the replace() method to replace the "C" with an empty string, and Choice B rewrote the original string by breaking the string into 3 pieces without the "C" and combining the two ends of the string. Out of 95 responses, Choice A got 65 votes (68.4%) and Choice B got 30 votes (31.6%). Many comments stated that Choice B was needlessly complex or was not feasible in all coding situations.

Question 8 provided two ways for a list of perfect squares with bases from 1 to 10 to be generated. Choice A used a for-loop from 1 to 10 and multiplied the current iteration by itself (1 * 1 = 1, 2 * 2 = 4, 3 * 3 = 9, etc.), while Choice B utilized the list comprehension structure, a slightly more advanced format of generating values for a list with less lines of code, in a similar style. Out of 95 responses, Choice A got 46 votes (48.4%), while Choice B got 49 votes (51.6%). Comments that were pro-Choice A stated that having the code separated out in a for-loop format made the code easier to read, while pro-Choice B comments stated that having less lines of code made Choice B the easier code snippet.

Finally, Question 9 provided a list of the numbers 1 through 5 and provided two ways of creating

a new list that contained the inverses of the numbers in the first list. Choice A utilized the map() function to run the original number list through a pre-made inverse function, while Choice B ran the original list through a for-loop that just appended the inverse of each number into the list of inverses, creating the inverses in the append statement. Out of 96 responses, Choice A got 56 votes (58.3%) and Choice B got 40 votes (41.6%).

While many participants left the comment section blank or left comments that were not exactly legible or relevant to the task at hand, their choices do indicate some potentially interesting findings. As the intention of the survey was to allow participants to choose their preferred method of reaching a common end goal, there was little to no variation in the choices except for the intended stylistic or functional choices. So even though there was only one significant difference in each set of code snippets, half of the questions had a nearly fifty-fifty split in preferred execution while the other half had closer to a ⅔ split (ignoring the irrelevant data provided by Question 5, resulting in 8 questions with usable data). Many commenters on either side of any question had extremely similar comments, stating that their specific style was more easily readable or made for better code. This is a prime example of the different mentalities that can form in development environments, even among peers or developers that consider themselves experienced in the same language.

Something else worth noting was that complexity seemed to be a highly correlative factor among the questions that had a ⅔ split in their votes (Questions 1, 4, 7 and 9). Some of the choices given were considered needlessly complex by a larger portion of the participants, and the comments showed that they thought this to be true. While the subject matter of each question varies at least

slightly, there does seem to be a somewhat stronger tendency for the survey participants to identify what they consider to be complex segments of code and to shy away from these complex segments. While complexity of code can sometimes correlate to more efficient code that is utilizes the more advanced functionalities of a language and is just harder to read, it can also just mean that code is being used inefficiently in comparison to other functionalities that already exist, such as methods or built-in functions.

On the other hand, the questions that had closer to a fifty-fifty split (Questions 2, 3, 6, and 8) had comments almost entirely focused on stylistic choice as opposed to the complexity of the code itself or the length brought about by the code's complexity. While code that is broken up over multiple lines (e.g. putting print statements on a separate line as per most Python style guides) does take up more space, it is almost always for the sake of readability and ease of access. This again highlights the common struggle over deciding to make code concise, complex and quick as opposed to making it larger and easier to understand at the cost of program speed.

Regardless of how anyone defines "good code: or "bad code", whether it be by line length, quantity vs quality in line contents, increased complexity for the sake of run time efficiency or vice versa, or any other commonly debated subject in the development world, I think it is very obvious at this point that personal preference plays a heavy hand in the decisions that get put it into the code that gets developed.

As with any subject being taught, the teacher was originally a student at one point taught by an imperfect mentor, and then they developed their own practices and distinct style that they most

likely imparted at least some of onto their students. Considering programming is such a diverse subject with thousands of styles, rules, and guidelines to implement across any language that you can work with, there is bound to be some “drift” to what is considered an adequate level of quality in the code, even within each individual language.

The scope of the survey we conducted definitely had it’s limitations and restrictions, especially considering the formatting errors of Question 5, and the amount of time and resources at hand throughout the process of research and analysis. but I think that the information it provided was definitely valuable to an extent.

I’m hesitant to say that this thesis had concrete findings, if any. The questions I was looking to answer (What makes for good code and how can we improve coding?) were very much so open-ended and therefore hard to truly answer empirically. I definitely believe there are definitely some practices that could be employed to help developers understand what about coding is complicated and how they can change their patterns to create an easier future in development for themselves and the others who will read their code. For instance, having an understanding of how the brain works from a psychological perspective could do wonders for people trying to make their code more readable. With concepts like The Magic Number in mind, you can actively work to make sure your code does not try to accomplish too many tasks at once. If you keep a strong mental model in mind, imagining what this code would like in a physical space, you can more easily keep track of details and variables and interactions between those variables, which can then be translated into code that has a strong foundation in reality. Thus, even when people who are unfamiliar with your coding style interact with your code, they can utilize the shared

knowledge that most people have of the world (such as knowing that a book will commonly have a table of contents, chapters, page numbers, etc.) to decode your intentions and fully comprehend the tasks you were trying to accomplish.

Also, the use of programming languages (or naming schemes within those languages) that utilize language that is commonly understood in a written or spoken circumstance can also help considerably with making code more readable. All programming languages and their parts are some form of abstraction on another (spoken) language, such as the list data type in Python allowing a person to make an ordered list of elements. Developers will have already experienced some form of a physical list in their lives, such as a grocery list, therefore giving them some basic connection to the meaning and usage of the Python list.

Finally, personal preference and diversity in programming languages and styles definitely play a strong part in what code developers write, and no single developer will ever know all the styles and rulings that all other developers follow. The development of personal stylistic choices, combined with the hundreds of different languages that were specifically created with a certain goal in mind (and therefore utilized syntax and formatting specific to the people who would interact with the language most), has created an intricate web of confusing syntax and programs that would be considered messy to some and pristine to others. In my opinion, this is not a bad thing. The diversity and specialization of languages allows for more productivity and specificity in what developers need to know. Every developer does not need to know how to make a window in Java, just as much as every programmer does not need to know how to design a neural network. While having a common language could be appreciated, I believe it is more so

important that programmers know the basics and logic that inspires the backbone of languages. You can make a binary tree in many languages, but understanding how to write the tree out on paper allows you to better learn how to program the tree out in multiple languages.

So while I may not be able to absolutely nail down what needs to be done to create the “best” language, I knew that such a goal was pretty unachievable. It has been much more insightful to focus on what aspects of coding are more important to writing good code in general, not just in a specific, undetermined language. Figuring out some potential reasons for having the languages we have today and how they could be improved will definitely prove vital in my own career, and I hope it can help others find some clarity in the more confusing spaces of software development and maintenance.

References

Buse, R. P. L., & Weimer, W. R. (2008, July 20). A Metric for Software Readability. Retrieved from https://web.eecs.umich.edu/~weimerw/p/weimer-issta2008-readability.pdf.

Dietrich, E. (2018, January 24). How Do You Evaluate Code Readability? Retrieved from https://blog.submain.com/evaluate-code-readability/.

Elbre, E. (2018, July 21). Psychology of Code Readability. Retrieved from https://medium.com/@egonelbre/psychology-of-code-readability-d23b1ff1258a.

Sajaniemi, J. (2008). Psychology of Programming: Looking into Programmers' Heads. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments, 4*(1), 4–8. doi: 10.17011/ht/urn.200804151349

van Rossum, G., Warsaw, B., & Coghlan, N. (2001, July 5). PEP 8 -- Style Guide for Python Code. Retrieved from https://www.python.org/dev/peps/pep-0008/.

Young, S. (2014, November 3). Why your code is so hard to understand. Retrieved from https://medium.com/on-coding/why-your-code-is-so-hard-to-understand-83057c115a2b.