UNIVERSITÄT HEIDELBERG
ZUKUNFT
SEIT 1386

Influence of identifier length and semantics on the comprehensibility of source code

A THESIS PRESENTED

BY

JOHANNES C. N. HOFMEISTER

TO

THE DEPARTMENT OF PSYCHOLOGY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

BACHELOR OF SCIENCE

IN THE SUBJECT OF

PSYCHOLOGY

UNIVERSITÄT HEIDELBERG

NOVEMBER 2015

Author:

Johannes C. N. Hofmeister

Rohrbacher Str. 45

69115 Heidelberg

Germany

Matrikelnummer / Student number: `30 400 19`

E-Mail: `bachelor-thesis@you.cessor.de`

Twitter: `@pro_cessor`

Web: `http://jcn.hm`

The source code and data used in this thesis are available on Github:

`http://github.com/cessor/bachelor-thesis`

Primary Advisor: Dr. Daniel Holt, University of Heidelberg, Germany

Secondary Advisor: Dr. Janet Siegmund, University of Passau, Germany

Thesis advisor: Dr. Daniel Holt Johannes C. N. Hofmeister

Influence of identifier length and semantics on the comprehensibility of source code

Abstract

Identifiers are essential for the understanding of source code. Programmers are at liberty to name them arbitrarily, which is a major source for hard-to-understand code. This thesis investigates how an identifier's length semantics affect code comprehensibility. We suggest a cognitive model for code comprehension. An experiment was conducted to test aspects of this model. Participants saw codes with different identifier types (normal words, abbreviated, and single characters) and the time for finding a defect was measured. We could show that identifiers, which were named using proper words, lead to faster defect detection than identifiers using abbreviated words or single characters.

Contents

List of Figures

List of Tables

List of Code Listings

Für den modernsten aller Menschen.

Acknowledgments

I'd like to thank all the people who supported me: Andy, Antony, Antti-Juhani, Beate, Bernd, Daniel, Golo, Horst, James, Janet, Katha, Lars, Marcel, Marion, Stefan, Tala. Special thanks to Katha for always having my back. Mum & Dad for supporting me. Lars for DevOps and visualization support. Daniel for his time and doulaing. Tala for helping me focus. Horst for proofreading. A big 'thank you' goes to all the people who shared, forwarded and participated in my study.

I will use the term "we" to refer to myself, although I am just a single author. I feel that it makes for a more comfortable reading experience. I use the Python 2.7 programming language in my explanations, because its reduced syntax helps to emphasize crucial points and prevents distractions. *Italics* in paragraphs refer to items in the glossary.

There are only two hard things in Computer Science: cache invalidation, naming things, and off-by-one errors.

Phil Karlton, Tim Bray, Leon Bambrick

1 Introduction

Source code isn't static and changes during a *program's* life-cycle. It has to be modified when specifications evolve, e.g. when a feature has to be added, or a part of the code that doesn't function properly, needs to be repaired. This type of maintenance work is common (Brooks, 1983; Pennington, 1987; Tiarks, 2011) and happens after the program has been deployed. It is often performed by programmers other than the original author. Changing code is only possible when programmers understand the consequences of their alterations, otherwise they might introduce defects (Liblit, Begel, & Sweetser, 2006; Pennington, 1987; Rajlich & Wilde, 2002). Thus, programmers need to understand how a program works, to identify relevant parts that are subject to change. Building an understanding of a program's functioning is called *code comprehension*. With this thesis, we would like to study code comprehension in order to improve software maintenance tasks. Because comprehension is a major task during maintenance, a better understanding of the underlying process could help in performing it more efficiently, thereby preventing mistakes and reducing development costs (Boehm, 1981; Siegmund, 2012; Siegmund & Schumann, 2015; von Mayrhauser & Vans, 1993).

We'll focus on two properties of identifier names, namely their **LENGTH** and **SEMANTICS**, and how they affect code comprehension. *Identifiers* are particularly interesting for code comprehension, because they make up major parts of source code and can be named freely (Lawrie, Feild, & Binkley, 2007). According to Deissenboeck and Pizka (2006), approximately 72% of characters in source code occur in identifiers, and they account for 33% of all *tokens*. Comprehension of source code relies mostly on the information encoded in its *identifier names*. Caprile and Tonella (2000, p. 97) called identifier names "one of the most important sources of information about program entities", however there is no guarantee that their names encode relevant information. Identifiers can be named using meaningless strings, which can make code hard to understand.

To find out how programmers are affected by meaningless names, we conducted an experimental study, in which we measured how fast programmers could find defects in code, depending on the length and semantics of identifier names. We found that performance in code comprehension tasks is affected by identifier names.

Before we describe our study, we'll explain why code comprehension is affected by identifier names, and why length and semantics are particularly important in this matter. We'll start by outlining a cognitive model of code comprehension.

1.1 A model of code comprehension

Our model is loosely based on the works of Blum (1985). Blum separates the software production process in **APPLICATION SPACE**, **IMPLEMENTATION SPACE** and **PROBLEM SOLVING SPACE**.

Application space defines problems in the real world and their relationship with the software product. Implementation space contains all knowledge of a computer system that is required to solve a problem. Both spaces can overlap. Decisions for programming are performed in problem solving space. The latter defines the transformations between application and implementation space through cognitive processes. In this internal space, the two external spaces

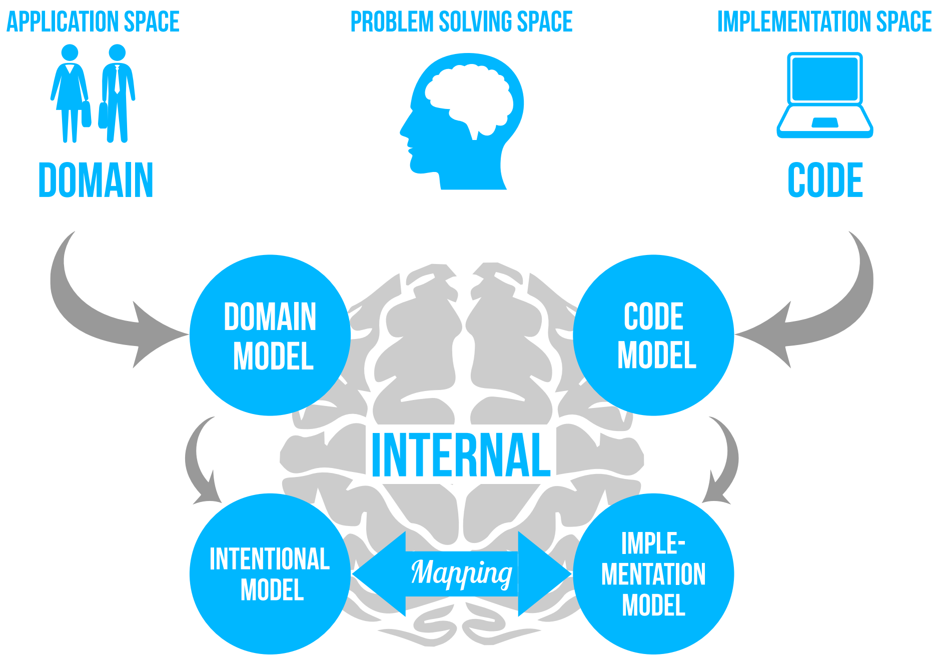


Figure 1.1: A programmer's mental representations of code in relation to the real world. The circles indicate cognitive models.

(the non-technical application space and the technical implementation space) are connected with each other. Source code is an element of the implementation space, whereas application space contains specifications, which are informal descriptions of real world problems. Programming itself can be seen as a cognitive process, which transforms problem statements between application space and implementation space, and is running in problem solving space. Figure 1.1 illustrates the three spaces, as well as involved cognitive elements and processes.

Programmers represent application space and implementation space as **COGNITIVE MODELS**. These cognitive models are simplified, reduced subsets of a programmer's reality and consist of mentally represented *entities*, their relations and processes, from the the individual spaces. The two external spaces are represented internally as a **CODE MODEL** and a **COGNITIVE DOMAIN MODEL**. The cognitive domain model contains all knowledge about the requirements and use of the implementation. It is a subset of all information from the actual domain in application space. The code model mirrors the implementation of the source code and contains all knowledge about its functioning, architecture and structure. It is an abstract model of the available source code. Both models are representations of their associated media (e.g. a code file, a specification). Individually, they contain too little information to allow an integration of both spaces. For this reason, two specific models are derived that facilitate such an integration: the **INTENTIONAL MODEL** and the **IMPLEMENTATION MODEL**.

The intentional model contains the entities of the real-world domain that are relevant for the implementation. It is a simplified view of the cognitive domain model and contains only representations of entities which are supposed to change during programming. From the code model an implementation model is derived, which augments the code model with acquired, semantic information, which is extracted from source code and other parts of the system.

The activity of programming connects the implementation model and the intentional model with each other. This happens through a process of **MENTAL MAPPING**, which associates elements from the intentional model with elements from the implementation model. The resulting associations are used as heuristics to identify target states during problem solving. Therefore, performance during maintenance tasks should be affected by how easily these associations are (mentally) created, maintained, manipulated, and recalled. In order to associate entities in code with elements from the domain, programmers rely on **IDENTIFIER NAMES**. If this is not possible, then the meaning will be deduced from the identifier's usage in the program.

To understand how identifiers facilitate code comprehension, we will now briefly explain what identifiers are, and how they can be used.

1.2 Identifiers

Programmers encounter parts of the code which are fixed by a programming language's grammar, such as *keywords* (e.g. `for`, `while`), and special characters (e.g. `[]`, `{}`, `&`, `+`, `-`, `=`, `%`). The rules on how these symbols have to be combined are checked during *compilation*. Other parts of the program, namely *comments* and *identifier names*, are not fixed, and are only analyzed by the compiler for their syntactic validity. When used properly, comments can facilitate code comprehension (Nurvitadhi, Leung, & Cook, 2003; Salviulo & Scanniello, 2014; Shneiderman & Mayer, 1979), but because they are only loosely connected to parts of the program, our research concentrates on identifier names.

Identifiers are character strings in code that are used to *name* and *identify* a program's entities. Compilers will allow any string to be used as an identifier name, as long as it complies with the lexical rules of the language. For example, consider the grammatical rule from the Python (2.7) programming language in Listing 1.1.

```
identifier ::=  (letter | "_") (letter | digit | "_")*
```

Listing 1.1: The lexical rules for identifiers as defined in the Python 2.7 grammar.

This rule states that an identifier must start with a letter or underscore, and may contain digits, but no special characters or whitespace. Listing 1.2 shows a *function* with a valid identifier named `fibonacci`. The rule controls the allowed characters for an identifier, but not the words built with them.

```
def fibonacci(): pass
```

Listing 1.2: A function definition that complies with the rule, as the identifier `fibonacci` contains no special characters.

The function could not have been named `ret%rieve results`, because the given rule does not allow spaces or special characters, but it could have been named `a`, `retrieve_results` or `superman`. Consequently, when writing code, programmers have a lot of freedom (Binkley et al., 2013; Carter, 1982; Liblit, Begel, & Sweetser, 2006).

In practice, programmers don't just use random strings as identifier names [1] but they mostly agree that identifiers should be named after 'a thing' or purpose. By naming identifiers after relevant 'things', the entities in the code are associated with the programmer's understanding of these things. An identifier is therefore not only a syntactic element, but also a symbol that associates a *concept* with an *entity* in the code. For example, naming a *class* `Person` will associate the concept of a person with the class in the code.

Identifiers fulfill two main purposes: **DESIGNATION** and **IDENTIFICATION**. Designation describes the aforementioned association between a concept and a word (Newell, 1980), used as an identifier name. Identification is the process of referring to an entity in code in order to evaluate it (e.g. calling a function or retrieving a *variable's* value). Identification is relevant to the machine, whereas designation allows a programmer to reason about a concept. Identifiers

[1] Usually, there are code conventions and style guides in place, see for example (Free Software Foundation, 2015; Microsoft Corporation, 2005; Sun Microsystems, Inc., 1997).

are structures that have a name, and thereby designate concepts. Because they are interpreted by machine and cognitive processes, they can be understood as *representations* (Newell, 1980, pp. 176).

Ideally, the concept designated by an identifier is reflected in its name. Most programmers hold the opinion that identifiers should have one or more of the following qualities: meaningful, descriptive, informative, clear, concise, understandable, correct, useful, should convey information (Caprile & Tonella, 1999, 2000; Dit, Guerrouj, Poshyvanyk, & Antoniol, 2011; Lawrie et al., 2007; Liblit et al., 2006; Ottinger, 2009). These attributes can be understood as valuations of how clearly an identifier's name designates a concept. Because programmers are at liberty of choosing a name, identifier names don't have to resemble concepts. It would make sense to name a class that represents a *Person* with the identifier name `Person`, but it is possible to name the class `Contact`, `P` or `DataInfoProvider`. The identifier will still represent the concept (in the context of the program), although its name does not reflect this relationship. This makes identifiers susceptible to various problems, such as *homonym* and *synonym* defects (Rațiu, 2009, p. 149). It is therefore possible to distort the designation of an identifier (by changing its name) but leave its identification intact.

During programming tasks, programmers build associations between concepts and entities in the code, thus aligning their cognitive implementation model and intentional model with each other. The associations can either be inferred from the entities' use in the code, or they are given away by the identifier names. It is possible for programmers to relate arbitrary entities in code with concepts from the real world (e.g. `f` with *Person*), but this will strain cognitive resources more, than when identifier names are congruent with the associations. We will now discuss how identifiers affect code comprehension from a psychological point of view.

1.3 Syntactic Aspects

Psycholinguistic research names various effects that can be transfered to code reading. In an fMRI study, in which participants tried to comprehend different pieces of code, or find syntax errors within them, Siegmund, Kästner, Liebig, Apel, and Hanenberg (2014) found activations of cortex areas which are linked to natural language processing. Caprile and Tonella (1999) examined the monogram frequencies of 10 programs with a total of 230,000 lines. They found that the character distribution in code is very similar to natural language English. These findings suggest that code reading be analyzed using psycholinguistic methods. However, findings that apply to natural language texts might not be as obvious and clear when reading source code [2]. We'll now see what specific properties identifiers in code share with normal words, and how they affect reading.

1.3.1 Word length

The most obvious surface property of a word is its **LENGTH**. A naïve approach would assume that shorter words are simply 'less to read' and are therefore easier to process than long words. A single character, would then be preferable to a longer word. For various reasons, this is not the case. No one would dare stripping down long texts to single letters[3]. However, word length is not irrelevant during reading but its effect is context-dependent.

Word length influences the processing speed during reading tasks, especially when many non-words are present. Balota and Chumbley (1985) conducted an experiment in which it took participants longer to pronounce long character strings than shorter ones. This effect can be explained by models of lexical access, for example with the Dual-Route-Cascaded-Model (DRC) as proposed by Coltheart, Rastle, Perry, Langdon, and Ziegler (2001). The DRC describes two routes of word-recognition, a lexical and graphemic-phonetic route.

Weekes (1997) found that the number of letters in a word can have an effect on their naming latency (i.e. the time to pronounce a shown letter string). He showed that the number of letters affects the naming latency for non-words. For low frequency words this effect also occurred, but was much weaker. He also found that low-frequency words have a higher naming latency than common (i.e. high-frequency) words. Controlling for the ease of articulation, he rejected the idea that the latency increase with non-words might result from the fact that they're harder to enunciate, but rather that their phonetics must be synthesized on the fly. Weekes predicted these effects based on the assumptions of the DRC. He took them as a confirmation of models that suggest multiple routes for the lexical access of words (like the DRC does). He rejected models

[2] See, for example, Binkley et al. (2013), who made this laborious experience.

[3] Reading 'Harry Potter' would be a much less pleasurable experience under such circumstances.

that propose a single, uniform process of lexical access (Weekes, 1997). Juphard, Carbonnel, and Valdois (2004) underlined that word length effects aren't merely artifacts of lexical decision tasks (word or non-word naming) but that word length actually affects the speed of reading non-words.

The naming delays during these tasks can be explained with the DRC like this: Common words (i.e. words that frequently appear in natural language) are stored in the internal lexicon and can be accessed via the lexical route, when they are perceived as a whole. Non-words don't exist in the mental lexicon and therefore can't be accessed directly. Their pronunciation has to be synthesized by matching their graphemes with the appropriate phoneme-patterns. This is also the case for low-frequency words. The naming latency is relative to the word length, since this process must be serial in nature. The more characters have to be processed, the longer it will take to access the phonetics of a word (Weekes, 1997). In conclusion, it appears that real words are faster to read than fake words. The length of a word is only relevant when it is uncommon or a non-word.

Word length also interacts with memory. Baddeley, Thomson, and Buchanan (1975) found that participants performed better at remembering lists of short words in comparison with lists of longer words. This effect is called the **WORD LENGTH EFFECT** and led Baddeley to question the predominant model of short-term memory, thus leading to its extension with several components, notably the process of sub-vocal rehearsal and the phonological loop (Jalbert, Neath, & Surprenant, 2011).

The effect in Baddeley's experiment occurred independent of how the stimuli were presented (visually or auditive). Participants were instructed to *verbally* recall presented word-lists after a short period of time. This might lead to the conclusion that the effect is amplified by the speech act and it might be that it only occurs because longer words take longer to pronounce. Similarly, short words benefit from sub-vocal rehearsal and can be repeated quicker during the same timespan than lists of longer words.

However, Jalbert et al. (2011) showed that the effect is better explained by a word's orthographic neighborhood. An orthographic neighbor is a word of the same length as a target word where one character has been changed. In their study they showed that words with many orthographic neighbors are easier to recall than words with few. This applies to shorter words (e.g. 'see' has the neighbors bee, gee, lee, sea, set, sew, she, and tee), whereas longer words tend to have fewer neighbors (e.g.: 'house' has horse, louse, mouse). The neighborhood size appears to account for the better recall, rather than the actual length (syllables or character count). The explanation that sub-vocal rehearsal improves recall is therefore insufficient and the effect is more likely to be linked to the phonological output buffer (Jacquemot, Dupoux, & Bachoud-Lévi, 2011). Independent from its true origin, the effect is quite stable and reproducible.

From the word length effect we deduce that very short identifiers are easier to process than long ones. An identifier's name can't be shorter than a single character. For this reason, one could argue that code with identifiers that only consist of single letters is faster to read and process.

1.3.2 Word superiority

Yet, this theory is not tenable, due to a finding called the **WORD SUPERIORITY EFFECT**: Letters are easier to detect when they are embedded in real words than when they're on their own or embedded in nonsensical words (Reicher, 1969; Spalek, 2010; Wheeler, 1970). In code, programmers might try to reduce their typing work by using abbreviations. However, abbreviated identifiers and also acronyms are non-words unless they're so common that the word-frequency negates their non-word nature. For example, 'Cfg' or 'Config' are common abbreviations for 'Configuration'. The time to detect them should be increased in comparison with properly written word identifiers. This would mean that, when many abbreviations or non-words are present, the performance in maintenance tasks will be decreased.

The same should hold true for identifiers that are formed from single letters or (seemingly) arbitrary character strings. Identifiers like `genymdhms` (Ottinger, 2009) or `x`, `y`, `f`, `g`, `h` should take longer to read than terms like `request`. Reducing identifiers to the shortest possible length will impede word recognition due to the word superiority effect and the word length effect and decelerate reading speed.

Although words are faster to read, single-character identifier names, acronyms, and abbreviations are common in everyday code (Carmack, 1999), and some people explicitly defend their use (see, for example, Seeman, 2015). Reasons might include the desire to type quicker or that the code is supposed to be generic and abstract, for example when defining a mathematical algorithm. When code is implemented in accordance to a mathematical formula in a scientific paper, a programmer might choose to name his variables after described mathematical variables. In such formulas, short Greek or Latin variable names are common and often conventions are applied (e.g. a cryptographic function might reference the cypher `c`, the message `m` and so on). Since words are quicker to read than individual characters, reading code must be slower when many single letter variables are present.

Reading code requires visual word recognition, which is slowed down by the use of non-words. Because their phonetics can't be queried from the internal lexicon, the naming of non-words is much slower, as the quoted studies show. Proper words are easier to process because they are available in the internal lexicon and can be accessed together with their semantics. Transferred to code, this leads to the conclusion that source code containing proper words can be read faster than code in which many abbreviations or other non-words are present. The word length effect has been reproduced by several studies and therefore appears to be quite stable. Transferred to source code it could imply that short identifiers are easier to remember than long ones. This could be useful for a programmer who navigates within or between large files, as it could allow him to remember more relevant context, thus keeping a better overview.

1.3.3 Word frequency

As discussed above, low-frequency words have a higher naming latency than high-frequency words (Weekes, 1997). Source code is usually concerned with a technical domain but can include non-technical terms. A domain specific vocabulary is less frequent in natural language and might cause the same penalty during reading as other low-frequency words. Special terms from a technical domain like `Server`, `Request`, `Service`, or `Daemon` might therefore be slower to recognize than non-technical terms like `Customer`, `Person`, `Contract`. Here, the length of the word might also be of relevance.

A short, real world name might be easier to process than an elongated technical term, like `RequestCarrier`, `AccountExchangeProvider` or even

`AbstractSingletonProxyFactoryBean`[4]. The latter are concatenations of proper words but are quite uncommon in everyday language. They should be harder to recall and take longer to read than their individual parts.

1.4 Semantic Aspects

So far, we have discussed word length, and frequency, but ignored their *semantics*, i.e. what they mean. The term semantics has different meanings in programming language research and linguistics. In the context of programming languages it refers to the meaning of programmatic symbols (e.g. whether operators are valid for a certain data type or whether variables are visible within a certain scope). In the following, we use it to refer to a word's *meaning*.

Just like the surface aspects, semantics affect reading processes. For example, people perform better in lexical decision tasks when they have to identify words of the same semantic *category*. Collins and Loftus (1975) called this effect **SEMANTIC PRIMING**.

Semantics interact with *memory*. Miller (1994) showed that the capacity of working memory is limited to only a few items. He claimed that it can hold 7 (+/- 2) items, whereas other studies show that it is more likely limited to 3 to 5 items (Cowan, 2001). It is possible to overcome this limitation by a recoding process called **CHUNKING**, during which bits of information are grouped and organized (Carver, 1970; Miller, 1994). This process is likely relevant for code comprehension, as the following example illustrates.

The connections between entities from the implementation model and the intentional model are held in memory and their number should be limited by its capacity. However, it is possible to regroup elements in code, similar to the described chunking process. For example, imagine a function `drawCell(a, b, c, d, e, f)`, which accepts six floating point value parameters. Changing their semantics allows to regroup them. This can be accomplished by renaming the parameter names, which does not affect the function's signature: `drawCell(x, y, size, r, g, b)`. It still accepts six individual parameters, but they can be joined together. The first two can be grouped as a `Point`, the last three could be rewritten as a `Color` object, resulting in `drawCell(point, size, color)`.

[4]Which is not a joke, see (Hoeller, 2008).

This process can be understood as a chunking process, as only three elements are required, rather than six individual values. The extraction of objects into classes and functions has been recognized as a useful tool by other authors (e.g. Fowler, 1999), although psychological factors are not addressed. As illustrated, regrouping added additional meaning to the individual parameters and reduced their number, which shows that semantics, or the lack thereof, could help relieve memory related processes.

Semantics have a very practical implication on code comprehension. They allow the detection of defects, which is best illustrated with an example. The following expression is syntactically correct (e.g. in Python), but its meaning remains unclear.

```
yma = bnc * ccnd
```

Listing 1.3: A simple formula with meaningless identifier names.

Replacing the identifier names doesn't affect the structure of the formula or its result:

```
superman = batman * spiderman
```

Listing 1.4: The same formula, but with neutral semantics. All names are drawn from the same semantic category.

As we have established in section 1.3, the words should now be easier to identify, but their use is still unclear. The identifier names are proper, pronounceable words and are drawn from the same semantic category (superheroes), yet changing the formula would be impossible because the consequences of a change can't be understood. The operands seem to be related somehow but they still have a great distance to the applied operators. This example underlines the idea that *readability* is not just a perceptual aspect of an identifier, but that higher cognitive functions are involved. Although recognition plays a big role in code comprehension, it can't be discussed without semantic aspects.

Whereas Listing 1.3 uses non-words (even worse: arbitrary strings), Listing 1.4 uses proper words. In theory, the words should be quicker to read and easier to remember in the second than in the first example, although the first example consists of fewer characters, which illustrates word superiority, word frequency, and word length effects. To connect the surface properties of the identifiers with their meaning, let's take a look at the *lexical access*. Reading activates knowledge about concepts from the internal lexicon. In order to understand the code, its syntactical and formal structure must be correct, but additional information must be available that indicates its intended use: What is it supposed to do?

The semantics of the identifier names don't reflect the relationship between the code's entities (the variables) and the represented concepts from the real world. Rewriting the code as follows would ease the process of interpretation:

```
distance = speed * mileage
```

Listing 1.5: The same formula with more meaningful semantics.

It is now possible to change the formula appropriately. The structure still hasn't changed but the semantic context can be derived directly from the identifiers. This clarity offers a crucial benefit: semantic defects are much easier to detect. The shown formula raises the question why `distance` is dependent on `mileage`. Depending on the context of the variables, this formula could make sense. However, here it is indicated that it's incorrect. It should be:

```
distance = speed * time
```

Listing 1.6: Appropriate semantics allow for the detection of semantic defects.

As Listing 1.6 illustrates, semantics of identifier names allow for the detection of inconsistencies that can't be checked by the compiler. Such inconsistencies manifest themselves by producing incorrect behavior, such as crashes, data-loss, or bad values being displayed. A programmer fixing a defect must first identify parts of the program where the problem emerges. The initial state might be unknown, but the desired target state is available as some form of specification.

This illustrates that maintenance programming tasks can be viewed as *problem solving* activities. Complex problem solving research substantiates the relevance of semantics. For example, Beckmann (1994) found differences in knowledge acquisition between complex systems, when modifiable system variables were embedded into concrete or abstract semantic contexts. Beckmann and Guthke (1995) showed that semantics can deteriorate problem solving performance when the system works in an unexpected way. When the preexisting knowledge was compatible with their understanding of the system, participants performed better in keeping the system stable. These findings can be transferred to software maintenance, when variables in code are viewed as the manipulable variables of a complex system. Incorrect or inconsistent identifiers could impair a programmer's problem solving performance. We interpret these findings as evidence that semantics play a major role in code comprehension, and that identifier names can facilitate the detection of semantic defects.

1.5 Hypotheses

The shown examples substantiate the idea that code comprehension is affected by word length and semantics. Short identifier names should be quicker to read than long ones and allow to keep a better overview, because more items can be held in memory. On the other hand, their semantics can't be deduced from their names and additional resources are required to store what concept in the intentional model an identifier designates. Semantics appear to contradict the idea of optimizing for shortness. To see which effect is more dominant, we will now examine their influence during code comprehension tasks.

We will measure how fast programmers can understand code. Programmers who understand code faster should be more productive during software maintenance work. If length is more important than semantics, participants should understand code faster, when shorter identifier names are given; if semantics are more important than length, then longer identifier names with rich semantics should lead to a quicker understanding of source code than shorter identifier names with less semantic information. We'll test the following hypotheses in an experimental setting:

(I) Source code using words as identifier names is more quickly understood than source code using abbreviated identifier names.

(II) Source code using abbreviated identifier names is more quickly understood than source code using meaningless, single letter identifier names.

(III) Reading code without understanding during syntax-related tasks is unaffected by identifier quality.

To test Hypothesis I, we'll compare identifier names using words with abbreviated identifier names. Normal words carry semantic information. Their meaning can be accessed by retrieving it from the mental lexicon, and there should be no latency in accessing its relevance to the given code. This relieves the process of retrieving the associations between the implementation model and intentional model. Non-word identifier names are read slower and require access to the mentally mapped associations between the models, thus slowing down performance during code comprehension. Very short identifier names allow for a good overview and relieve memory, but their lack of semantics might impede comprehension.

In Hypothesis II we compare identifier names using abbreviations with identifier names using single letters which have no semantics. Abbreviations contain cues about a semantic context but carry less information than normal words. Single letters don't carry any meaning and should solely rely on associative retrieval processes and are therefore much slower to process than abbreviated identifiers.

Hypothesis III helps to counter-check our assumptions. We expect differences in comprehension, because identifier names affect building and maintaining mental mappings between the intentional model and implementation model. We assume that this is a distinct process which is relevant for code comprehension. Reading code without understanding it should therefore be unaffected by identifier names.

The hypotheses can be related to our cognitive model of comprehension in Section 1.1. During the code comprehension process, the implementation model is built by creating a code model first, and then expanding it with semantic information. It is then aligned with the intentional model, which allows to evaluate the consequences of code changes. When a code's intentional model is irrelevant, so is aligning it with an implementation model. Therefore, the semantics of an identifier should not affect the performance of working with code when a deep understanding is irrelevant. Syntax-related tasks should be unaffected by identifier quality.

In testing these hypotheses, we'll see how the different identifiers affect the understanding of code. In summary, short identifiers facilitate the creation of the code model, whereas identifier names, which resemble the associations between the models, facilitate mental mapping. We assume that identifier names like `c` lead to a slower understanding than identifier names like `cus`. Identifiers like `customer` should lead to the quickest understanding of the code, because of their semantics.

2 Method

We conducted a web-based experimental study to test our hypotheses. We captured how quickly people could detect a defect in a small piece of code. Identifier quality of the codes was treated as an independent variable in a within-subjects design.

We operationalized code comprehension by measuring how fast participants detected defects.

2.1 Participants

Professional German programmers were recruited to participate. All participants ($N = 72$) were male and between 20 to 51 years old ($M = 35.3$, $SD = 6.8$). They had between 4 to 35 years overall programming experience ($M = 14.0$, $SD = 5.8$) and between 2 and 15 years experience with C# ($M = 7.8$, $SD = 3.6$).

The experiment was announced at a technology industry conference in 2014 during which people had signed up to a mailing-list. Other contacts were invited via Twitter, inquiries in public C#-forums, and via a German social network for professionals called 'Xing.de'. A total of 221 people participated online[1]. From all people who had started the study, only 135 finished with a complete and usable record. 63 participants were removed because we considered them unfit for our sample based on their provided information.

We captured participant information and behavior using a handcrafted web client. Because the experiment was conducted online, we could not ensure an undisturbed work environment, and controlled for mentions of disturbances instead. The data of 17 participants were removed from the set due to distractions, which we captured with a direct question, and by evaluating a free text field for mentions of distractions. Trials without interaction for more than a minute ($n = 4$). Participants were asked whether they had worked on the tasks conscientiously, and their data was excluded upon denial ($n = 1$).

When a person had failed to find a defect in three attempts, the trial was marked as failed, and the complete record removed ($n = 15$). Slow trials that took longer than 10 minutes to fix a defect were also excluded ($n = 14$).

Participants who had partaken in a pretest (which was conducted two months earlier) were excluded, to control for possible learning effects ($n = 4$). Finally, participants were asked if they had participated in the same experiment before and expelled upon affirmation ($n = 8$).

Sufficient language skills were required in both natural languages and C#. Participants provided self-ratings of their language proficiency on a scale from 1-6, for German and English (CEFR, 2011). Participants with a rating below 4 were excluded ($n_{\text{German}} = 2$, $n_{\text{English}} = 9$). They were required to understand C# code, so that the specific features of the language wouldn't confuse them. 24 participants rated themselves as non-regular C# users (rating < 4 on a 1 - 5 scale) and were removed. In addition to their self rating, we asked for their years of experience in C#. Others, who had less than a year of experience with C#, were excluded ($n = 8$).

81% of participants had Abitur (German school degree after 12 to 13 years of schooling, similar to British A-Levels), 11% had Mittlere Reife (10 years of schooling, similar to British

[1]`https://code.psychologie.uni-heidelberg.de`

GCSE), 8% had a different degree. They reported training in technical fields such as mathematics, physics, and computer science, in which 42% had earned a Master's, 18% a Bachelor's degree. 18% had a vocational training as software developers. 15% had no secondary education degree. The remaining 7% had other qualifications. Most participants were employed (81%) or doing freelance work (17%), the rest were Students (2%). Describing themselves, 51% reported to be working as software developers, 12.5% called themselves consultants, 12.5% identified as software engineers. Other self-descriptions included leading or project management positions (10%), and software architects (8%). The remaining 6% chose broader descriptions, such as 'computer scientist'; 2 students were included.

2.2 Design

Participants were shown snippets of code, in which we had built either a single semantic defects or syntax errors. We measured how long they examined the code until they had found the defect. Using an implementation of the 'Restricted Focus Viewer' paradigm (Jansen, Blackwell, & Marriott, 2003), we captured coarse-grained data of visual attention. This allowed us to measure how long participants looked at specific areas, which we captured for exploratory purposes.

Each participant saw several different **snippets** of code. In each snippet, the identifiers were named using either normal words, abbreviations, or single letters. The particular **identifier Type** was altered as a within-subjects factor, i.e. each participant saw all different realizations. The snippets were kept constant in every other regard so that renaming altered the identifier's designation but not their identification. Examples for the different types are given in Table 2.1.

Table 2.1: Examples of the different identifier types used.

Type	Example
Word	`request`, `histogram`
Abbreviation	`rqs`, `hst`
Letter	`a`, `b`

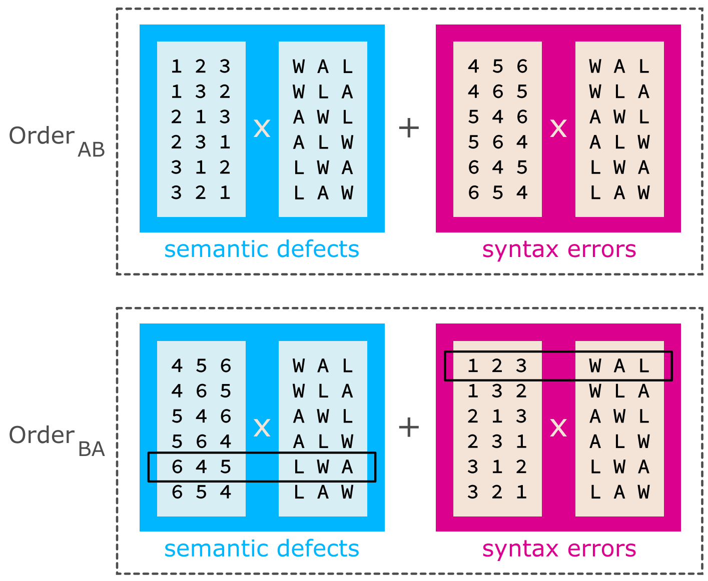


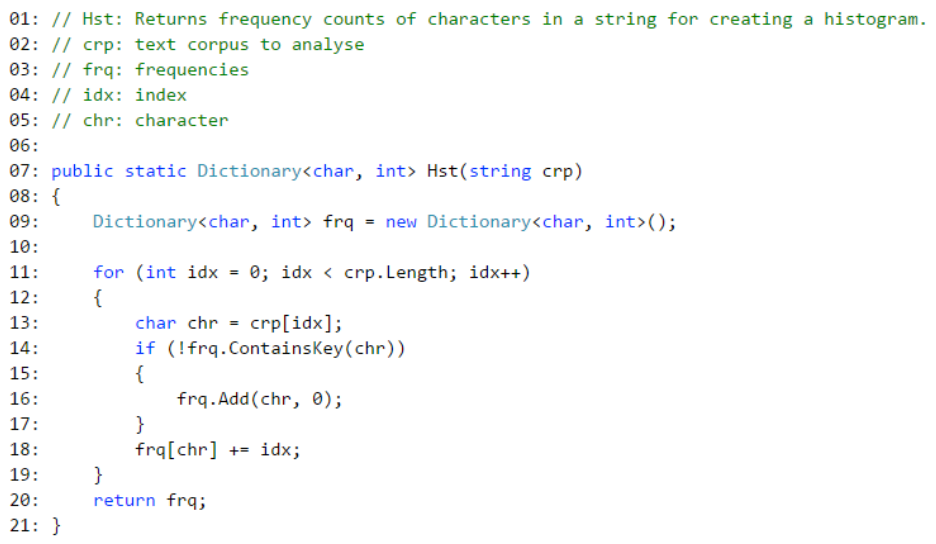
Figure 2.1: The balanced design. Snippet Group A: 1,2,3. Snippet Group B: 4,5,6. The groups were permuted and multiplied with the identifier type: Word (W), Abbreviation (A), Letter (L). Each order contained 36 sets. Each set consisted of 6 trials, for example 6L-4W-5A-1W-2A-3L.

'Word' identifiers were named using normal words. 'Abbreviations' were exactly three characters long. 'Letter' identifiers consisted of a single character.

Before the actual study, we performed a pretest to identify whether there were differences between the snippets. We identified two clusters based on their difficulty and split the complete set into two groups (A and B), containing three snippets each.

During the experiment, participants performed two tasks. First, they were asked to find SEMANTIC DEFECTS and then SYNTAX ERRORS. The snippets during each task were drawn from the same group, e.g. a participant saw only snippets from group A during the semantic task and snippets from group B during the syntactic task. The snippets in each task were permuted, and multiplied with the permuted set of identifier types (Word, Abbreviation, Letter). The order of the tasks was kept constant, but the order of snippets and identifier types was permuted, so that no ordering effects would occur. This resulted in a balanced plan which required a total of 72 participants performing 6 trials each. Participants were randomly assigned to the individual trial sets. The layout is displayed in Figure 2.1.

Our design can best be described as a within subject design, with three different factor levels (identifier type), which was repeated twice, with a different task and different snippets. It was built to make efficient use of the snippets, control for sequence effects, and make economic use of participants.



```
// Hst: Returns frequency counts of characters in a string for creating a histogram.
// crp: text corpus to analyse
// frq: frequencies
// idx: index
// chr: character

public static Dictionary<char, int> Hst(string crp)
{
    Dictionary<char, int> frq = new Dictionary<char, int>();

    for (int idx = 0; idx < crp.Length; idx++)
    {
        char chr = crp[idx];
        if (!frq.ContainsKey(chr))
        {
            frq.Add(chr, 0);
        }
        frq[chr] += idx;
    }
    return frq;
}
```

Figure 2.2: A code snippet with abbreviated identifiers. The head comment contains the function's specification.

2.3 Materials

A total of 11 snippets was developed, six of which were used for the study. They were written in C#. We provided unit-tests to make sure that they produced a valid output. The snippets are available in Appendix B.

Each snippet consists of a self-contained, 15 line long C# function, with a comment on top. The first line of the comment explained the function's use and conveyed what it is supposed to do, so that participants could create an intentional model.

Each snippet was adapted to the experimental factor. First, we created a version with normal word identifier names. From these words, three-letter abbreviations were deduced by removing all vowels and using the first three remaining consonants (e.g. `query` would become `qry`). Single letter identifier names were created by alphabetically renaming identifiers in order of their occurrence (`a` for the first, `b` for the second, etc.). The used identifiers were commented in the comment at the top, by showing their original name and its replacement. An identifier named `auxiliary` that had been renamed to `f` resulted in a comment as shown in Listing 2.1.

```
// f: auxiliary
```

Listing 2.1: Example of an explanatory comment.

Every snippet was altered to contain a single semantic defect, which we define as a non-terminal logic error. They were built in such a way that the code would compile, and produce output which would not fit the desired specification.

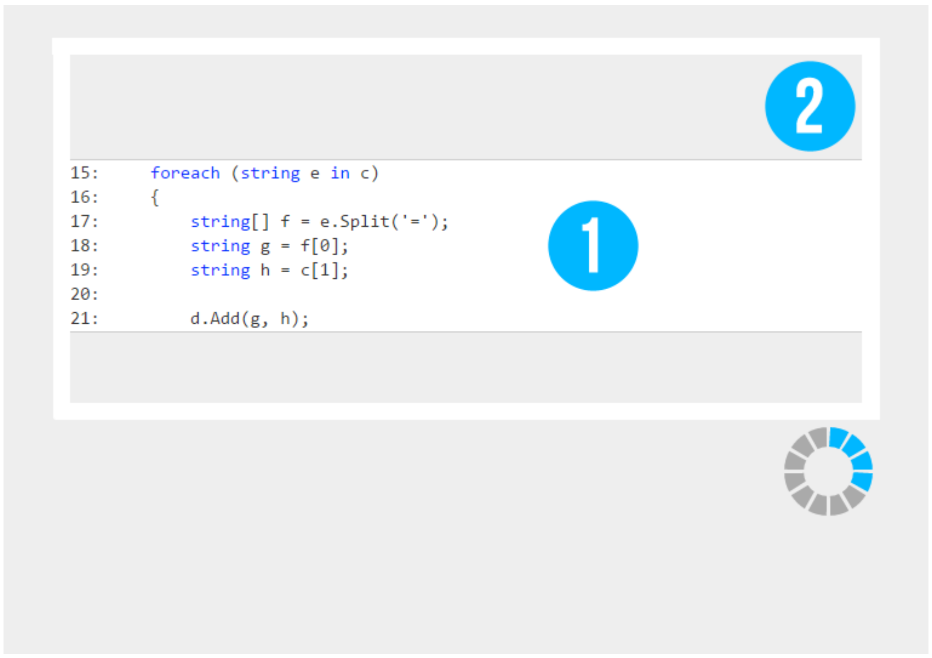
```
...
15 string line = rawLine.Trim();
16 string[] setting = line.Split('=');
17
18 string identifier = setting[0];
19 string property = setting[1];
20
21 settings.Add(identifier, line);
...
```

Listing 2.2: Semantic defect in the excerpt of a code snippet. The defect resides in line 21. In this context, `settings` is a dictionary which is supposed to capture the function's output. The parsed `property` should be added to the dictionary, rather than the line object.

Listing 2.2 shows an example of a possible programming mistake. The value `property` is extracted but never used, which results in incorrect data in the settings dictionary. Semantic defects are particularly interesting because they are common in programming and can be hard to detect. For the second task, a syntax error (e.g. a missing semicolon) was built into the snippets, each in a similar location as the semantic defect.

To distribute the snippets over the web, they were converted into HTML. A monospaced font was used and whitespace characters preserved to ensure consistent formatting. Line numbers and syntax highlighting were added, mimicking a bright Visual Studio style with blue keywords and green comments. We assumed this style would be natural for C# developers.

During the task, participants would face some peculiarities. As shown in Figure 2.3, the code (1) was not fully visible but partially overlaid with a cover (2), so that only 7 lines ($^1/_3$ of total line count) were visible at once. We called this cover the letterbox, as it mimics looking through a letter deposit slit in a door. Participants could shift the letterbox up and down using the cursor keys in order to gradually reveal parts of the code until they had seen it completely. As an implementation of the restricted focus viewer paradigm (Jansen, Blackwell, & Marriott, 2003), it is a simple alternative to eye tracking that can also be distributed over the web and does not require expensive apparatus or calibration.


```
foreach (string e in c)
{
    string[] f = e.Split('=');
    string g = f[0];
    string h = c[1];

    d.Add(g, h);
```

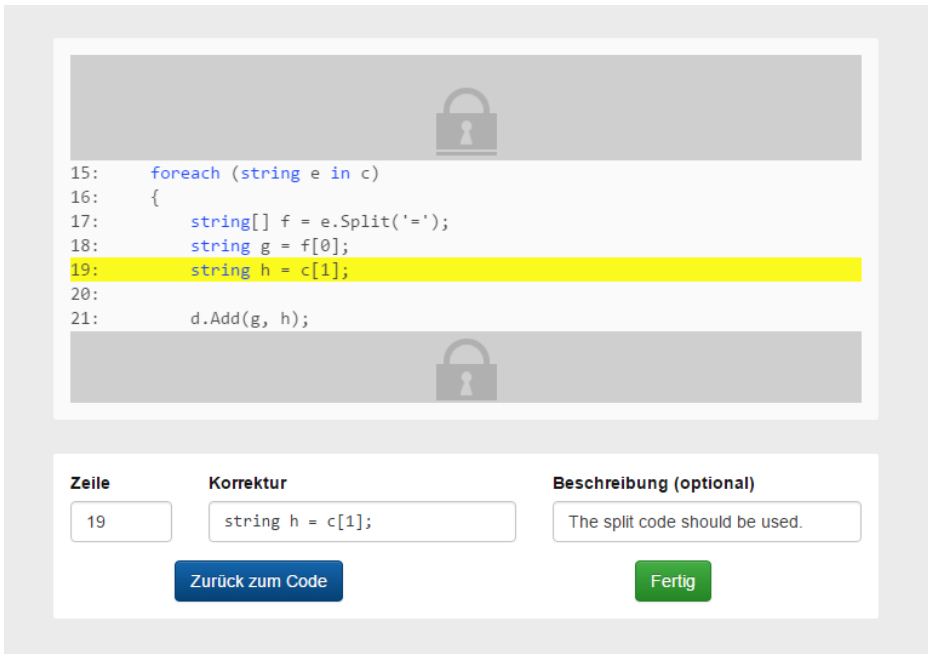
Figure 2.3: The letterbox. During the task only seven lines were visible at once (1), the remainder was overlaid with a movable cover (2). The clock in the bottom right corner displayed a warning after 8 minutes and skipped to the next trial after 10 minutes.

2.4 Procedure

Participants were invited to a public website that introduced the study and explained conditions and requirements (C# knowledge, 20 minutes duration, code snippets, 'Please use a desktop or laptop computer'). Legal information and a privacy statement were available. Clicking a big green button led to a first questionnaire where participants provided information about their current employment status. During procedure a progress bar indicated how far the participant had progressed. The second page showed a form asking about their professional experience.

In the following step participants saw a tutorial introducing them to the views and controls of the upcoming trials. The tutorial gradually introduced participants to the upcoming task. It outlined a scenario, explained the required controls and gave instructions on the task.

The start of each trial was activated by pressing the space bar. During the trial, participants were required to slide the letterbox up and down using the cursor keys. These constraints were implemented in order to normalize input behavior, by discouraging people from using the mouse and keeping their hands on the keyboard. This prevented participation from mobile devices, which are usually not used for software development.



```
15:     foreach (string e in c)
16:     {
17:         string[] f = e.Split('=');
18:         string g = f[0];
19:         string h = c[1];
20:
21:         d.Add(g, h);
```

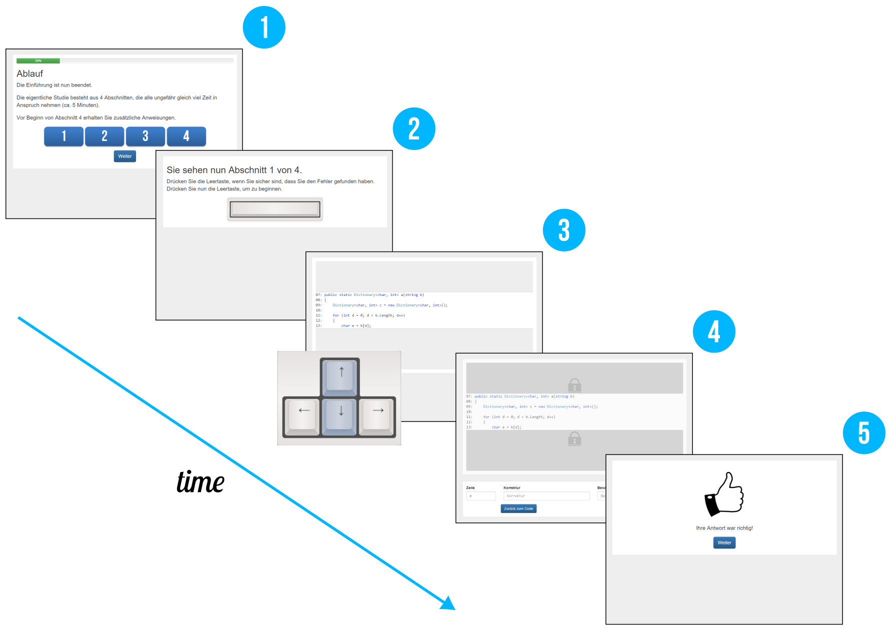
Zeile	Korrektur	Beschreibung (optional)
19	string h = c[1];	The split code should be used.

Figure 2.4: The correction dialog. Participants were asked to give detailed explanations about the found defect. They were asked for the line number, a correction of the defect and an optional, free-text description.

When participants had found the defect in the code, they pressed space, thus activating the correction dialog shown in Figure 2.4. This also locked the letterbox. Participants were asked to specify the defective line, a correction, and a free, optional description of the defect. The provided line was highlighted in order to allow participants to verify their answer.

After the tutorial, participants entered the experiment phase. Figure 2.5 outlines the procedure. Participants first saw three snippets containing semantic defects and then continued with three syntactic defects. They saw an instructional screen before the changeover, preparing them for the new task. We assumed that finding all three syntactic defects would take as much time as a single semantic trial. In order not to discourage participation with the upcoming amount of work, we mentioned at the beginning that four trials would follow (see Figure 2.5, step 1).

Each trial was started by pressing the space bar, so that participants would move their hands to the keyboard (2). They moved the letterbox up and down using the cursor keys, until they had spotted the defect (3) and pressed space to open the correction dialog (4). When they were finished, they briefly received feedback about their results (5) and were guided to the next trial (restarting at step 2 with a different snippet). If their answer was incorrect they were allowed two more attempts to give a correct answer. A warning was displayed after 8 minutes. After 10 minutes the software marked the trial as failed and skipped to the next trial.



Ablauf

Die Einführung ist nun beendet.

Die eigentliche Studie besteht aus 4 Abschnitten, die alle ungefähr gleich viel Zeit in Anspruch nehmen (ca. 5 Minuten).

Vor Beginn von Abschnitt 4 erhalten Sie zusätzliche Anweisungen.

Sie sehen nun Abschnitt 1 von 4.

Drücken Sie die Leertaste, wenn Sie sicher sind, dass Sie den Fehler gefunden haben.
Drücken Sie nun die Leertaste, um zu beginnen.

```
public static Dictionary<char, int> a(string b)
{
    Dictionary<char, int> c = new Dictionary<char, int>();

    for (int d = 0; d < b.Length; d++)
    {
        char e = b[d];
```

Ihre Antwort war richtig!

Figure 2.5: The sequence for a single trial. The experiment was introduced (1). Before each trial, a splash screen prepared the participant for the upcoming task (2). Participants then searched for a defect (3) and filled out a form on the correction dialog (4). They received short feedback (5) when they were finished. During the experiment each participant performed six trials (step 2 to 5).

After they had seen the last snippet, participants filled out another final questionnaire about their demographic data (e.g. age, gender, language proficiency). They provided additional control data, such as whether or not this was their first attempt or if they had encountered distractions.

The experiment ended with a 'Thank You' page, with the possibility to subscribe to a mailing list in order to receive study results or invitations for future studies.

3 Results

During the experiment, we collected data of participant interactions. Each participant's record contained data from the questionnaires and the interaction events from the defect finding trials. We calculated how much time participants had spent finding each defect and subtracted how much time they spent filling out the correction dialog. For our calculations we used only this value, which indicates how long participants were exposed to code. The descriptive data are shown in Table 3.1, separated by identifier type.

The distribution of response times was skewed to the right, which is common for reaction time experiments. Fast responses had accumulated on the left and the distribution exhibited a long tail of slower response times, which is likely to be caused by the presence of slow outliers. According to Ratcliff (1993), outliers are caused by a process other than the process under study (e.g. guesses or distractions), and overlap with the distribution from the real process. He

Table 3.1: Durations of how long participants interacted with code, split by identifier type. The values show the median and interquartile range (IQR) in `mm:ss.ms` (Minute, Seconds, Milliseconds).

	Semantic		Syntactic	
Type	Median	IQR	Median	IQR
Word	01:24.48	01:12.78	00:39.42	0:49.00
Abbreviation	01:38.57	01:05.37	00:36.71	0:53.92
Letter	01:40.36	01:24.87	00:35.74	0:30.22

suggests various methods to reduce their effect, e.g. transforming the data or cutting off at a certain value. We decided against a cutoff criterion, because we had already excluded distracted participants and slow trials, and chose an inverse transformation ($^1/_{\text{RT}}$) instead. Removing a single value would have dropped two others, due to the balanced repeated measures design, which facilitated this decision. A logarithmic transformation was discarded because the resulting data would be harder to interpret. Also, logarithmic transformations can result in a reduced power under some circumstances, e.g. when the effect occurs in the distribution's right tail (Ratcliff, 1993, p. 514). The inverse transformation is a good compromise to maintain the power of our statistical tools (ANOVA) and keeping the transformed data interpretable. Instead of 'minutes to fix a defect', our calculations use values that we interpret as **Defects per Minute**. A higher value indicates that a participant was faster than one with a lower value.

To isolate the effect of our experimental factor, we had tried to reduce the variance caused by our stimulus materials. The snippets were designed to be similar in structure so that the variances between them would be very small, but dissimilar enough to prevent learning effects. Their length was held constant to 15 lines and each contained between 91 and 94 tokens ($M = 92$). Figure 3.1 shows box plots of how long participants worked on the code during the semantic tasks, split by snippets. Participants spent much more time on snippet number 3, which was added after the pretest. We did not consider the snippets in our calculations as an experimental factor, which is unproblematic because we had used a balanced design. The snippets are listed in Appendix B.

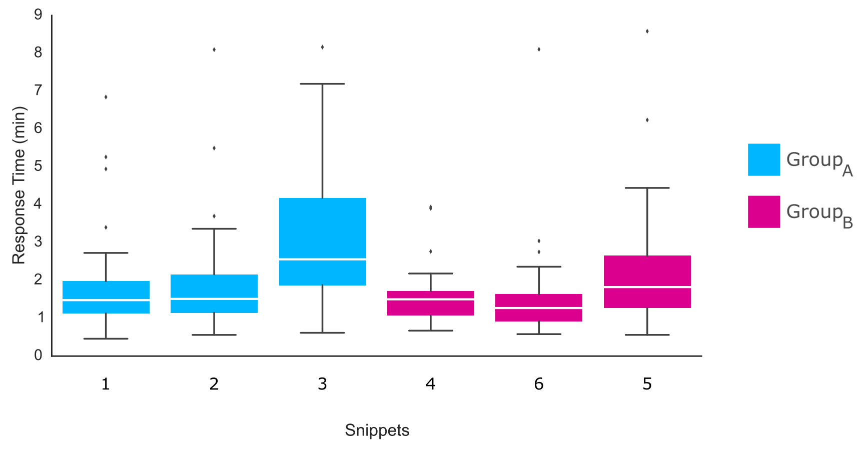


Figure 3.1: Box plots of the time during which participants saw code in minutes, separated by snippets.

Hypotheses I and II were tested using linear contrasts to gain detailed insight into the relations between our the identifier types. For Hypothesis III, we used a repeated measures analysis of variance (ANOVA). All calculations assume a significance level of $\alpha = .05$. Contrast Ψa compared normal word identifiers to non-words by grouping together abbreviations and single letter identifiers (Hypothesis I). Contrast Ψb compared single letters with abbreviated identifiers, omitting normal word identifier names (Hypothesis II). The relation between the different identifier types can be seen in Figure 3.2 for the semantic task and Figure 3.3 for the syntactic task.

The descriptive data for our tests can be seen in Table 3.2. We found a statistically significant difference between normal words and non-words (abbreviations and single letters together) ($t_{\Psi a}(71) = 2.73$; $p = .004$). There was no significant difference between single letters and abbreviations ($t_{\Psi b}(71) = 0.07$; n.s.). The difference between words and non-words showed a medium sized effect ($d_z = 0.32$, Cohen, 1988). A power analysis using G*Power (Faul, Erdfelder, Lang, & Buchner, 2007) yielded appropriate power ($1 - \beta = .85$).

Hypothesis III states that reading code without understanding is unaffected by identifier quality. We anticipated that syntactic errors would be quicker to find than semantic defects. A Student-T test for dependent samples indicated that participants finished faster in the task

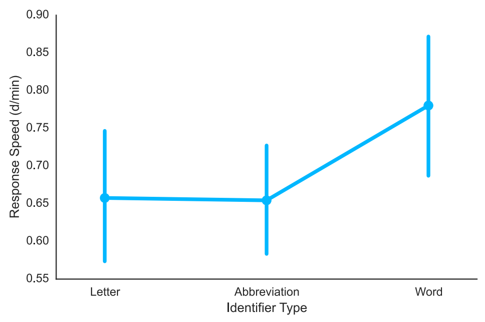


Figure 3.2: Response speed during semantic task by identifier type. Values show 'defects per minute'. Participants detected defects faster when code contained identifier names using words instead of abbreviations and single letters. The vertical bars show 95% confidence intervals.

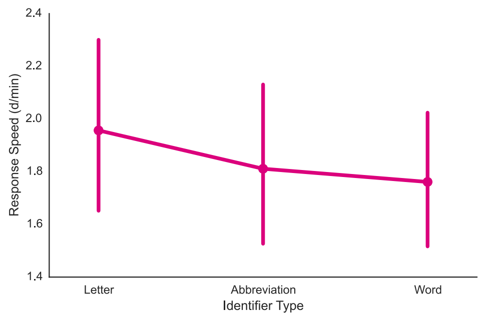


Figure 3.3: Response speed during syntactic task by identifier type. Syntax errors were found much faster than semantic defects. Although there appears to be an effect of identifier type, it is not statistically significant.

Table 3.2: Response speeds during the semantic task, split by identifier type.

		Semantic		Syntactic	
Measure	Type	*M*	*SD*	*M*	*SD*
Defects / Minute	Word	0.78	0.42	1.76	1.13
	Abbreviation	0.65	0.31	1.81	1.31
	Letter	0.66	0.39	1.96	1.39
Comment Reading Speed	Word	3.27	2.96	17.85	12.66
	Abbreviation	2.64	2.59	14.34	10.75
	Letter	2.33	1.88	14.77	10.37
First Pass Reading Speed	Word	0.03	0.05	0.05	0.04
	Abbreviation	0.02	0.01	0.05	0.05
	Letter	0.03	0.02	0.04	0.02

to find syntactic errors ($M_{\text{Syntactic}} = 1.84$; $SD_{\text{Syntactic}} = 1.01$) in comparison to find semantic defects ($M_{\text{Semantic}} = 0.70$; $SD_{\text{Semantic}} = 0.29$; $t(71) = -10.43$; $p < .001$; $d_z = 1.23$), so apparently, finding syntax errors is faster than finding semantic defects. No significant effect could be found in a repeated measures ANOVA for different identifier types during the syntactic task ($F(2, 142) = 0.8$; n.s.).

During the study, participants moved the letterbox up and down, which we analyzed with a repeated measures ANOVA. The duration how long participants read comments was affected by the identifier quality ($F_{\text{Semantic}}(2, 142) = 5.35$; $p = .006$; $\eta^2_{\text{Semantic}\langle\text{Type}\rangle} = 0.07$; $F_{\text{Syntactic}}(2, 142) = 3.60$; $p = .03$; $\eta^2_{\text{Syntacic}\langle\text{Type}\rangle} = 0.05$). Participants spent significantly different amounts of time studying the comments, depending on what type of identifier they saw. There was no significant effect of identifier quality on first pass reading speed ($F_{\text{Semantic}}(2, 142) = 2.55$; n.s.; $F_{\text{Syntactic}}(2, 142) = 1.46$; n.s.).

4 Discussion

Using a within-subjects design, we measured the time participants needed to detect semantic defects and syntax errors in code. The lengths and semantics of identifier names were changed to single letters, abbreviations, and normal words. We analyzed our data using linear contrasts and repeated measures analyses of variance (ANOVA). Semantic defects were found significantly faster when the presented code contained identifier names using normal words, than when non-words were presented. When code used abbreviations or single letters as identifier names, participants were equally fast, but slower than during trials with normal word identifier names. Finding syntax errors takes significantly less time than finding semantic defects, and also appears to be unaffected by identifier names.

Hypothesis I states that participants understand code faster when normal words are used as identifier names, as opposed to abbreviations. Our data supports this hypothesis. Participants found defects faster in code using normal word identifiers, as opposed to abbreviations, which we attribute to the additional semantic information of the identifier names. Although they were longer, normal words lead to quicker defect detection than abbreviations. Apparently, word length was less important during the examined tasks, and semantics seemed to play a bigger role. Identifier names which are less to read do not necessarily lead to a quicker comprehension.

We could not find support for Hypothesis II. We assumed that code using abbreviations would be faster understood than code using single letter identifier names, but our data showed no significant difference between these two conditions. Defects in code with single letter identifier names were found as fast as defects in code with abbreviations. Our hypothesis was based on the idea that abbreviations contain traces of semantics, as they were derived from normal words. Apparently, the procedure we used to abbreviate normal word identifier names did not retain enough semantic cues to make a measurable difference. Another possible outcome would have been an improved defect detection in code with single letter identifier names. Single letter names are shorter, and we anticipated that they should be quicker to read and easier to memorize. Since no difference could be found between abbreviations and single letters, we conclude that identifier name length does not affect the performance of code comprehension tasks as much as semantics.

Hypothesis III was substantiated by our data. We assumed that reading code without understanding is unaffected by identifier quality. We presented participants with a second task in which they had to find syntax errors. They found them faster than the semantic defects in the first task. Their performance was unaffected by the different identifier names. Normal words, abbreviations, or single letters did not affect how fast participants found syntax errors. It appears that the detection of syntax errors doesn't require a deep understanding of the code's meaning, otherwise there should have been an improved speed during trials with normal word identifier names.

In summary, our findings underline the relevance of semantics during code comprehension. The data obtained from the letterbox substantiates this view. When participants worked on code using single letters and abbreviations, they spent more time reading the specification comment section at the top.

In section 1.1, we suggested that during code comprehension, the elements of a mental Intentional Model and Implementation Model are associated with each other, by a process of mental mapping. Identifier names that mirror these associations make it easier to create, mentally manipulate and evaluate them during code comprehension tasks. Our findings seem to support this conception.

In our experiment, identifiers with a word name lead to a quicker defect detection, because they were readable, easy to memorize and recall and thus facilitated understanding the consequences of changes to the code. Code comprehension was hindered by abbreviations and single letters, because they did not mirror the associations between an entity (i.e. an element from the Implementation Model) and a designated concept (i.e. an element from the Intentional Model). We saw that identifier length and semantics had no effect on the time it took to find syntax errors. To identify a missing semicolon or a mismatched bracket, it could be sufficient to match a visual pattern without understanding the code. The mental mapping process seems to be relevant for understanding code, but not for this type of pattern matching, which may work on a much lower perceptual level.

In the frame of our model, the process of finding syntax errors can be described as building an internal Code Model, because the code is looked at, but semantic information from identifiers and given specifications was ignored. The model's components seem appropriate to explain various aspects of code comprehension, as the mental mapping explains the improved speed when normal word identifier names were presented.

Our results could be explained with other theories, for example top-down and bottom-up comprehension. Bottom-up approaches argue that programmers look at the low-level details and create a semantic structure, which is then abstracted by chunking processes until the program is fully understood (Shneiderman & Mayer, 1979). Brooks (1983) outlines a top-down approach to code comprehension. In this perspective, programmers posses knowledge of other domains from which they draw hypotheses. They try to find evidence for these hypotheses in the form of *beacons*, which are patterns that activate *knowledge schemas*. For example, a code containing swaps inside a pair of loops activates a sorting schema (Brooks, 1983).

While these approaches focus on the strategies applied by programmers during code comprehension tasks, our model focuses on the mental representation of entities from Application Space and Implementation Space, and how they are brought together. The strategies can be understood as different 'routes' in the scope of our model. Bottom-up processes are similar to building an Implementation Model from a Code Model, whereas a top-down approach could be explained by building an Implementation Model from an Intentional Model. We believe that in practice these two strategies interact with each other. von Mayrhauser and Vans (1993) presented an integrated meta-model of code comprehension, in which they described the formation of understanding by switching between bottom-up and top-down processes.

4.1 Threats to Validity

To discuss the validity of our findings, we should first address the relatively high dropout rate. We had used a balanced repeated measures design, which required less participants, compared with a between-subjects design. Its disadvantage was that discarding a participant's record removed the data of six trails from the complete set. Overall 221 people filled out at least one questionnaire form but only 135 finished all 3 questionnaires and 6 trials. 63 records were excluded because the participants did not match our criteria. Our design required a total of 72 participants but considerably more people participated in our study. The high dropout rate is a result of strict filtering and can be seen as a measure of quality control.

We observed a sample of German, adult, male, C# programmers. Although participants were required to have appropriate English skills, the language barrier might still affect their performance. Two participants identified as female but were not included in the final data set, based on the aforementioned criteria. Because we had filtered our sample for programming experience and not gender, we have no reason to believe that women would perform differently in the presented tasks.

Pennington (1987) substantiated the idea that comprehension depends on the used programming language. She found that COBOL programmers succeeded better at answering questions about a program's data flow than FORTRAN programmers, who in turn had built a better understanding of control flow. Applied to our findings, we argue that the effects we found might also occur in a sample of Java programmers, but C programmers or people using functional languages (e.g. F#, Haskell) might work with code in a fundamentally different way and be less susceptible to the lack of semantics in identifier names. When the problem domain includes mathematical formulas or memory management is important, the semantics of the problem domain might be less relevant. Abstract code with single characters or abbreviations might be appropriate to describe such problems and the lack of semantics might not affect comprehension.

Normalizing the snippets was supposed to help isolate the effect of identifier type by reducing the variances caused by the materials, but this also limits how much the findings can be generalized. Large code bases often contain functions with thousands of lines, deep nesting, high cyclomatic complexity (McCabe, 1976), mixed identifier types and aspects such as recursion or side effects. Since we could measure the effect in just 15 lines of code, we are certain that identifier names affect larger code bases as well, possibly in interaction with other properties of code. For example, the effect might be amplified, when other concepts, such as recursion or specific design patterns, are present.

Finally, we should discuss the defects we had built into the snippets. Each snippet was syntactically correct but produced valid, but unusable output. This was achieved by 'confusing' a variable for another, while meeting the programming language's syntactical and semantical requirements. In some cases this lead to unused variables which modern tools can detect automatically, so that these particular defects might not be relevant in practice, but in sum the defects were complex enough to require a certain period of reasoning about the code.

4.2 Further Research

Our experiment could be repeated with materials written in other languages, for example F#, Python, C or JavaScript. This could show whether identifier names affect code comprehension in general or whether the effect is specific to our sample of C# programmers. If the found effect is caused by the properties of mental processes and not by our experimental settings, the results for different languages should be similar. Future studies should consider interactions between identifier names and other aspects of code such as architectural patterns, or data flow, to show how identifier quality affects performance during comprehension in a larger scope.

Variances among the code snippets bear opportunities for future research. They could pose as items with varying difficulty, and be used as a diagnostic instrument to differentiate individual programming aptitude. We had used a repeated measures design which controls for inter-individual differences, but these differences might very well be the focus of future experiments.

We would like to investigate how clearly our model's components are linked with other cognitive mechanisms. For example, controlling for a participant's working memory capacity might help understand its role in the mental mapping process in depth. In order to understand how similar code reading is to text reading, eye tracking or electroencephalography (EEG) measures could be analyzed instead of the restricted focus viewer.

Our experiment covered length and semantics of very short identifier names, but in practice they tend to become very long, and use concatenations of words. We found that the longer, more semantically rich words facilitated defect detection, but we would suspect that very long identifier names diminish this effect, because it becomes harder to identify which concept is the most relevant. Therefore the boundaries of identifier names should be analyzed.

4.3 Practical Implications

Normal words facilitate the understanding of code and make it easier to detect semantic defects. Because understanding code and finding defects is a common task during software maintenance, code with normal words could help reduce the costs of development. Developers will spend less time finding and fixing defects. We would like to encourage programmers to use normal word identifiers where ever possible, so that others can understand their code faster. This suggestions appears feasible even for programmers who work on their own, and look at their own code after a long time, e.g. scientists or self-employed software developers. Normal word identifiers then might help quickly recreate the context the code was originally written in.

However, there are exceptions to this suggestion. Abbreviations might be part of the problem domain's specific vocabulary. Their meaning can then be accessed from the mental lexicon. Such common abbreviations might not impede comprehension as much as the meaningless abbreviations we presented. In our study, we chose single letter identifier names alphabetically to make sure they contained no semantic information. In practice, even single letter identifiers may express semantics depending on their context. For example, it is common to name the current element of an iteration `i`, or when the identifier names `x` and `y` are used successively, a programmer interprets them as coordinates. These semantics are established by convention. Experts who have spent a lot of time building complex algorithms might be accustomed to generalized solutions and abstract or short identifier names that have no concrete semantics. They might be confused when such conventions are broken and normal words might impair their performance during software maintenance, when they expect variables to be named with single letters. Novices, on the other hand, may benefit from normal word identifier names. When new programmers are introduced to a team, they have to be introduced about the source code and get an overview over its functioning. Normal word identifier names therefore could help a programmer to become productive faster. Students might benefit from word identifier names illustrating concrete examples, instead of meaningless abstract names like `foo` and `bar`.

4.4 Conclusion

We conclude that normal words lead to a faster understanding of code and facilitate finding errors. They save time because they help programmers to understand code faster. In the bigger picture, normal word identifiers help to reduce errors because they make them easier to discover. Naming identifiers using proper words reduces costs during a program's development and life-cycle and could be considered a sustainable investment.

A
Glossary

Italics refer to other items in the Glossary.

C#: (pronounced 'see sharp'), a multi-paradigmatic, multi-purpose programming language by Microsoft.

Category: Set of items that are treated equivalently.

Class: In Object-Oriented Programming (OOP), a class is a combination of data and behavior.

Code: See *source code*.

Comment: Text elements of source code that are ignored by the compiler. They mostly provide documentation or licensing information, although sometimes they are utilized by preprocessing facilities (Lemburg & von Löwis, 2001).

Compilation: The process of translating human-readable *source code* into machine language, usually in a binary or intermediate format.

Compiler: A program that compiles. See *compilation*.

Concept: Mental *representations* of *categories*. The form units of knowledge and reasoning Rajlich and Wilde (2002). The concepts represented in a program are manifold and depend on the program's purpose. A program for booking seat reservations with an airline might include concepts such as "seat" or "reservation". A program that organizes a user's virtual desktop might include "shortcuts", "files" and "destinations".

Designation: The association of a word with a *concept*. Other descriptions of this relationship include, for example, "reference, denotation, naming, standing for, aboutness, or even symbolization or meaning" (Newell, 1980, pp. 156).

Entity: An entity is a "thing that is". In the context of a program this refers to relevant elements (e.g. *classes*, *variables*, *functions*).

Function: A named grouping of *statements*.

Homonym: A word with multiple meanings, e.g. 'bank', which could mean a financial institution or the bottom of a river.

Identification: Referring to an *entity* within a program, in order to evaluate it in its semantic context (e.g. to call a *function*, store data in a *variable*, etc.).

Identifier name: A concrete string *literal* and can be chosen according to a grammar's lexical rules. The chosen word of the *identifier* then *designates* a *concept*.

Identifier: A lexical *token* of a programming language that names and refers to a program's *entities*. An identifier has an *identifier name*. Identifiers have two purposes: *identification* and *designtation*.

Interpreter: Much like a *compiler*, an interpreter is used to read *source code*, but rather than outputting a different format, it immediately runs it as a *program*.

Keyword: A keyword is a reserved word in a programming language that has a specific meaning. For example `for` indicates a specific kind of loop (e.g. in C, Java or Python).

Program: A sequence of instructions that are executed by a machine.

Programming: The task of manipulation *source code* in order to create software.

Representation: A structure that *designates* by carrying information about a concept that can be consumed by a process (Newell, 1980, pp. 176). *Identifiers* are representations because they are structures that designate *concepts* and are processed by cognitive or machine processes.

Snippet: A delimited fragment of *source code*.

Source Code: A text that is written in a programming language into a file which defines the inner workings of a program. To prepare and execute the program, the source code is processed by a *compiler* or an *interpreter*.

Statement: The smallest independent unit of instruction in *source code*.

Synonym: Multiple words with the same meaning, e.g. buy and purchase.

Token: A typed combination of input characters. It is the basic lexical unit of *source code*.

Variable: A storage point for a data value.

B

Snippets

The shown snippets were used in our study. Other snippets are available online at `https://github.com/cessor/bachelor-thesis`. Some comments had to be split by the end of the line to fit to the page.

In section 3 the snippets were referred to by their numbers. Their names map to numbers as follows:

Group A

1. ParseQueryString
2. Histogram
3. ConcatLists

Group B

4. CodeStructure
5. ReadIni
6. CountChildren

```
// ParseQueryString: Parses a http query-string
// querystring: raw query-string containing key=value pairs,
    separated by &
// parts: parts
// query: query
// part: part
// setting: setting
// parameter: parameter
// parameterValue: parameterValue

public static Dictionary<string, string> ParseQueryString(string querystring)
{
    string[] parts = querystring.Split('&');
    var query = new Dictionary<string, string>();

    foreach (string part in parts)
    {
        string[] setting = part.Split('=');
        string parameter = setting[0];
        string parameterValue = parts[1];

        query.Add(parameter, parameterValue);
    }
    return query;
}
```

```
// Histogram: Returns frequency counts of characters in a string for
    creating a histogram.
// corpus: text corpus to analyse
// frequencies: frequencies
// index: index
// character: character

public static Dictionary<char, int> Histogram(string corpus)
{
    Dictionary<char, int> frequencies = new Dictionary<char, int>();

    for (int index = 0; index < corpus.Length; index++)
    {
        char character = corpus[index];
        if (!frequencies.ContainsKey(character))
        {
            frequencies.Add(character, 0);
        }
        frequencies[character] += index;
    }
    return frequencies;
}
```

```
// ConcatLists: Concatenates two lists of the same length
// start: collection of elements at the start
// end: collection of elements to append
// length: length
// result: result
// index: index
// first: first
// second: second

public static int[] ConcatLists(int[] start, int[] end)
{
    int length = start.Length;
    var result = new int[length * 2];

    for (int index = 0; index < length; index++)
    {
        int first = start[index];
        int second = end[index];

        result[index] = first;
        result[index + 1] = second;
    }
    return result;
}
```

```
/ CodeStructure: Generates a broad overview over the block structure
    of C-family source code
// sourceCode: codefile to be surveyed
// blockCharacters: blockCharacters
// structure: structure
// index: index
// character: character

public static IEnumerable<char> CodeStructure(string sourceCode)
{
    List<char> blockCharacters = new List<char> { '{', '}' };
    List<char> structure = new List<char>();

    for (int index = 0; index < sourceCode.Length; index++)
    {
        char character = sourceCode[index];
        if (blockCharacters.Contains(character))
        {
            blockCharacters.Add(character);
        }
    }
    return structure;
}
```

```
// ReadIni: Parses the lines of an ini file
// lines: collection of lines containing one setting each, like key=value
// settings: settings
// rawLine: rawLine
// line: line
// setting: setting
// identifier: identifier
// property: property

public static Dictionary<string, string> ReadIni(IEnumerable<string> lines)
{
    var settings = new Dictionary<string, string>();
    foreach (string rawLine in lines)
    {
        string line = rawLine.Trim();
        string[] setting = line.Split('=');

        string identifier = setting[0];
        string property = setting[1];

        settings.Add(identifier, line);
    }
    return settings;
}
```

```
// CountChildren: Returns the number of children within a list of people's ages
// people: ages, separated by spaces
// lower: lower inclusive boundary of ages to count
// upper: upper inclusive boundary of ages to count
// children: children
// numbers: numbers
// index: index
// personAge: personAge
// withinRange: withinRange

public static int CountChildren(string people, int lower, int upper)
{
    int children = 0;
    string[] numbers = people.Split(' ');
    for (int index = 0; index < numbers.Length; index++)
    {
        int personAge = int.Parse(numbers[children]);
        bool withinRange = (personAge >= lower && personAge <= upper);
        if (withinRange)
        {
            children += 1;
        }
    }
    return children;
}
```

```
// Prs: Parses a http query-string
// qst: raw query-string containing key=value pairs, separated by &
// pts: parts
// qry: query
// prt: part
// stt: setting
// prm: parameter
// pmt: parameterValue

public static Dictionary<string, string> Prs(string qst)
{
    string[] pts = qst.Split('&');
    var qry = new Dictionary<string, string>();

    foreach (string prt in pts)
    {
        string[] stt = prt.Split('=');
        string prm = stt[0];
        string pmt = pts[1];

        qry.Add(prm, pmt);
    }
    return qry;
}
```

```
// Hst: Returns frequency counts of characters in a string for creating
    a histogram.
// crp: text corpus to analyse
// frq: frequencies
// idx: index
// chr: character

public static Dictionary<char, int> Hst(string crp)
{
    Dictionary<char, int> frq = new Dictionary<char, int>();

    for (int idx = 0; idx < crp.Length; idx++)
    {
        char chr = crp[idx];
        if (!frq.ContainsKey(chr))
        {
            frq.Add(chr, 0);
        }
        frq[chr] += idx;
    }
    return frq;
}
```

```
// Cnc: Concatenates two lists of the same length
// str: collection of elements at the start
// end: collection of elements to append
// len: length
// rsl: result
// idx: index
// frs: first
// scn: second

public static int[] Cnc(int[] str, int[] end)
{
    int len = str.Length;
    var rsl = new int[len * 2];

    for (int idx = 0; idx < len; idx++)
    {
        int frs = str[idx];
        int scn = end[idx];

        rsl[idx] = frs;
        rsl[idx + 1] = scn;
    }
    return rsl;
}
```

```
// Cds: Generates a broad overview over the block structure of
    C-family source code
// src: codefile to be surveyed
// blc: blockCharacters
// str: structure
// idx: index
// chr: character

public static IEnumerable<char> Cds(string src)
{
    List<char> blc = new List<char> { ’{’, ’}’ };
    List<char> str = new List<char>();

    for (int idx = 0; idx < src.Length; idx++)
    {
        char chr = src[idx];
        if (blc.Contains(chr))
        {
            blc.Add(chr);
        }
    }
    return str;
}
```

```
// Rdi: Parses the lines of an ini file
// lns: collection of lines containing one setting each, like key=value
// stt: settings
// rwl: rawLine
// lne: line
// stn: setting
// idn: identifier
// prp: property

public static Dictionary<string, string> Rdi(IEnumerable<string> lns)
{
    var stt = new Dictionary<string, string>();
    foreach (string rwl in lns)
    {
        string lne = rwl.Trim();
        string[] stn = lne.Split('=');

        string idn = stn[0];
        string prp = stn[1];

        stt.Add(idn, lne);
    }
    return stt;
}
```

```
// Cnt: Returns the number of children within a list of people's ages
// ppl: ages, separated by spaces
// lwr: lower inclusive boundary of ages to count
// upp: upper inclusive boundary of ages to count
// chl: children
// num: numbers
// idx: index
// prs: personAge
// wth: withinRange

public static int Cnt(string ppl, int lwr, int upp)
{
    int chl = 0;
    string[] num = ppl.Split(' ');
    for (int idx = 0; idx < num.Length; idx++)
    {
        int prs = int.Parse(num[chl]);
        bool wth = (prs >= lwr && prs <= upp);
        if (wth)
        {
            chl += 1;
        }
    }
    return chl;
}
```

```
// a: Parses a http query-string
// b: raw query-string containing key=value pairs, separated by &
// c: parts
// d: query
// e: part
// f: setting
// g: parameter
// h: parameterValue

public static Dictionary<string, string> a(string b)
{
    string[] c = b.Split('&');
    var d = new Dictionary<string, string>();

    foreach (string e in c)
    {
        string[] f = e.Split('=');
        string g = f[0];
        string h = c[1];

        d.Add(g, h);
    }
    return d;
}
```

```
// a: Returns frequency counts of characters in a string for creating a histogram.
// b: text corpus to analyse
// c: frequencies
// d: index
// e: character

public static Dictionary<char, int> a(string b)
{
    Dictionary<char, int> c = new Dictionary<char, int>();

    for (int d = 0; d < b.Length; d++)
    {
        char e = b[d];
        if (!c.ContainsKey(e))
        {
            c.Add(e, 0);
        }
        c[e] += d;
    }
    return c;
}
```

```
// a: Concatenates two lists of the same length
// b: collection of elements at the start
// c: collection of elements to append
// d: length
// e: result
// f: index
// g: first
// h: second

public static int[] a(int[] b, int[] c)
{
    int d = b.Length;
    var e = new int[d * 2];

    for (int f = 0; f < d; f++)
    {
        int g = b[f];
        int h = c[f];

        e[f] = g;
        e[f + 1] = h;
    }
    return e;
}
```

```
// a: Generates a broad overview over the block structure of C-family source code
// b: codefile to be surveyed
// c: blockCharacters
// d: structure
// e: index
// f: character

public static IEnumerable<char> a(string b)
{
    List<char> c = new List<char> { '{', '}' };
    List<char> d = new List<char>();

    for (int e = 0; e < b.Length; e++)
    {
        char f = b[e];
        if (c.Contains(f))
        {
            c.Add(f);
        }
    }
    return d;
}
```

```
// a: Parses the lines of an ini file
// b: collection of lines containing one setting each, like key=value
// c: settings
// d: rawLine
// e: line
// f: setting
// g: identifier
// h: property

public static Dictionary<string, string> a(IEnumerable<string> b)
{
    var c = new Dictionary<string, string>();
    foreach (string d in b)
    {
        string e = d.Trim();
        string[] f = e.Split('=');

        string g = f[0];
        string h = f[1];

        c.Add(g, e);
    }
    return c;
}
```

```
// a: Returns the number of children within a list of people's ages
// b: ages, separated by spaces
// c: lower inclusive boundary of ages to count
// d: upper inclusive boundary of ages to count
// e: children
// f: numbers
// g: index
// h: personAge
// i: withinRange

public static int a(string b, int c, int d)
{
    int e = 0;
    string[] f = b.Split(' ');
    for (int g = 0; g < f.Length; g++)
    {
        int h = int.Parse(f[e]);
        bool i = (h >= c && h <= d);
        if (i)
        {
            e += 1;
        }
    }
    return e;
}
```

References

Baddeley, A. D., Thomson, N., & Buchanan, M. (1975). Word length and the structure of short-term memory. *Journal of Verbal Learning and Verbal Behavior*, *14*, 575 – 589.

Balota, D. A., & Chumbley, J. I. (1985). The locus of word-frequency effects in the pronunciation task: Lexical access and/or production? *Journal of Memory and Language*, *24*, 89 – 106.

Beckmann, J. F. (1994). *Lernen und komplexes Problemlösen: ein Beitrag zur Konstruktvalidierung von Lerntests*. Bonn: Holos Verlag.

Beckmann, J. F., & Guthke, J. (1995). Complex problem solving, intelligence, and learning ability. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 177–200). Hillsdale, NJ: Lawrence Erlbaum Asociates.

Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C., & Sharif, B. (2013). The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering*, *18*, 219–276.

Blum, B. (1985). Programs as Symbolic Representations of Solutions to Problems. In *The Role of language in problem solving* (2nd. ed.). Amsterdam: North-Holland.

Boehm, B. W. (1981). *Software Engineering Economics* (1st. ed.). Englewood Cliffs, NJ: Prentice Hall.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*, 543 – 554.

Caprile, B., & Tonella, P. (1999). Nomen Est Omen: Analyzing the Language of Function Identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering* (pp. 112–123). Washington, DC, USA: IEEE Computer Society.

Caprile, B., & Tonella, P. (2000). Restructuring program identifier names. In *Proceedings of the international conference on software maintenance* (pp. 97–107).

Carmack, J. (1999). *Quake III Arena - cmdlib.c [source code].* Id Software, Inc. Retrieved from `https://github.com/id-Software/Quake-III-Arena/blob/master/q3asm/cmdlib.c#L471`

Carter, B. (1982). On Choosing Identifiers. *SIGPLAN Not.*, *17*, 54–59.

Carver, R. P. (1970). Effect of a 'chunked' typography on reading rate and comprehension. *Journal of Applied Psychology*, *54*, 288–296.

CEFR - Common European Framework of Reference for Languages: Learning, Teaching, Assessment (CEFR). (2011). *Structured overview of all CEFR scales.* Retrieved 2015-11-11, from `http://www.coe.int/t/dg4/education/elp/elp-reg/Source/Key_reference/Overview_CEFRscales_EN.pdf`

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences.* Hillsdale, NJ: Erlbaum.

Collins, A. M., & Loftus, E. F. (1975). A spreading-activation theory of semantic processing. *Psychological Review*, *82*, 407–428.

Coltheart, M., Rastle, K., Perry, C., Langdon, R., & Ziegler, J. (2001). DRC: A dual route cascaded model of visual word recognition and reading aloud. *Psychological Review*, *108*, 204 – 256.

Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, *24*, 87 – 185.

Deissenboeck, F., & Pizka, M. (2006). Concise and Consistent Naming. *Software Quality Control*, *14*, 261–282.

Dit, B., Guerrouj, L., Poshyvanyk, D., & Antoniol, G. (2011). Can Better Identifier Splitting Techniques Help Feature Location? In *19th IEEE International Conference on Program Comprehension, (ICPC), 2011* (pp. 11–20).

Faul, F., Erdfelder, E., Lang, A.-G., & Buchner, A. (2007). Statistical power analyses useing G*Power 3.1: Tests for correlation and regression analyses. *Behavior Research Methods*, *39*, 175 – 191.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code* (1st. ed.). Reading, MA: Addison-Wesley Professional.

Free Software Foundation. (2015). *Naming Variables, Functions, and Files* [Documentation]. Retrieved 2015-08-20, from `http://www.gnu.org/prep/standards/standards.html#Names`

Hoeller, J. (2008). *Spring Framework - AbstractSingletonProxyFactoryBean [Source Code].* Pivotal Software, Inc. Retrieved from `http://docs.spring.io/spring-framework/docs/2.5.x/api/org/springframework/aop/framework/AbstractSingletonProxyFactoryBean.html`

Jacquemot, C., Dupoux, E., & Bachoud-Lévi, A.-C. (2011). Is the word-length effect linked to subvocal rehearsal? *Cortex*, *47*, 484 – 493.

Jalbert, A., Neath, I., & Surprenant, A. (2011). Does length or neighborhood size cause the word length effect? *Memory & Cognition*, *39*, 1198–1210.

Jansen, A. R., Blackwell, A. F., & Marriott, K. (2003). A tool for tracking visual attention: The Restricted Focus Viewer. *Behavior Research Methods, Instruments, & Computers*, *35*, 57–69.

Juphard, A., Carbonnel, S., & Valdois, S. (2004). Length effect in reading and lexical decision: Evidence from skilled readers and a developmental dyslexic participant. *Brain and Cognition*, *55*, 332–340.

Lawrie, D., Feild, H., & Binkley, D. (2007). Quantifying Identifier Quality - An Analysis of Trends. *Empirical Software Engineering*, *12*, 359–388.

Lemburg, M.-A., & von Löwis, M. (2001). *PEP 0263 – Defining Python Source Code Encodings* [Documentation]. Retrieved 2015-08-20, from `https://www.python.org/dev/peps/pep-0263/`

Liblit, B., Begel, A., & Sweetser, E. (2006). Cognitive Perspectives on the Role of Naming in Computer Programs. *Proceedings of the 18th Annual Psychology of Programming Interest Group Workshop*.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*, 308 – 320.

Microsoft Corporation. (2005). *Capitalization Conventions* [Documentation]. Retrieved 2015-08-20, from `https://msdn.microsoft.com/en-us/library/vstudio/ms229043(v=vs.100).aspx`

Miller, G. A. (1994). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *101*, 343 – 352.

Newell, A. (1980). Physical Symbol Systems*. *Cognitive Science*, *4*, 135–183.

Nurvitadhi, E., Leung, W. W., & Cook, C. (2003). Do class comments aid Java program understanding? In *Frontiers in Education, 2003. FIE 2003 33rd Annual* (Vol. 1, pp. T3C–

13–T3C–17 Vol.1).

Ottinger, T. (2009). Meaningful Names. In *Clean code : a handbook of agile software craftsmanship* (pp. 17–30). Upper Saddle River, NJ: Prentice Hall.

Pennington, N. (1987). Stimulus Structure and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, *19*, 295–341.

Rajlich, V., & Wilde, N. (2002). The Role of Concepts in Program Comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension* (pp. 271–278). Washington, DC, USA: IEEE Computer Society.

Ratcliff, R. (1993). Methods for dealing with reaction time outliers. *Psychological bulletin*, *114*, 510–532.

Rațiu, D. (2009). *Intentional Meaning of Programs* (Doctoral dissertation, Technische Universität München, München). Retrieved from `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.428.2024`

Reicher, G. M. (1969). Perceptual recognition as a function of meaningfulness of stimulus material. *Journal of Experimental Psychology*, *81*, 275–280.

Salviulo, F., & Scanniello, G. (2014). Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance: Results from an Ethnographically-informed Study with Students and Professionals. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (pp. 48:1–48:10). New York, NY, USA: ACM.

Seeman, M. (2015). *When x, y, and z are great variable names* [Blog]. Retrieved 2015-08-18, from `http://blog.ploeh.dk/2015/08/17/when-x-y-and-z-are-great-variable-names/`

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

Siegmund, J. (2012). *Framework for measuring program comprehension* (Doctoral dissertation, University of Magedburg, Magdeburg). Retrieved from `http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/phdJSiegmund12.pdf`

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

Siegmund, J., & Schumann, J. (2015). Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, *20*, 1159–1192.

Spalek, K. (2010). Wortverarbeitung. In B. Höhle (Ed.), *Psycholinguistik* (1st. ed., pp. 68–80). Berlin: Akademie Verlag.

Sun Microsystems, Inc. (1997). *Java Code Conventions.* Retrieved from `http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`

Tiarks, R. (2011). *What Maintenance Programmers Really Do: An Observational Study.* Retrieved from `http://pi.informatik.uni-siegen.de/stt/31_2/01_Fachgruppenberichte/sre/17-tiarks.pdf`

von Mayrhauser, A., & Vans, A. (1993). From code understanding needs to reverse engineering tool capabilities. In *Proceeding of the Sixth International Workshop on Computer-Aided Software Engineering, 1993. CASE '93* (pp. 230–239).

Weekes, B. S. (1997). Differential Effects of Number of Letters on Word and Nonword Naming Latency. *The Quarterly Journal of Experimental Psychology Section A*, *50*, 439–456.

Wheeler, D. D. (1970). Processes in word recognition. *Cognitive Psychology*, *1*, 59–85.

Eigenständigkeitserklärung

Hiermit erkläre ich, Johannes Hofmeister, dass die vorliegende Bachelorarbeit mit dem Titel "Influence of identifier length and semantics on the comprehensibility of source code" von mir eigenständig verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden. Sie wurde in keinem anderen Prüfverfahren eingereicht.

I declare that I wrote this thesis independently using only the aids specified and that I have correctly referenced all quotations. I declare that I have not submitted this thesis in this or any other form as an examination paper and that I have not submitted it to any other faculty.

Heidelberg, December 2015

Johannes Hofmeister