Investigating the Perception and Effects of Misunderstandings in Java Code

Chris E.I. Langhout

Technische Universiteit Delft

```
                                        s*
                                        w,
                                    a={x      ,x},
                                    b={x      ,y},
c={y,x},d={y,y};s l[]={&b,&d,w      (w(w(w(w(w(&d))))),w(&d),&c,&d,w(w(w(
&d))),&c,w(&b),&d,&a,z(&d),z(w      (w(w(w(w(&d))))),&b,&b,w(&d),&a,&b,w(&
d),                                                                  z(w
(w(                                                                   w(w

(&d)))))),w(w(w(&d))), &b,&c,&d, &d,&a,&c ,w(w(&d)), &c,z(&b) ,w(&d),w(&a)
};s* C(s* a, s* b) {s* _=malloc(s(s));_(_)=a;_((t)_)=b;return _;}e k2=s(l
);p                                                                  f(p
a,p                                                                   b)

                    {    e k;s d;p v,r,          q,i,C,c,u,g,m=t(   )
_               (   _(_(w)));C=t()           _(_(&l[fread(&k,s(e   )
,s             (   e),stdin)]));v              =C(_,__);d=l[k=(e   )
C             (   k2,k)];c=(u=a(                b,_),i=(t()_((t   )
_           (   _(w))))(_,_(_(o))              ),(a(_(_((t)o)),(t(   )
_         (   _((t)o)))(_,__)                    ))(b(_((t)_((t)o)   )
,        (   t()_((t)_((t)o)))                  (_,__)),_));{p a=t(   )
_        (   (t)_(o));{p b=                       C(_,i(u(_,__),_)   )
;       {   p u=C(_,(t()_(_(o                   ))(_,__));_(_(_(w))   )
=      (   s*)i(__,_(_(                           _(w))));r=b(m(c,_   )
,C     (   m(_,(t()_((                            t)_(_(w))))(_,__)   )
,     (   f)));v=b(_,v);i                         =b(a(_,__),_);g=(b   )
       (     a(m(c(_,__),__                            ),_),_);q=u(_((t   )
_    (   o)),_)((t()_(                              (t)_(_(w))))((t(   )
_    (   _((t)o)))(_(_((t                          )_(w))),(t()_(_(((t   )
_    (   w))))(_,__))(                               _,__),_)(_,__),_   )
;_   (   _((t)_(w)))                    =           (s*)u(_((t)_(o)),_   )
       (    (t()_((t)_(_(w))))                        (_,__),_)(_(_((t)o)   )
,_   (   _((t)_(w))));}                             }_((t)_(_(w)))=(s*   )
q    (   a=(t()_((t)_(_(w)))                      )(_,__),_((t)_(_(w)))   )
       ;    fwrite((k=k,&k2)                             ,s(e),u(_,g)(s(e   )
,    (   e)s(s[s(s[s(s[s(s[s                            (s)])])])),stdout   )
       ;     fwrite((k2=k,&k)                             ,s(e),u(__,g)(s(e   )
,     (   e)s(s[s(s[s(s[s                               (s)])])])),stdout   )
;_    (   _(_((t)w)))=(s*)                            u((t()_(_(_((t)w)))   )
       (   _,__),_(_(_((t)w                                ))));_(_((t)_(w))   )
=     (   s*)q(a(_(_((t)                                o)),_(_((t)_(w)))   )
,_     (   _((t)_(w))));}                            _((t)_(_((t)w)))=(s*   )
         (    t()_(_(_((t)w)))                            )(_((t)_(_((t)w))   )
,u       (   (t()_((t)_(_((                            t)w))))(_,__),_((t   )
_         (   _((t)w)))));g                             =u((q=u(C(__,(t(   )
_          (   _(_((t)w))))(_((t                      )_(_((t)w))),_)),_)   )
             (    _,__),g);v=g(                            __,i(u(_,i(f,_)   )
             (   _(_((t)o)),_(                       (t)_((t)o))),v)   )

      ,r=                                                          q(_
      ,g(                                                          f,i
      (v(f,(_(_(_(w)))=(s*)m,_)),r)));return q(_,v)(r(g(__,a),g(__,b)),r);}

      int                                                           main
      (){w=C(C(h,h),C(h,h));return       printf((e*)f(_,_)("OK\n","\n ^ Error\n"
                                      ""  ))
                                      ;}
```

TU Delft

Investigating the Perception and Effects of Misunderstandings in Java Code

by

Chris E.I. Langhout

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday May 27th, 2020 at 2:00 PM.

Student number:	4281705	
Thesis committee:	Prof. Dr. A. van Deursen, Faculty EEMCS	TU Delft, chair
	Dr. M.F. Aniche, Faculty EEMCS	TU Delft, supervisor
	Dr.ir. F.F.J. Hermans, LIACS	Leiden University

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Investigating the Perception and Effects of Misunderstandings in Java Code

by

Chris E.I. Langhout
4281705

Abstract

Although writing code seems trivial at times, problems arise when humans misinterpret what source code actually does. One of the potential causes are "*atoms of confusion*"; the smallest possible patterns of misinterpretable source code. The misunderstandings and errors have been studied in past for the C programming language. They are found to occur in many large projects and style guides. In this work, the existing tested set of atoms of confusion has been translated to Java. With this new set, our aim was to find out what atoms of confusion hinder the comprehensibility of Java programs. Additionally, we wanted to find out how these confusion patterns are perceived. To this end, the new code snippets are used in a two-fold experiment. The first part of the experiment asked the participant to write down the output of a code snippet. The results of this showed us that 7 out of the 14 translatable atoms are the cause of misunderstandings for students. We measured a significant increase in mistakes caused by the atoms of confusion. In the second part of the experiment we asked the participants to compare two code snippets on how confusing they are. One code snippet included the confusing pattern, while in the other, the pattern is avoided. Results showed us that these students also perceive the atoms of confusion as being more difficult to understand. The combined results show us the significance of these atoms of confusion, and show us examples of situations where we cannot assume programmers to simply grasp the meaning of what we write down as code. The code snippets for the experiment, scripts, and data used for this experiment are provided in an online appendix [29].

Thesis committee:	Prof. Dr. A. van Deursen, Faculty EEMCS	TU Delft, chair
	Dr. M.F. Aniche, Faculty EEMCS	TU Delft, supervisor
	Dr.ir. F.F.J. Hermans, LIACS	Leiden University

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Preface

Thank you for taking your time to read through my thesis report. The making-of took me more time than I like to admit. Back in July 2018 I had my first meetings with my supervisor, Maurício. He had quite some nice ideas that seemed more than interesting to explore during my master thesis project. We decided to go into the direction of a static analysis tool for Javascript, inspired by the abilities of the PVS-Studio Analyzer by Viva-64 [1].

After some time, however, we decided to change directions. To be able to tell the usefulness of a tool like this, we would like to research what rules are needed. From javascript, we moved to a combination of C, Java and Python, and took many inspiration from the awesome research done by Gopstein et al. [21]. Thanks a lot to the amazing availability of the resources those researches used and provided [2].

Another moment of realization came, we deemed focussing on 3 languages at the same time unfeasible. We would get better results, and save additional effort in the experiment setup if we only focus on Java. This also means that all TU Delft computer science students are feasible candidates, as they all have a baseline knowledge of Java. Many thanks to all the students that took the time to fill in the experiment!

The journey of my master thesis is now finally coming to an end. Many thanks to Maurício for sticking with me, all the discussions and meetings we had in the beginning, the support and motivation throughout the project, and the reviews and feedback on the process and the result. I am really happy with the result and really happy with the role you played.

Another person to whom I own many thanks, is Martijn. You have supported my work al lot, and encouraged my creativity. You prereading my work always resulted in helpful points of feedback, you are always willing to listen, discuss and think along.

Many thanks to Dereck as well, working through the summer of 2019 was way better doable together. Full working days at the 4th floor were much better to sustain together. Thanks as well to the rest of the 4th floor gang. The code snippets I used in my experiment improved in quality during the discussions about them, and mostly, I explored the workings of the most specific details of them with you.

Lastly, I would like to thank all others that have read my thesis, have proposed to preread, or will read my work. Your support was very helpful.

Chris Langhout
Delft, the Netherlands
May 13, 2020

[1] https://www.viva64.com/en/pvs-studio/
[2] https://atomsofconfusion.com

Contents

List of Figures

1

Introduction

When creating, adapting, maintaining and reviewing source code, it is necessary to understand what the code does [21, 38, 40]. In contrast to natural languages, programming languages have an unambiguous meaning for a syntactical valid piece of code [4]. However, developers do not necessarily draw the correct conclusions on the behavior of a piece of code [21]. They often mistake the meaning of code and misjudge the program's behavior, which can lead to errors. This shows a need for improving code understandability [30].

We can find numerous examples of misunderstanding causing code. We can observe that people tend to disagree about what code is understandable and what code is not.

Different programming languages give the software developer many ways of writing a solution to a problem. For example, the simple task of converting a boolean `true` or `false` value into a numeric int value can be coded in a vast amount of ways. One example of this can be found in the difference in answers given to the question *How to convert boolean to int in Java?* [1]. Eight different answers are displayed in Figure 1.1, ordered by amount of votes. The solutions show a great amount of variation in logic, readability and understandability. Using the `ternary if` operator (option 1.) is by comments considered 'most readable', and received the most votes, making it the most accepted solution according to the rules of the community the question was posted in. This is in contrast with the findings of Gopstein et al. [21], which show that the use of the *conditional operator* atom is found to be significantly confusing. (The ternary if operator is the only conditional operator used in the code examples for the conditional operator atom.) Furthermore, the author of the 8th answer starts his answer with "*If you want to obfuscate, use this:*" showing that the intention of this answer is not readability, but rather showing of a less known alternative to solve the question.

```
int i;

1. i = b ? 1 : 0;

2. i = (Boolean b).compareTo(false);

3. i = Boolean.compare(b, false);

4. i = -("false".indexOf(""+b));

5. i = 5 - b.toString().length;

6. import org.apache.commons.lang3.
                BooleanUtils;
   i = BooleanUtils.toInteger(b);

7. if(b){
      i = 1;
   } else{
      i = 0;
   }

8. i = 1 & Boolean.hashCode( b ) >> 1;
```

Figure 1.1: Examples from StackOverflow answers on how to convert boolean `b` to an int in Java [1]

Discussions about misconceptions happen regularly. On the 14th of June 2019, user Jonathan Wakely starts a discussion on the bug-tracker of GCC about introducing warnings when people use the boolean operator ∧ with integer literals.[2] The ∧ operator represents a bitwise XOR operation in most programming languages. Instead, people confuse the symbol with the mathematical representation of a power. The author states:

[1] https://stackoverflow.com/questions/3793650/convert-boolean-to-int-in-java [Accessed July 25, 2019]
[2] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90885

> "*There's nothing wrong about implicit fallthrough, misleading indentation, ambiguous else, or missing parentheses in nested logic expressions either. But people get it wrong all the time.*
>
> *I can't see a good reason to write 2∧16 when you mean 18, or 10∧9 when you mean 3, so it's probably a bug. And there's an easy workaround to avoid the warning: just write the exact constant as a literal, not an XOR expression.*"[3]

Other people jump in to show examples of occurrences of this particular pattern on GitHub and other source code hosting sites. Several responses provide suggestions to specific cases when a warning should, or should not be raised, depending on the use of literals or not. These examples show that readable code is a relevant topic, and that the details up to the smallest code snippets can make a difference in understandability.

The motivation for this study comes from the work of Gopstein et al. [21]. They observe a trend in notable software bug examples, where the failure is caused by *"a single, well-contained, programming error at the syntactic or semantic level, rather than the algorithmic or system-levels of the project"*. Seeking validation for reoccurring misunderstandings caused by these small code patterns, an experiment is set up. **Atoms of confusion**, 'atoms' for short, are defined by Gopstein et al. [21] as minimal portions of code that cause a person and a machine to come to different conclusions on the output. Castor [12] expands this definition by formalizing atoms as: precisely identifiable, likely to cause confusion, replaceable by a functionally equivalent code pattern that is less likely to cause confusion, and indivisible. Atoms of confusion do specifically not include non-deterministic, undefined/non-portable, computational, and API related code, since the target of the atoms is programmer mistakes caused by misunderstanding [21].

With 73 participating students, Gopstein et al. [21] show a significant increase in misunderstanding caused by the patterns, opposed to code without the atoms of confusion. To show the impact of these confusion patterns, another experiment with 43 participants and larger confusing programs was performed and described within the same work [21]. The results of this second experiment show statistical significant higher error rates in the evaluation of obfuscated variants of programs.

The goals of this work are to generalize the knowledge on atoms of confusion to the Java programming language, show the perception of developers towards the atoms of confusion and gain insights in what makes the atoms confusing. The study is geared towards TU Delft students, to show where confusion is likely to occur and to raise the discussion on areas of interest for education. A two-fold experiment is used to collect the necessary data and to extend the findings of previous work with data on an additional programming language. First, the effects of the atoms of confusion will be evaluated by an experiment similar to the original research. Second, participants will be asked about their perception of the included atoms fo confusion by showing both the code example with the candidate confusion patterns, as well as the variant where this potential confusion pattern is left out.

The complete design of the experiment, along with the rest of the scientific methodology can be found in Chapter 3. Chapter 4 will display the results of the experiment, will go into detail on the outcomes and raise a discussion for each individual atom. Finally, in Chapter 5, a conclusion is drawn and future work is discussed.

[3] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90885#c6

2

Related Work

This chapter discusses topics related to program comprehension and code readability, specifically atoms of confusion and misunderstandings in source code, and how this impacts programming education.

2.1. Program Comprehension

Program comprehension is a widely explored domain in computer science. For this study, we are mainly interested in the skill level in program comprehension of novice programmers. We are interested in research that finds factors of influence on program comprehension, and explanations of code readability for novice programmers.

In 1985, Bonar and Soloway [9] stated that *'many programming bugs can be explained by novices inappropriately using their knowledge of step-by-step procedural specifications in natural language'*. We can take away that certain bugs can be caused by lack of expertise. Finding out what programming code does, or what code is needed, our intuition sometimes tries to connect our present knowledge from other fields to find a solution. In this case, knowledge of natural language tricked participants into making mistakes when writing code [9].

We want to find out how well students understand code that is shown to them. A study by Brooks [10] provides a theory that predicts three sources of comprehension ability differences in programmers: programming knowledge, domain knowledge and comprehension strategies. The provided theory aims to explain four sources of variation in behavior: the kind of computation, the intrinsic properties of the written code and documentation, the reason why the documentation is provided, and differences between the individual participants. The provided theory explains comprehending of a program as the ability to reconstruct mappings from a problem domain, through intermediate domains, into the programming domain. We are interested in the comprehension of very small snippets of code, that lack the initial problem mapping. Although few code is to be processed, it provides an interesting perspective on the research of comprehension on (small) snippets of code.

We are interested in research methodologies to measure how well programmers understand various snippets of code. Ajami et al. [1] use an experimental platform fashioned as an online game-like environment to measure how quickly and accurately 220 professional programmers interpret code snippets with similar functionality but different structures. The findings include that there is no relation between errors made and time taken to understand the snippets, but snippets that take longer to understand are considered harder [1]. When a snippet contains a `for` loop, the code is considered much harder to understand compared to snippets containing `if`s. Snippets with predicates become harder to understand when negations are present, and `for` loops counting down are harder to understand than loops that count up. This shows that the slight differences in the way of expressing predicates can be measured when compared against the use of known idioms. The syntactic structures of code are shown to not necessarily take up the biggest part in the measurement of complexity of code. They also found that the metrics of time to understanding and the amount of errors made are not necessarily related. This means that the amount of errors made is not related to how long a participant takes to solve a problem.

Atom Name	Obfuscated	Transformed
Infix Operator Precedence	`2 - 4 / 2`	`2 - (4 / 2)`
Post-Increment/Decrement	`V1 = V2++;`	`V1 = V2; V2 += 1;`
Pre-Increment/Decrement	`V1 = ++V2;`	`V2 += 1; V1 = V2;`
Constant Variables*	`V1 = V2;`	`V1 = 5;`
Conditional Operator	`V2 = V1 == 3 ? 2 : 1;`	`if (V1 == 3) { V2 = 2; } else { V2 = 1; }`
Arithmetic as Logic*	`(V1 - 3) * (V2 - 4)`	`V1 != 3 && V2 != 4`
Logic as Control Flow	`V1 == ++V1 || ++V2`	`if ( !(V1 + 1) ) { V2 += 1;} V1 += 1`
Repurposed Variables	`int main(int argc, char **argv) { argc = 7; ...`	`int main(int argc, char **argv) { int V1 = 7; ...`
Dead, Unreachable, Repeated*	`V1 = 1; V1 = 2;`	`V1 = 2;`
Change of Literal Encoding	`V1 = 013`	`char V1 = 23;`
Omitted Curly Braces	`if (V1) F1(); F2();`	`if (V1) { F1(); } F2();`
Type Conversion	`3/2;`	`trunc(3.0/2.0);`
Preprocessor in Statement	`int V1 = 1 #define M1 1 +1;`	`#define M1 1 int V1 = 1 + 1;`
Macro Operator Precedence	`#define M1 64 - 12 * M1`	`2 * 64 - 1`
Assignment as Value	`V1 = V2 = 3;`	`V2 = 3; V1 = V2;`
Reversed Subscripts	`1["abc"];`	`"abc"[1]`
Comma Operator	`V3 = (V1++, V1);`	`V1++; V3 = V1;`
Implicit Predicate	`if (4 % 2)`	`if (4 % 2 != 0)`
Pointer Arithmetic*	`"abcdef"+3`	`"abcdef"[3]`

Table 2.1: Atoms of confusion from Gopstein et al. [21]. Atoms marked with a * failed to meet statistical significance.

Going into more depth on program understanding, the difference in program comprehension between different representations of programs is researched by Bednarik and Tukiainen [6]. In their work, eye-tracking is used to help find program comprehension strategies. Their experiment makes use of a tool that represents a program in the program code, and an animated visualization of the code. In the experiment with 18 novice programmers, two different comprehension strategies were observed. In general, participants with less previous programming experience used the provided animation to gain insights in the program's behavior. However, the more experienced participants looked at the provided source code first, and used this to form a hypothesis of the program. The animation is then used only to verify and improve their formed beliefs.

Now that we found out what makes programs harder or easier to understand, we can find ways to improve readability of source code. To increase understandability of programs, Jbara and Feitelson [26] researched and measured the impact of code regularity. Repeating similar structures may significantly reduce complexity, since readers notice repetition. Therefore, future occurrences become easier to understand [26]. Controlled experiments are used to show this reduction in complexity. Participants are shown three different implementations of a program with varying percentages of repetition within the code. Significant regularity is shown to improve comprehension, demonstrating impact of repeated structures when investigating code comprehension. Furthermore, Jbara and Feitelson [26] state that "*the fact that regularity may compensate for LOC and MCC demonstrates that complexity cannot be decomposed into independently addable contributions by individual attributes*".

Research on program comprehension can also be used in a total opposite way. Intentionally decreasing the readability of source code is used as a security measure to prevent attackers from understanding the specific implementation of a program. Code obfuscation techniques are used to hide the meaning of source code implementations by transforming the program into an obfuscated program while keeping the same observable behavior [4, 14]. Code obfuscation makes understanding the meaning and conceptual purpose of code harder for humans. Collberg et al. [14] stated in their research from 1997 that "automatic *code obfuscation* is currently the most viable method for preventing reverse engineering." In their work, they analyse the working of different code obfuscation transformations, and measure the effect they have on human understandability, how well attacks are prevented, and what the costs of these methods are. For our own research, the most interesting part is the measuring of the confusing effect the obfuscations have on humans. Their method, however, is based on *Software Complexity Metrics*. The field of research around software complexity metrics is definitely interesting regarding code understandability. Various different metrics exist that provide a score based on code maintainability, understandability, or quality [31, 36]. While the potential of metrics to determine the quality of software is great for production code, the code snippets we use for this experiment are so limited in size, the metrics are not well applicable.
Turning our focus back on research on code obfuscation, techniques that are used for the obfuscation of code can be used for finding areas of interest in code comprehension [4]. By understanding what makes code less readable, we can draw conclusions about what code constructs are understandable and which to avoid. This is shown by the research of Gopstein et al. [21]. The main source of the code patterns used for the atoms of confusion comes from a contest on writing obfuscated code called IOCCC (the International Obfuscated C Code Contest). The original 19 atoms of confusion, on which this research is based on, are described in Table 2.1. The obfuscated column shows example code that includes the pattern causing the atom of confusion, and the transformed column shows example code where this pattern is avoided, while the code behavior remains equivalent. To show the real world relevance of these selected atoms of confusion, follow up research from Gopstein et al. [23] shows that the 15 atoms that were proven to be confusing, occur in practice once per 23 lines. Their research is based on the analysis of 14 of the most popular and influential C and C++ software projects. Medeiros et al. [30] also researched the rate of occurrences of most of the atoms and show that all but one occur in the analyzed projects. They based their numbers on a set of 50 open-source C projects using a mixed method approach including repository mining and developer surveys. Four of the 12 atoms researched by Medeiros et al. [30] are shown to be commonly used.

A similar approach to researching patterns of code that cause misinterpretation can be found in the work of Dolado et al. [16]. This work provides insights in misinterpretations caused by code that has side-effects. The researched code fragments are comparable to code examples used in this study. The atoms *pre increment decrement*, *post increment decrement* and *logic as control flow* have most similarities, as they also make use of expressions with side-effects.

2.2. Misconceptions

Other misconceptions that happen in reading and writing source code can be found that are similar to atoms of confusion, but not really fit the definition. Beller et al. [7] found that 'micro-clones' are much more likely to have an error in the last line or statement than the previous lines [7, 8]. Micro clones are explained as tiny, duplicated pieces of code, for example:

```
x += other.x;
y += other.y;
z += other.y;
```

By analyzing open source projects and detecting these situations, they show that the last occurrence is more likely to contain the mistake, whenever a mistake is present. In follow up research, the cause of these kinds of errors is investigated, and "action slips" (mistakes made during routine tasks [3]) are shown to have influence. The *PVS-Studio* tool, used to find the researched occurrences, can help in detecting these error-prone situations.

When comparing natural language to programming languages, programming languages have more repetition and higher predictability. The reason behind this, as found out by Casalnuovo et al. [11] is not due to grammatical constraints in programming languages and has to be resulting from developers' choice. ARCC: Assistant for Repetitive Code Comprehension is a tool made by Nunez et al. [33] to use repeating structures in helping understand programs. The effect of repeating structures might be of influence to results for the *Dead, Unreachable, Repeated* atom.

The perception on readability of 11 different coding style practices is tested by dos Santos and Gerosa [17]. Using a custom tool, participants were shown a pair of code examples, one violating the coding practice while the other complied with it. They find that 7 out of the 11 tested practices increase readability, 1 decreased readability, and the remaining 3 did not present statistically significant effects.

Misconceptions in the object oriented programming approach, that are related to education of the subject, are researched by Holland et al. [24]. Six misconceptions are identified and characterized within a teaching environment. The main focus lies on the educational part of these misconceptions and how to educate new students to avoid these misconceptions.

In the study by Bonar and Soloway [9] bugs produced by novice programmers were studied. They discovered the influence of natural language preprogramming knowledge on the created code quality. The experiment participants encounter an obstacle when solving a programming problem, and use their knowledge of natural language and keywords of the programming language to try and work around [9, 35]. Pane et al. [35] study natural tendencies non-programmers have when solving programming problems in order to guide the design of future programming languages, that will be easier to learn and use for beginners, since they more closely resemble the way they tackle problem solving.

A lot has changed since the first programs were written. Some of the atoms of confusion were originally encouraged for performance gains or even necessary due to limitations in length of variables or functions due to memory size. Nowadays, maintainability is much more valuable than the performance gained by using these obfuscated patterns [23]. Maintenance of code bases takes approximately 40% to 80% of software costs [20]. In order to keep maintainability of code bases high and assure quality of the source code, code reviews are often used. Code reviews are not easy to perform: reviewers often have misunderstandings or confusions about the code being reviewed [5, 13, 18]. The main motivation for reviewing code is to find defects. Reviews also have a positive effect on transfer of knowledge, awareness of the team, and exploration of different solutions for problems [5]. Ebert et al. [18] took a look at confusion in code reviews by examining the comments left by reviewers. What they show is that reviewers often do not understand the context of the code change well, which has an impact on the understanding of the code. However, reviewers are decently well in detecting and pointing out sources of confusion.

Tools meant to increase readability and reduce confusion of program code already exist. In particular, static analysis tools are used to check program code by using a set of rules to find defects and styling issues, without running the code [41]. Depending on what programming language, what tool is used, and what rules are enabled, atoms of confusion can be avoided or introduced. For the C programming language, compiler warnings can be enabled to warn for potentially confusing or erroneous code. For example, since GCC version 6, a warning for misleading indentation exists, namely, the by

using the flag `-Wmisleading-indentation`[1]. Using GCC (version 7.4.0) to compile an example of the *Remove Indentation* atom from the dataset of Gopstein et al. [22], namely:

```
#include <stdio.h>

void main() {
  int V1 = 5, V2 = 5;
  while (V2 > 0)
    V2--;
    V1++;
  printf("%d\n",V1);
}
```

The result of compiling this with GCC without any additional flags succeeds without any output. If we pass the `-Wall` flag (to list all warnings) however, a very detailed output is presented on the misleading indentation.

```
17_remove_indentation_atom_obf.c:7:4: warning: this 'while' clause does not guard... [-Wmisleading-indentation]
    while (V2 > 0)
    ^~~~~
17_remove_indentation_atom_obf.c:9:7: note: ...this statement, but the latter is misleadingly indented as if it were guarded by the 'while
      V1++;
      ^~
```

The first line warns about the missing `{}`-brackets, and the note after that notifies us about the misleading indentation.

For Java, due to the design of the language, some of the atoms of confusion that do exist in C are impossible to recreate. For example, the implicit predicate atom is nearly nonexistent in Java since the condition of an if-statement requires a value with the boolean type. The atoms that have no translation to Java are discussed in more detail in Chapter 3.1. Additionally, tools specifically targeting the quality of Java source code exist. Well known tools, such as FindBugs [2], PMD [3] and Checkstyle [4] provide numerous rules that help maintain a coherent code style. Research by Flanagan et al. [19] provide an extended static checker that can help in finding common programming errors.

Johnson et al. [27] researches why software developers do not use static analysis tools, despite their proven benefit. The main reason they found is related to the configuration of these tools. Often, default configurations provided by the tools result in false positives. Other reasons include poorly presented output and not being integrated in the existing workflow [27].

[1] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
[2] http://findbugs.sourceforge.net/
[3] https://pmd.github.io/
[4] https://checkstyle.sourceforge.io/

3

Methodology

The goal of this study is to measure the impact of atoms of confusion in Java code among first year students. To that aim, we propose the following research questions:

RQ1 Which atoms of confusion hinder the comprehensibility of Java programs, and to what extent?
RQ2 How do students perceive confusion in Java programs that include atoms of confusion, as opposed to the translated, confusion-free, Java programs?

Figure 3.1 illustrates our methodology. To answer these research questions, we first determine what code constructs we want to test. We take the questions from the experiment of Gopstein et al. [22] and manually translate them from C to Java program code. This process is described in Section 3.1. Section 3.2 describes the online experiment for the collection of data. The experiment consists of two parts. The first part focusses on the effects of atoms of confusion, asking the participant what they believe the code will print out. The answers are checked against the actual program output. The second part shows the two variants of one selected atom of confusion instance, and we ask the participant to compare the two on level of confusion. Furthermore, we ask the participant to indicate what makes the one more confusing than the other. We recruit TU Delft Computer Science and Engineering students to fill in our survey. How participants are solicited is described in Section 3.3, and the analysis of the resulting data is described in Section 3.4 of this chapter.

3.1. Measuring Atoms of Confusion

In this section, we describe what we want to measure in this experiment. We take the atoms of confusion from the experiment of Gopstein et al. [21] as a basis.

We devised a list of atoms of confusion in Java, based on Gopstein et al. [21][1] work on atoms of confusion in C. The list of Gopstein et al. [21] contains 19 atoms, of which 15 were shown to be

[1] https://atomsofconfusion.com/

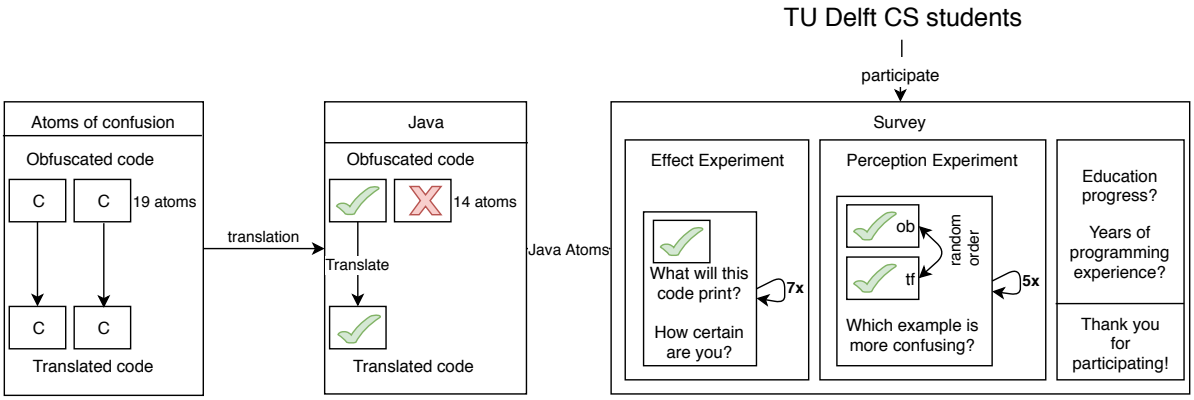

Figure 3.1: Design of the study.

Atom name	C obfuscated example	Reason why translation is not possible
Implicit Predicate	`if (4 % 2)`	Java requires that the condition of the if statement is a boolean value. Non boolean values will be rejected.
Macro Operator Precedence	`#define M1 64 - 12 * M1`	Java does not have support for macro support.
Pointer Arithmetic	`"abcdef"+3`	Pointer arithmetic is not possible.
Comma Operator	`V3 = (V1++, V1);`	The comma symbol is not an operator.

Preprocessor in Statement	`int V1 = 1`
	`#define M1 1`
	`+1;`
		Preprocessing is not part of the standard language specification of Java.

Assignment as Value	`V1 = V2 = 3;`	Java does not allow this behavior, the type system disallows this atom.
Reversed Subscripts	`1["abc"];`	This syntax is invalid in Java.

Table 3.1: Descriptions where translations of atoms are not possible

significantly confusing. The existing set of atoms from Gopstein et al. [22], listed in Table 2.1, resulted in a set of different code snippets described in the C programming language. The code snippets are, in essence, short programs with simple logic, and are affected by their respective atom of confusion. For each of the 19 atom of confusion, 3 different pairs of snippets are included. Each pair consists of one code example that includes the pattern that is described by the atom of confusion, and one snippet that is free of the confusion pattern. Both versions (affected and not affected by the atom of confusion) represent the same program and have the exact same behavior. The resulting set contains of 19 atoms of confusion, each with 3 pairs of code examples, adding up to $19*3*2 = 114$ code examples.

For this study, we explored which atoms have a Java equivalent. To attempt to create a complete set of confusing atoms, specialist knowledge of the selected programming language is needed [12]. We created a set of 80 code examples, based on the dataset described above. The translation of the atoms of confusion from C to Java consists of multiple steps. In order to stay in line with the original study, the programming language related translations will be as similar as possible to the behavior of the original sources [22]. In C, a program has to start with `void main (){`. This part is replaced by `class Snippet { public static void main (String[] args) {`, since for a Java program to work we need a class, and a more specific main method. The print statements are replaced by the language specific counterpart. All C code atoms are converted to Java atoms, changing as little code as possible. While some of these changes are translations between the languages, others require alternative solutions beyond trivial translations.

We revisited the atoms with experienced MSc students in order to refine them. Each resulting atom is inspected on actual behavior. Possible pitfalls are analyzed and if unrelated to the targeted confusion in this atom, the code is adapted to avoid this as much as possible. Aside from this, additional pitfalls were taken out by carefully studying the errata[2] Gopstein et al. [21] provided after completing their study. The final result of this process can be found in Appendix A.

The cases in which the original atoms could not be translated are described in Table 3.1. It was not possible to mimic the *Implicit Predicate* atom, since the type system requires boolean values for predicates. The type system also prevents the *Pointer Arithmetic* and *Assignment as Value* atoms from having a suitable Java translation. In the case of *Pointer Arithmetic* a number cannot be added to a String type, and in the *Assignment as Value* case, the inner assignment does not have a valid return type for the outer assignment. The *Macro Operator Precedence* and *Preprocessor in Statement* atoms are not translated since preprocessing and macros are not part of the language specification. Finally, the *Comma Operator* and *Reversed Subscripts* atoms cannot be translated as the syntax is not present, and no similar behavior could be found.

[2] https://atomsofconfusion.com/2016-snippet-study/errata.html

The following section will describe the mentionable differences between the code examples provided by Gopstein et al. [22] and the created Java equivalents. Code examples 26 and 54 were simplified, to prevent interference with their purposed confusion. Two code examples were removed: code example 52 from the dataset of Gopstein et al. [22] related to *Change of Literal Encoding* is removed since it requires knowledge of the ASCII character table to give an answer. A wrong answer will therefore be much less likely related to the targeted confusion. Example 59 from the original dataset, related to *Type Conversion* is removed because no trivial and small translation to an unsigned cast was found. All atoms are manually checked against the original source and it was ensured that the behavior of the transformed code examples is equivalent to their obfuscated counterparts. The translated part of snippet 53 was adapted to match the behavior in the obfuscated part. Table 3.2 shows these differences in more detail.

To illustrate the results of the translation process, Table 3.3 shows the translation per atom of confusion. The full translated dataset can be found in Appendix A.

3.2. Design of the Study

The study consists of two parts, which will answer the first and second research question respectively. In the first part of the study, the participant is shown one of the code examples, and is asked to evaluate this code and predict an output, based solely on their own thinking process. In the second part, the participant should form an opinion on the obfuscated and transformed code examples, and is asked to explain the differences. The experiment in this study is executed online, using the SurveyGizmo[3] platform.

The survey starts off with an introduction on the study and an approximation of time needed. The structure of the survey is briefly touched upon, and some guidelines are given. The participant is asked to fill in the answer solely based on own knowledge, so without help of others and tools. The Furthermore, the guide specifies several points of information that can be summarized as:

- There is no time pressure, but the participant is encouraged to not stick to one question for too long.
- No syntax errors are present in the given code. If the participant does think errors are present, he/she is asked to explain in the comments where this error would take place.
- Encouragement not to use a calculator, the computer or a search engine for finding answers during the experiment.
- It is not possible to go back to previous questions.
- Discouragement of taking a break while filling in the survey.
- Not using the back- or refresh button of the browser, as this prevents finishing the survey.

These guidelines are based on the guidelines used and provided by Gopstein et al. [21] in their original study. The last text before the *Next* button read that: for the integrity of the study, the answers of the participant should not be influenced by external factors. The exact wording in the introduction of the survey can be found in Appendix C, Figure C.1. The estimated total time required for filling in the survey is 15 minutes, and consists of 12 tasks, 7 for the first part and 5 for the second.

3.2.1. Part 1: Effect

The first part of the survey is based on the study of Gopstein et al. [21]. We want to understand whether the atoms of confusion hinder the understanding of the students. We showed 7 tasks to each participant. Each task shows a code example and asks the participant to write down what this program will print when executed. From our database of snippets, the program showed is picked by randomly selecting an atom of confusion and whether or not this is affected by the confusion. For each atom of confusion, two to three different pairs of code example snippets are possible, with one variant containing confusion and one without.

In addition to the question on what the code will print, the participants are asked how certain they are of their answer. This question can indicate when the participant answers a question correctly while also being confused by the code, or even worse, when the participant answers a question wrong, believing he understood the code. This checks if the participant understands the code, rather than

[3] https://app.surveygizmo.com

	Old C Code Example	New Java Code Example

code example 26
 	
simplified
	`void main() {`
	`  int V1 = 2;`
	`  int V2 = 3;`
	`  int V3 = 1;`
	
	`  int V4 = (V1 == 2 ?`
	`  (V3 == 2 ? 1 : 2) :`
	`  (V2 == 2 ? 3 : 4));`
	
	`  printf("%d\n", V4);`
	`}`
		`class Snippet {`
		`  public static void main(String[] args) {`
		`  int V1 = 3;`
		`  int V2 = 5;`
		`  int V3 = 2;`
		
		`  int V4 = V1 == 3 ? V2 : V3;`
		
		`  System.out.println(V4);`
		`  }`
		`}`

code example 54
 	
simplified
	`void main() {`
	`  int V1 = 208 & 13;`
	
	`  printf("%d\n", V1);`
	`}`
		`class Snippet {`
		`  public static void main(String[] args) {`
		`  int V1 = 11 & 32;`
		
		`  System.out.println(V1);`
		`  }`
		`}`

code example 52
 	
removed
	`void main() {`
	`  char V1 = 104;`
	`  printf("%c\n", V1);`
	`}`	

code example 59
 	
removed
	`void main() {`
	`  int V1 = -1;`
	
	`  unsigned int V2 = V1;`
	
	`  int V3;`
	`  if (V2 > 0) {`
	`  V3 = 4;`
	`  } else {`
	`  V3 = 5;`
	`  }`
	
	`  printf("%d\n", V3);`
	`}`	

code example 53
 	
adapted
	`void main() {`
	`  char V1 = 23;`
	
	`  printf("%d\n", V1);`
	`}`
		`class Snippet {`
		`  public static void main(String[] args) {`
		`  int V1 = Integer.parseInt("13", 8);`
		
		`  System.out.println(V1);`
		`  }`
		`}`

Table 3.2: Code examples where the translation is notably changing the code

Atom Name	Java Code Snippet with Atom of Confusion	Java Code Snippet Free of the Confusion
Infix Operator Precedence	`2 - 4 / 2`	`2 - (4 / 2)`
Post-Increment/Decrement	`V1 = V2++;`	`V1 = V2; V2 += 1;`
Pre-Increment/Decrement	`V1 = ++V2;`	`V2 += 1; V1 = V2;`
Constant Variables*	`V1 = V2;`	`V1 = 5;`

Remove Indentation atom	`while (V2 > 0)`
	`  V2--;`
	`  V1++`
		`while (V2 > 0)`
		`  V2--;`
		`V1++`

Conditional Operator	`V2 = V1 == 3 ? 2 : 1;`	`if (V1 == 3) { V2 = 2; }` `else { V2 = 1; }`
Arithmetic as Logic*	`(V1 - 3) * (V2 - 4) != 0`	`V1 != 3 && V2 != 4`

Logic as Control Flow	`V1 == ++V1> 0 || ++V2 > 0;`	`if ( !(V1 + 1 > 0) )`
		`  { V2 += 1;}`
		`V1 += 1`

Repurposed Variables	`for(int V1 = 0;...; V1++) {` `  for(int V2 = 0;...; V1++) {`	`for (int V1 = 0;...; V1++) {` `  for (int V2 = 0;...; V2++) {`
Dead, Unreachable, Repeated*	`V1 = 1; V1 = 2;`	`V1 = 2;`
Change of Literal Encoding	`V1 = 013`	`V1 = Integer.parseInt("13", 8)`
Omitted Curly Braces	`if (V1) F1(); F2();`	`if (V1) { F1(); } F2();`
Type Conversion	`V1 = (int) 1.99f;`	`V1 = (int) Math.floor(1.99f);`
Indentation	`if (V1 > 0) { }` `V2 = 4`	`if (V1 > 0) { }` `V2 = 4`

Table 3.3: Atoms of confusion; Summarized Java variants. Atom names marked with a * failed to meet statistical significance in the work of Gopstein et al. [21].

just guessing where mistakes are made. We decided on asking this question in the form of a Likert scale [25]; the participant is asked to which degree they agree/disagree with the statement "I am certain of the correctness of my answer above." The choice of Likert scale was made due to the possibilities of analysis. The ranked order of the options is useful to categorize opinions, and are familiar and quick to fill in for participants. Other considered options include: a yes-no question to specify if the previous provided answer was a guess and a slider to indicate confidence.

Before the participant will be directed to the real questions, we first show one example exercise. A predefined code example is given, with example answers already filled in for the questions. The goal of the examples is to show the participant what is expected and to give an example of how the open questions can be answered.

Similar to the study by Gopstein et al. [21], we incorporate strategies to cope with the possibility of a learning effect [32]. Since there are less questions in the survey than code examples, a randomized subset of the available snippets is constructed. The available code example pairs are shuffled and the new ordering is used to construct the participant specific question set. The randomized order distributes bias of question order over the participants. No specific actions are taken to prevent a single participant from seeing multiple variants of the same atom category, but it is guaranteed that every obfuscated-translated pair is only used for one question.

3.2.2. Part 2: Perception

The second part of the survey attempts to answer RQ2: How do students perceive confusion in Java programs that include atoms of confusion, as opposed to the translated, confusion-free, Java programs? Participants will be presented with both the variant that includes that atom of confusion, and the variant that is transformed to not include this confusion for 5 different pairs. Participants do not know what atom of confusion is presented to them, as well as which variant contains the atom of confusion and which does not. The first question asks which of the two variants the participant perceives as more confusing. The four options are:

- 1 is more confusing

- 2 is more confusing
- Both are equally confusing
- Neither are confusing

These four options provide the participant to add some granularity to their answer while still being easily categorizable. While the the third and fourth option do not distinguish the two variants, a difference in meaning is present in the options. Answering 'Both are equally confusing' would mean that both variants of this atom are not very readable code and can confuse people, while answering 'Neither are confusing' hints that both variants are not confusing or sufficient readable.

Next, the participant is asked to explain their answer. This will allow us to more precisely identify what is the reason of the confusion. The confusion might be unrelated to the purpose of the atom, and would indicate that (the representation of) this atom is not suitable.

3.2.3. End of the Survey

After the main parts are finished, some demographics are collected. The first question asks what year of study the participant is in. Additionally, the programming experience is asked. Since students generally add any programming experience to such a question, the answer on this has a lower level of importance than the year of study.

Lastly, the participant can optionally leave an email address to enroll for a possibility of winning a gift card. The email answer will never be used in analysis and only the winner will be contacted. The design of the survey was approved by the Human Research Ethics Committee of the TU Delft. The approval of the submission can be found in Appendix B.

The resulting design of the survey can be found in Appendix C, where screenshots of the pages are shown.

3.3. Participants

To emphasize measuring the impact on programming education, the target group of the study is selected to be beginner level programmers. Computer Science students from Delft University of Technology are targeted as participants. For first year students, the majority of the participants, course lab hours were used to reach out. During shared lab hours (where student assistants for multiple courses are available to help students) we reached out to each student present, briefly introduced them to the topic of the study and asked them to fill in the survey. The link to the survey was presented on big screens in lab rooms, so students were reminded to fill in the survey after finishing course work.

For second year students, we got permission to post one message, without channel notifications, on their group communication platform.

From the data collected in the survey, the distribution shown in Figure 3.2 shows the distribution over the study progress of 96 students that completed the survey. The majority of the participants is currently in the first year of the Bachelor's degree. The *'other'* category consists of two teachers and two students in the bridging program to the Computer Science Master's programme, the last participant in this group did not provide a valid answer. Additionally, 36 participants filled in at least one question but did not complete the experiment. Their results are used where possible, but since the question about the year of study is asked at the end of the survey, no data is available to display in Figure 3.2.

3.4. Data analysis

Since the order and selection of code examples is random, and different for every participant, the results needs to be processed first before the data can be analyzed. Answers need to be grouped by code example instead of survey question.

For RQ1, the answers participants gave on the question 'What do you think this code will print?' is compared to the correct answer for that code example. The correct code examples are generated by a script that compiles the final version of the Java code, runs the generated class file, and saves the console output to csv. Some answers by participants provide the correct answer, but due to differences in formatting, or typos does not match the correct answer. To ensure that these answers are still counted as correct, correction rules are applied to the given answer, before comparing it again to the correct answer. The first correction is to ignore differences in upper- and lowercase characters. Then, any additional spacing is removed, creative answers including in plain text `\n`, `+` or `*enter*`

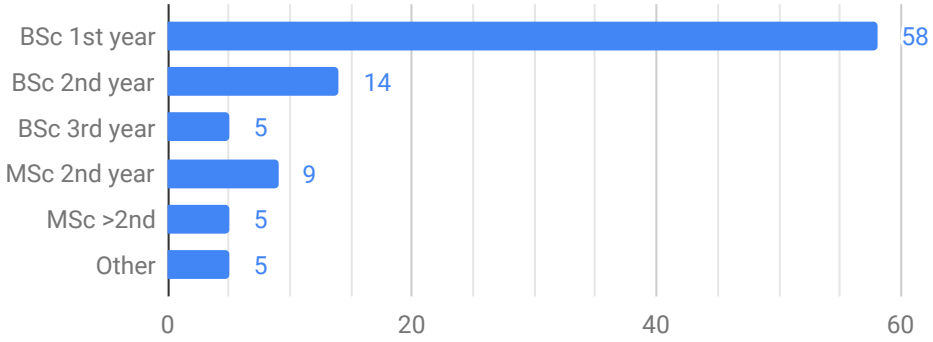

Figure 3.2: Amount of participants per year of study.

are cleaned from additional information. Additional correction rules include converting the answer to lowercase, and adding or removing `.0` in answers on atoms where the type is not part of the targeted confusion. The detailed implementation of these corrections can be observed in our online appendix [29].

The results will be used to compute the odds ratio of wrong answers being caused by the atoms of confusion. The corresponding confidence interval will show if the confusion caused by the atom is significant. The odds ratio, its standard error and 95% confidence interval are calculated according to Altman [2]. Where zeros cause problems with computation of the odds ratio or its standard error, 0.5 is added to all input values [15, 34].

The question regarding the certainty of the answer the participants gave is used to find out wether participants feel less certain in more confusing situations. Additionally, the situation when the participant indicates high certainty of their answer while they answered incorrectly can be used as an indicator that this atom is important to be avoided in real life. This case is especially relevant in the context of code reviews, as this implies that people will not question the behavior when seeing this type of code, while they should be alarmed.

For the second research question, the perception part of the survey is used. Answers are again collected per atom and are corrected for the order in which the two variants of the atom are presented to the participant. Furthermore, the answers per atom are grouped by the answer what atom the participant found more confusing. The results indicate to what extend the obfuscated atoms are perceived as being more confusing than their transformed counterpart. Now, the results can be categorized; the main interest is checking if the participant is confused by the atom or other reasons. Additionally, the answers from participants can give new insights on what makes the atom confusing. Lastly, results from the perception part can be compared to the comments left by confused participants on the *effect* part of the experiment. This will show if seeing both the obfuscated and transformed version of an atom changes the view on the code example.

When participants do not finish the survey, the answers they filled in so far are still stored. These responses will be marked as 'incomplete' but the given answers might still be useful. Most registered incomplete responses result from people opening the instruction page and leaving. The incomplete responses with some answers filled in are extracted and used in the described analysis.

3.5. Threats to Validity

Despite efforts to maximize validity, no research comes without potential threats to validity. In this section the potential issues in the design and execution of the experiment that could possibly have an effect on the results or conclusions are listed.

3.5.1. Threats to Internal Validity

The selected tasks were inspired by the question set used by Gopstein et al. [21]. In their work, the subjectiveness of making this set is mentioned as a threat to validity. This work takes a subset of the original. We chose not to add additional atoms, to allow comparison to the original work, and take away nonobjective contents of the set.

The only data collected related to programming experience is years of study. While this is not a precise measure, it does give guarantees on the minimal knowledge of our subjects. The curriculum of Computer Science and Engineering starts with a course on object oriented programming (OOP), where the used language is Java. After that, the students took part in the OOP project course, requiring the students to build an application in Java. While it is unsure if all participants passed the course, they have been in contact with the Java programming language for a minimum of half a year. This gives us a strong indication of the minimal amount of experience for every participant. The two main groups in the set of participants consists of 1st year bachelor students and 2nd year master students. This is quite a big gap in years of study, and also in experience. Mixing these groups can have an influence on the data. In the first experiment, we expect an influence on the data towards less significant results on the confusing effect. More experienced students might spot pitfalls and tricks better. For the second experiment, the textual explanations of more experienced students could be selected as highlights, but no significance is measured here. This will increase the generalizability of the results, as a set of mixed experienced participants is used.

Participation in the experiment is completely voluntary, potentially introducing a bias due to volunteers generally having more motivation. To counter this threat, no selection was made in the recruitment process of participants, as described in Section 3.3.

One instance of the atom *type conversion* contained a translation that is not equivalent in functionality to the obfuscated atom. In task 60, as seen in Appendix A, the integer 288 is casted to Java's `byte` type. The translation shows a cast to byte of $288 \% 256$, removing the loss of data, but not representing the functionality of a cast to byte, since a byte will take values between -128 and 127. The incorrect translation could be introducing more confusion in the perception part of the study, since both variants are displayed next to each other, but none of the participants mentioned this mistake. Furthermore, inspection of results shows no indication that this error made an impact on the results.

3.5.2. Threats to External Validity

Gopstein et al. [21] mentions that the source of the atoms comes from the IOCCC (the International Obfuscated C Code Contest), and that these code examples are not representative for normal code bases. However, follow up research from Gopstein et al. [23] and Medeiros et al. [30] show that the selected atoms of confusion do occur in open source projects.

All code examples used in the experiment use short, nondescriptive identifier names like `V1`, `V2`. Nondescriptive identifier names reduce comprehension of source code [37]. Due to the limited size of the code examples, it is almost impossible to use more descriptive variable names. Since the code examples are isolated, it is hard to describe a meaning of the individual variable names, and finding names that will not introduce additional bias between the different code examples will be even harder. However, this does not rule out the effect the identifier names have on the comprehensibility of the source code atoms.

The experiment is executed within an online survey tool. This is far from the normal development environment for the participants. No tools are allowed to be used to help understand or to try the code out, so usage of an IDE would already have an influence on the results.

Working with participants naturally brings some uncertainties. It is never guaranteed that participants take part in the experiment seriously, and follow all the set rules. To minimize possibility that participants make use of external tools to help find results for the code execution tasks, all code snippets are presented in the form of images. This makes it extremely hard to directly copy and paste the code into a tool. This is not a foolproof solution, since most examples are tiny by definition, lowering the effort of copying the task contents by hand, but lowers the possibility.

4

Results and Discussion

This chapter will report the statistical results of the process described in the previous chapter. The performed experiment resulted in 96 survey responses, of which 58 are first year students (60%), shown in Figure 3.2. The time these participants took to complete the experiment ranges from 4 to 372 minutes. With the median at 17.78 minutes, the majority of participants finished within 20 minutes. Figure 4.2 shows the distribution of the total time taken per participant. Notably, the total time taken for 9 of the participants is not shown in the figure to improve clarity of the plot. These 9 participants took longer than 100 minutes to finish due to interruption of the process.

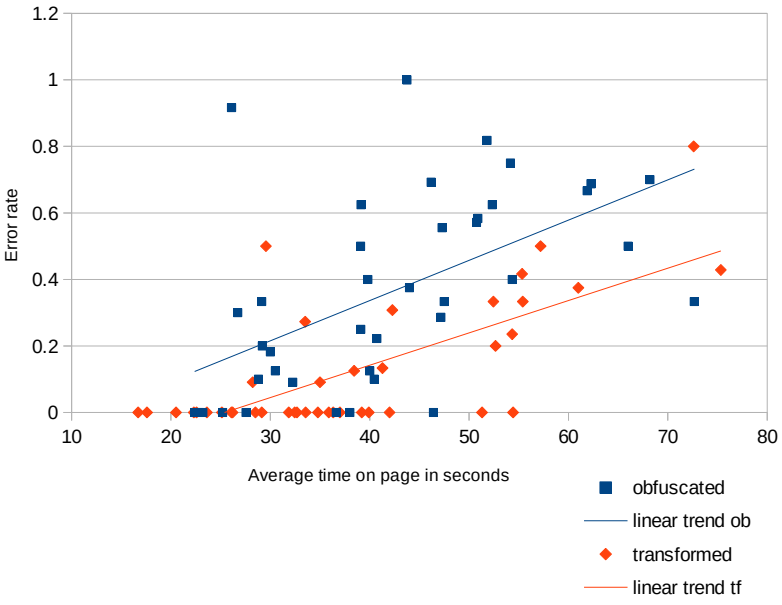

Figure 4.1: Error rate per task vs the average time spend on page for that task.

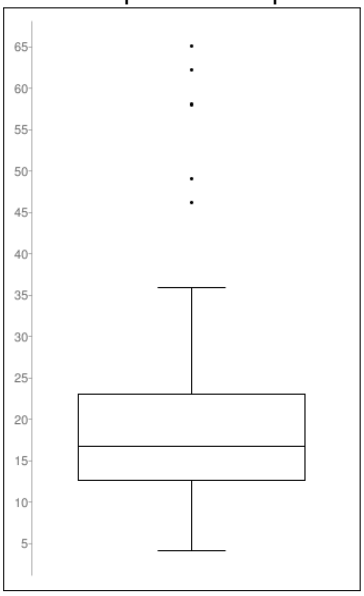

Figure 4.2: Distribution of duration of the experiment. *Outliers above 100 minutes left out (9x)*

The effect of the time spend per task on the rate of errors for that specific task can be seen in Figure 4.1. Every dot represents one task (see the list in Appendix A for all tasks). The colors divide the tasks in the obfuscated variant, that incudes the atom of confusion, and the transformed variant. The linear trend lines show that the error rate is increasing as participants spend a longer time on the page. This can be explained by that if the code example is clear an answer is quickly given, and, when the answer is not clear, a longer thought process is necessary to form an answer. The difference between the two trend lines, obfuscated above transformed, indicates that more errors are made in the tasks including an atom of confusion. A notable outlier is the blue square in the top left corner. This square represents the obfuscated variant of task 53 (Appendix A) and is part of the *change of literal encoding* atom. The y-position in the graph shows that this is a task with the second highest error

	Obfuscated		Transformed		Odds ratio	Confidence Interval	
Atom	**Correct**	**Wrong**	**Correct**	**Wrong**		**From**	**To**
infix operator precedence	14	4	31	1	8.86	0.91	86.63
post increment decrement	14	17	32	3	**12.95***	3.26	51.42
pre increment decrement	17	17	27	10	**2.70***	1.00	7.26
constant variables	30	0	27	0	0.90	0.02	47.00
remove indentation atom	16	17	26	0	**56.21***	3.16	998.97
conditional operator	23	6	27	0	15.21	0.81	284.53
arithmetic as logic	24	4	41	0	15.24	0.79	295.42
logic as control flow	5	22	20	8	**11.00***	3.09	39.21
repurposed variables	13	12	14	18	0.72	0.25	2.05
dead unreachable repeated	25	2	29	0	5.78	0.27	126.15
change of literal encoding	4	16	12	10	**4.80***	1.21	19.08
omitted curly braces	19	13	27	4	**4.62***	1.30	16.36
type conversion	10	13	18	1	**23.40***	2.66	206.16
indentation	31	2	24	0	3.89	0.18	84.78
Totals:	245	145	355	55	**3.82***	2.69	5.42

Table 4.1: Collected results of the first part of the survey. The number of correct and incorrect answers are combined for every atom of confusion, and based on this, the odds ratio is calculated. The 95% Confidence interval is presented in the last two columns. For the numbers marked as bold and with a * symbol, the confusion is statistically significant.

rates. The x-position shows a short average time spent on this task. The combination of these two factors may indicate that participants are very quick to draw an incorrect conclusion for this specific task. The other specifically interesting outlier is the blue square at the $y = 1$ line, indicating that all participants that have seen this task gave the wrong answer. This data point represents the obfuscated variant of task 41 (on the *logic as control flow* atom). We dive deeper into these observations in the next section.

4.1. RQ1: Which atoms of confusion hinder the comprehensibility of Java programs, and to what extent?

To answer this question, the first part of the described survey is used. Participants are given seven questions randomly selected from the code examples (in Appendix A). In Table 4.1 the collected number of correct and wrong answers are shown. For 7 out of the 14 tested atoms of confusion the confidence interval lies above 1, allowing us to draw the conclusion that the calculated odds ratio for this 7 atoms is statistically significant. For the other atoms, even though the odds ratio for every atom is greater than 1, the amount of responses is not large enough to draw significant conclusions.

The last row of Table 4.1 shows the collected amount of answers per column. In total, 315 questions with an atom of confusion in the question code were answered, of which 115 received a wrong answer, opposed to 46 wrong answers on 337 questions with the atom translated out. We will discuss the results of every individual atom of confusion in section 4.3.

Conclusion: *The totals row(Table 4.1) shows an odds ratio of* 3.82 *within a 95% confidence interval of* 2.69 *to* 5.42*, showing a clear result that in general, code with atoms of confusion is causing more understanding errors for Computer Science students from TU Delft.*

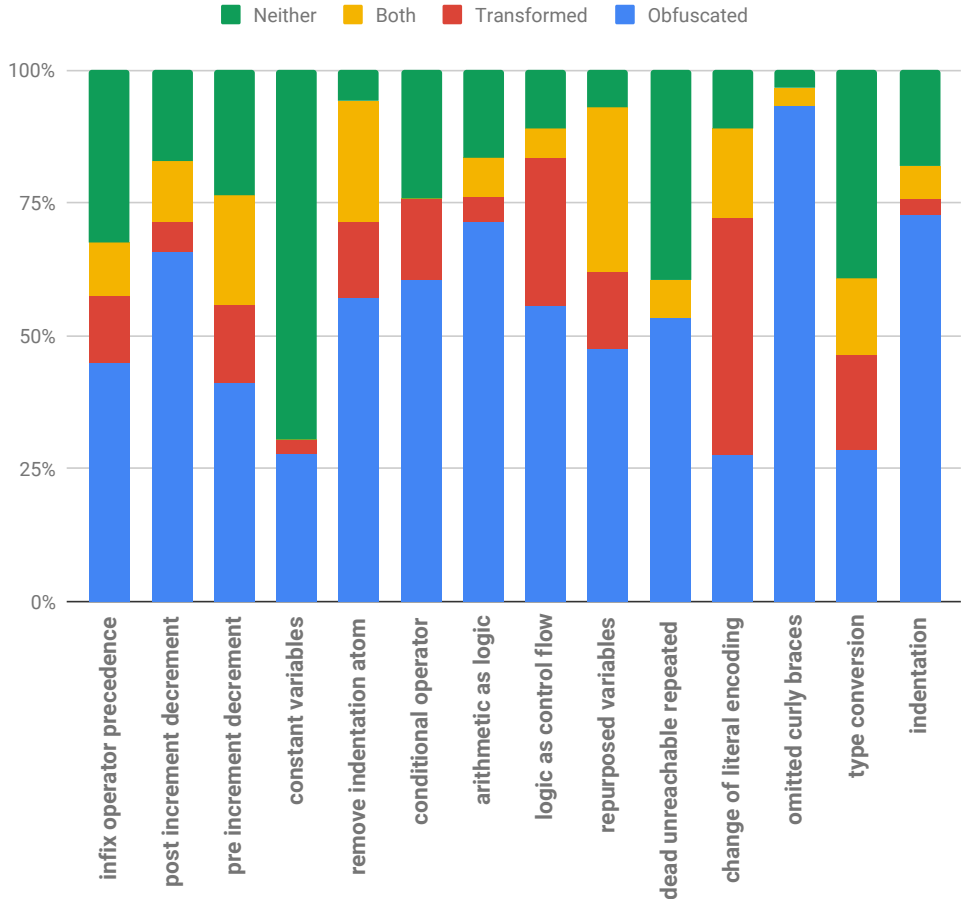

Figure 4.3: Survey answers on what atom variant is perceived more confusing per atom category.

4.2. RQ2: How do students perceive confusion in Java programs that include atoms of confusion, as opposed to the translated, confusion-free, Java programs?

To find out how developers think about the confusion atoms used in this experiment, we show the participant a pair of both the obfuscated and the transformed variant of a question, as explained in Chapter 3.2.2. The results of this part of the experiment are shown in Table 4.2. For every atom, we see the total answers per available option. The three rightmost columns show percentages of answers for obfuscated, transformed and the combination of the neither and both options. For 8 out of the 14 atoms, the majority of participants agreed that the obfuscated variant is more confusing. 10 of the 14 atoms have the obfuscated category as the highest percentage, but not necessarily above 50%. The remaining 4 atoms have a majority in the combination of Neither and Both except for the *change of literal encoding* atom, where the biggest group voted for the transformed variant.

In general, we can conclude that including atoms of confusion in code is perceived as confusing. The results show what the majority of developers see as readable, or understandable code. In the following section, we will discuss the results of every atom of confusion in detail. We will dive into the explanations and reasoning our participants indicated while making a decision between the two code example variants and draw conclusions per atom of confusion.

Conclusion: *In 54 percent of the cases, participants indicated the atoms of confusion to be more confusing. However, differences between individual atoms of confusion are present.*

	Aggregated results				Percentages per answer		
Atom	**Ob**	**Tf**	**Both**	**Neither**	**Ob**	**Tf**	**NB**
infix operator precedence	18	5	4	13	45.0	12.5	42.5
post increment decrement	23	2	4	6	65.7	5.7	28.6
pre increment decrement	14	5	7	8	41.2	14.7	44.1
constant variables	10	1	0	25	27.8	2.8	69.4
remove indentation atom	20	5	8	2	57.1	14.3	28.6
conditional operator	20	5	0	8	60.6	15.2	24.2
arithmetic as logic	30	2	3	7	71.4	4.8	23.8
logic as control flow	20	10	2	4	55.6	27.8	16.7
repurposed variables	20	6	13	3	47.6	14.3	38.1
dead unreachable repeated	23	0	3	17	53.5	0.0	46.5
change of literal encoding	5	8	3	2	27.8	44.4	27.8
omitted curly braces	28	0	1	1	93.3	0.0	6.7
type conversion	8	5	4	11	28.6	17.9	53.6
indentation	24	1	2	6	72.7	3.0	24.2
Total	263	55	54	113	54.2	11.3	34.4

Table 4.2: The amount of participants that indicated what variant of the question is perceived as more confusing. Ob = Obfuscated, Tf = Transformed, NB = Neither and Both combined

4.3. Discussion

In this section, the results and data of every atom will be discussed individually. The results from Tables 4.1 and 4.2 are used to draw conclusions along with individual responses of participants.The atoms of confusion will be inspected from different point of views with the help from the opinions of participants and the work from other research and researchers.

Infix Operator precedence

obfuscated

```
2 - 4 / 2
```

transformed

```
2 - (4 / 2)
```

The *infix operator precedence* atom is all about order of operations in single line statements. The results in Table 4.1 show that this atom is not significantly confusing. The targeted confusion is caused by assuming an incorrect order of execution when more than one operator is used in the same line of code. The transformed variant of the atom includes parenthesis around the operators to make the order of operations more clear. For RQ2, 45% of participants indicate the obfuscated variant to be more confusing. The explanations from this group describe that the additional parenthesis improve readability by making the order of operations more clear. One of the participants in the work of Medeiros et al. [30] (in the research on atoms of confusion in open-source projects) states: "[he] prefers to have parenthesis always, to [him] it makes it simpler to read". Another 42.5% (combining the 'Neither' and 'Both' answers) states there is no difference between the two. Arguments here state that the order of operations is clear, but knowledge of the precedence rules is needed. 12.5% of answers given indicate the code without the confusing pattern to be actually more confusing. All these answers came from one specific used code example where the precedence of the `!` operator was made more explicit by adding parenthesis. The translation changes the order the operations are listed and adds optional parenthesis to indicate the order of operations. In this example, opposed to the other two examples, the parenthesis are considered unnecessary and 'making it look cluttered' or harder to follow/read by the participants. Other similar research also show participants agreeing with this statement: "*Brackets are there to alter the normal operator precedence, and when seeing a bracket you should be able to assume that the operator precedence has been altered. Extra brackets make the code less readable and less understandable*" [30].

Post Increment Decrement

obfuscated

```
V1 = V2++;
```

transformed

```
V1 = V2;
V2 += 1;
```

The post increment operator increments the variable and returns the original value of this variable. Results for the first research question show that this atom is significantly confusing. The confusion can be caused by different misconceptions. First, the operator might not be recognized. Only two of the participants that had any question regarding the post increment atom indicated that they are not sure if the value of the variable will be changed or if the value of the original variable will be returned. This shows that this operator is familiar to the participants.

A second possible misunderstanding is confusing the postfix increment/ decrement operator with the prefix increment/ decrement operator. Instead of returning the original value, the pre increment or decrement operator returns the result of the expression. In the obfuscated variant of questions 8 and 9, the majority of explanations provided for wrong answers given by participants indicate that this is the cause of the answer being incorrect. Another reason for confusion is forgetting that the operator actually changes the variable. One given answer indicates: "*... Not sure if it will change the value ...*".

Given these causes of confusion, and a significant result in Table 4.1, with a lower bound of the confidence interval of 3.26, we can say that this atom is significant confusing.

Moreover, 65.7% of participants answer that the code examples with the atom of confusion are perceived as more confusing. In the explanations, most participants indicate they know they are unsure about this atom. A participant indicates "*I always forget were V1++ evaluates to (old or new value).*" clearly describing what makes this participant confused. Another participant answers "*Pretty sure that having the "++" after the variable makes sure it is evaluated before it is incremented, but I am still not sure and would probably have to just run the code to find out*" making it even more clear that he/she knows where the confusion is and knows how to find the behavior in the current situation.

With this result, and the ways that this operator can be misinterpreted, it is best to avoid this pattern. Even though many people recognize the situation, it would save a lot of thinking time writing a few extra characters to avoid this pattern.

A potential exception to this might be the use of this operator within the setup of an iterative for-loop. The standard way of defining a for-loop in Java is `for(int i = 0; i < X; i++)`. Since this line is so often used, it is likely that people know what the resulting behavior will be, without thinking about the steps that happen in between. To confirm this potential exception, future research is needed.

Pre Increment Decrement

obfuscated

```
V1 = ++V2;
```

transformed

```
V2++;
V1 = V2;
```

This atom is very similar to the post increment decrement atom. The difference, as explained before, is that instead of the original value of the variable, the result of the expression is returned.

This is caused by the larger amount of wrong answers on the transformed variants of this atom. In contrast to the previous atom, participants indicate they do not know what `--V1` would do. Five participants state they are unsure about or are unfamiliar with the syntax. The transformed variant of this atom does not remove the operator, but instead isolates it. This removes the confusion with the behavior of prefix operator, but does not resolve confusion caused by syntax. One notable observation by participants is that the transformed variant of question 10 initializes two variables in the same line: `int V1 = 5, V2;`. This causes unnecessary confusion since some participants indicated this as the reason of confusion.

The first experiment shows that this atom is confusing. The confidence interval starts ever so slightly above 1. The incorrect answers include "*Idk if ++V1 is a thing. I always use V1++*" and "*I have no clue what "–v1" means*". This shows that these people are confused by the atom.

The second experiment confirms this finding. The participant answering "*I can never remember what –V1 really does so to me this is confusing*" clearly indicates that this atom is causing confusion.

41.2% of the participants agree with the person above that the code variants with the atom of confusion included are more difficult to understand.

For future work, the double variable initialization in one line should be removed, and, the transformation of both this atom and the post increment atom should avoid the pre and postfix operator and should instead write out what behavior this operator represents.

Constant Variables

obfuscated

```
V1 = V2;
```

transformed

```
V1 = 5;
```

The constant variable atom was not shown to be statistically significant in the research by Gopstein et al. [21]. This research confirms this result. None of the participants answered a question wrong that contained this atom. The only slight confusion indicated is related to question 15 where it is unclear if the printed text will include the `.0` caused by the *double* type of the variable. Answers not including the `.0` in their answer but giving the correct numerical answer are considered correct since this is not part of the confusion under study. 69.4% of the answers in the second part of the survey indicate no difference in confusion between the variants of the questions for this atom. The majority of explanations state that the code is very simple to understand. The main reason why the obfuscated atom would be considered more confusing is caused by the unnecessary extra code. The added verbosity contributes to complexity in the used code examples. This particular pattern does not fit the atoms of confusion since, when isolated does not cause confusion. It will be interesting to research this pattern when present in larger code examples.

Remove Indentation Atom

obfuscated

```
while(V2 > 0)
   V2--;
   V1++;
```

transformed

```
while(V2 > 0)
   V2--;
V1++;
```

Along with the atom *indentation*, these two atoms are included in this research despite being removed in the original experiment by Gopstein et al. [21]. In their errata, they state that: "*To remove the bias introduced by code formatting, we chose not to study the effect of whitespace in this study*". This study did include these two atoms to explore the impact of misformatted code. The results show that this atom, remove indentation, is significant confusing. The lack of indentation makes it difficult to see where scopes are ending, resulting in a majority of wrong answers for the obfuscated variants. All participants that encountered a transformed variant of this atom in the first part of the survey gave the correct answer.

The perception of both variants of the questions is that the missing a are a big contribution to the confusion. The main argument of the participants that find both variants equally confusing is related to missing brackets. In the *indentation* atom, the brackets are included, and only the indentation itself is wrong or missing, that atom is not significant confusing, with only two wrong answers for the obfuscated variants.

One thing to note, question 16 is an example of the 'dangling else' pattern as researched by Medeiros et al. [30]. They state that many coding standards enforce the use of brackets to avoid this pattern. Other authors also show the confusion of nested *if/else* statements with misleading indentation [30, 39]. Our results cohere with this conclusion, showing major confusion with missing indentation when brackets are left out and no confusion when the brackets are included.

A way to avoid misleading indentation is by the use of code formatting tools. "Most formatting obfuscation transformations can be trivially undone by a source code beautifier" [4]. Regular or enforced use of this kind of tools keeps the formatting of source code consistent.

Conditional (Ternary) Operator

obfuscated

```
V2 = V1 == 3 ? 2 : 1;
```

transformed

```
if (V1 == 3) {
    V2 = 2;
} else {
    V2 = 1;
}
```

The conditional operator atom is all about the ternary operator. In Java, the only available ternary operator is the `? :` operator, which provides a shorthand way of writing an `if-then-else` statement.[1] Although 20% of the answers for the obfuscated variants of this atom were wrong while only correct answers were given to the transformed variants, not enough data is present to draw significant conclusions (Confidence Interval 95% between 0.81 and 284.53, Table 4.1). The participants that were incorrect and/or were less sure on their answers state that they are unsure about the syntax. The notes left by participants that answered correct show they are unsure about the functioning of the operator but have a correct intuition. Explanations regarding the second research question indicate that a major part of confusion for the obfuscated variants comes from the lack of brackets around the condition. In order to take away this particular confusion, the provided example above would change to `V2 = (V1 == 3) ? 2 : 1;`.

A big amount of participants indicate that the ternary operator is unclear. Others state that it can be confusing to people unfamiliar with the operator. The main reason participants picked an option other than 'obfuscated' in the second part is due to the added verbosity of the regular if-else statements. Having to read or write 4 or more lines of code instead of 1 makes the understanding take longer when familiar with both syntax variants. This conclusion falls in line with what Kernighan and Pike [28] write, stating that using the ternary operator to replace four lines of if-else code is fine. In a similar studies to this, the decision was made not to include this atom in the experiment, due to the large amount of usage of this atom in practice [30].

Arithmetic as Logic

obfuscated

```
(V1 - 3) * (V2 - 4) > 0
```

transformed

```
V1 != 3 && V2 != 4
```

This atom is one of the four atoms in the work of Gopstein et al. [21] that did not show to be confusing. The results of this research shows the same conclusion for the Java programming language. The main assumption from Gopstein et al. [21] is that using arithmetic operators instead of logical operations, will imply a non-boolean range, which might be confusing. Due to the translation to Java, the resulting number has to be explicitly compared to 0 to create a boolean value, taking away this implication of a non-boolean range. The results of part 2 do show that the obfuscated variant is much less preferred to read. The additional calculations that are unusual lead to a much longer time to see the intention of the code, according to the participants. Some complaints do come in about the order of comparison in the translated variants of this atom. Reversing the order of `variable <comparison operator> value` is disliked by one of the participants. To isolate the atom more, this order can be kept constant by translating these conditions to a consistent order. The effect of the order reversal is interesting for future research.

[1] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html

Logic as Control Flow

obfuscated

```
int V1 = 1;
int V2 = 5;

if (++V1 > 0 || ++V2 > 0) {
   V1 = V1 * 2;
   V2 = V2 * 2;
}
```

transformed

```
int V1 = 2;
int V2 = 4;

if (++V1 > 0) {
   V1 = V1 * 2;
   V2 = V2 * 2;
} else if (++V2 > 0) {
   V1 = V1 * 2;
   V2 = V2 * 2;
}
```

Due to lazy evaluation of the || and `&&` logical operators, they can also be used as conditional operators. This means that, depending on the value of the left side, the right sight may or may note be executed. To measure the possible confusion of the control flow, the tasks for this atom make use of the pre or post increment operators. Similar to the previous atom (*Arithmetic as Logic*) the translation to Java added `> 0` in order to use numbers for boolean values. The most notable single result for this atom in part 1 of the survey is that the obfuscated variant of task 41, that is used as the above example, was answered incorrect by every participant. In total, the obfuscated variants for this atom have an error rate of 81%. This plays a role in this atom having a 3.09 lower bound of the confidence interval, showing that this atom has significant confusing effect.

A slight majority of the participants agreed that the code examples for this atom were more confusing when the confusing atom was present. Differences between the code examples are however present, therefore, we will dive deeper into the individual code examples.

For the example shown above, the participants did not agree on one answer. Participants are spread across all the answers. The arguments against the atom say the extended `if-else` flow is easier to follow. The participants that prefer obfuscated code example indicate that a single condition is easier to check and therefore faster to process, and the same code in both the if, and the else body is bad practice. Both are perceived confusing due to the use of `++V`.

obfuscated

```
int V1 = 1;
int V2 = 5;

boolean _test = V1 == V2 &&
    ++V1 > 0 || ++V2 > 0;
```

transformed

```
if (V1 == V2) {
   ++V1;
} else {
   ++V2;
}
```

For this example, all participants agreed the obfuscated variant, with the atom of confusion, is more confusing. Some explanations are: "*Using logic expressions to perform effectfull computation is cool [..], but more confusing because it breaks the vertical flow of effectfull expressions/statements.*" and "*++variables should be replaced with something more clear*".

obfuscated

```
int V1 = 3;
int V2 = 5;
int V3 = 0;

while (V3 < V2 && ++V1 > 0) {
    V3++;
}
```

transformed

```
int V1 = 1;
int V2 = 11;
int V3 = 0;

while (V1 != V2) {
    ++V1;
    if (!(V1 > 0))
        break;

    V3++;
}
```

For the last example, the participants are more spread out again. 10 participants indicate the obfuscated variant to be more confusing, 7 selected the transformed variant, 1 participant found both confusing and 3 selected the 'neither' option. Some takes from participants are:

- "*The logic is easier to follow when its spread out*"
- "*The inline condition looks better than a if condition within a while loop*"
- "*I really dislike nesting whiles and ifs as done in [transformed variant]. You cannot easily trace it back yourself. [Obfuscated variant] is not perfectly readable, but it only has one check which is easier to do by hand.*"

Using either pre- or post-increment within this atom is hard to avoid, but since the presence of these atoms within these code snippets effects the confusion, the *logic as control flow* pattern is not suitable for the set of atoms of confusion. While this pattern is clearly confusing, as shown by the existence experiment, it violates the *indivisible* part of the definition by Castor [12].

Repurposed Variables

obfuscated

```
for(int V1 = 0; ...; V1++) {
    for(int V2 = 0; ...; V1++) {
```

transformed

```
for(int V1 = 0; ...; V1++) {
    for(int V2 = 0; ...; V2++) {
```

The *repurposed variables* atom 'misuses' an already existing variable for another purpose. In the above example, both for-loops make use of the V1 variable to keep track of their state. This is different from what can be expected of the usage of a for-loop, and can potentially throw the reader off guard.

An exceptional result for this atom is that the variants where code examples did not include the atom of confusion are answered incorrectly more than the variants with the atom of confusion. For the variants with the atom of confusion, 13 participants answered correctly and 12 gave an incorrect answer. The variants without atom of confusion resulted in 14 correct and 18 incorrect answers.

The results of the perception part for this atom fall more in line with the expected results. 47.6% of participants agreed that the obfuscated variant is the more confusing one. A substantial amount of participants indicated that they found both variants to be confusing (38.1%). This is more in line with the findings in the first part of the experiment.

One thing that was overlooked in the translation of these snippets from C to Java is the instantiation of arrays. In C, this is often done as `int V1[] = …` while in Java, the square brackets are often at the type level, like `int[] V1 = …`. While in Java both variants provide the same result, the variant with the array indication on the type is taught and used much more frequently. This might have caused additional confusion for participants, but exists in both variants for this atom, so not influences the difference between the two.

Dead, Unreachable, Repeated

obfuscated

```
V1 = 1;
V1 = 2;
```

transformed

```
V1 = 2;
```

Dead, Unreachable, Repeated is again one of the atoms that will not cause significant confusion according to Gopstein et al. [21]. The shared factor in the 3 terms of this atom name is that the removal of the corresponding line(s) of code will not change the behavior of the program. Only 2 wrong answers were given out of 56 times this atom was present in a part 1 question. This atom is difficult to test since the isolation makes it easy to spot what is happening. In the study on finding atoms of confusion in real world code bases, Gopstein et al. [23] did not search for this atom. Future work could look for instances of this pattern in exiting code and check if occurrences are related to bugs. In the case of dead code, most (if not all) well known IDEs and static analyzers for Java will warn the user if dead code is detected.

Change of Literal Encoding

obfuscated

```
V1 = 013;
```

transformed

```
V1 = Integer.parseInt("13",8);
```

This atom focuses on the general case when the encoding of the characters written down will change in the meaning of the program. In the example above, prepending a number with a 0 tells the computer to parse this number in the octal numbering system, meaning 013 will result in the decimal value of 11. The specific task corresponding to this example is the most notable outlier in Figure 4.1. This indicates that this task is very often made wrong, and participants quickly jump to a wrong conclusion.

The other questions from Gopstein et al. [21] include assigning a numeric value to a variable with the char type and using a bitwise AND operator on decimal numbers. The question converting a number to a character is not included in this research since, in the situation when a participant is not confused about the atom, it is still really hard to give the correct answer because it would require exact knowledge of the corresponding character to that number. This atom is significant confusing, resulting from the amount of wrong answers in the first part of the research in Table 4.1. The obfuscated variant is answered four times more wrong than right. In the second part of the experiment is this the only atom where the transformed variant is most frequently marked as most confusing.

Omitted Curly Braces

obfuscated

```
if(V1) F1(); F2();
```

transformed

```
if(V1) {
    F1();
}
F2();
```

This atom is similar to the *Remove Indentation Atom* and the *Indentation* atom in the sense that the targeted confusion is related to unclear separation between code blocks. This atom removes the curly braces that normally follow an if statement, while loop, etc. to determine the start and end of the scope. One participant identifies the confusion as: *"Putting several statements on one line is unnecessarily confusing."* This atom is significant confusing with a lower bound of the confidence interval of 1.30. 41% of participants that were asked to evaluate a task with this atom of confusion in the code answered incorrectly. The confusion for these tasks is indicated by one of our participants as: *"I thought Java only executes the next statement after a for-loop without brackets but it also could be that the whole next line is executed."* For this atom of confusion, a stunning 93% of participants agree that it is more confusing to omit the curly braces from these code blocks.

Type Conversion

obfuscated

```
V1 = (int) 1.99f;
```

transformed

```
V1 = (int) Math.floor(1.99f);
```

The *Type Conversion* atom is all about converting one type into another. Next to the example above, the other included code variant is casting the integer 288 to byte. In both cases, the cast executes an irreversible action since we remove details and lose precision. This atom showed the most uncertainty as indicated by our participants. The task regarding casting to the byte primitive type is especially often answered wrong.

One of the examples of Gopstein et al. [21] included transforming a negative number to an unsigned integer. This example was not easily transformed into Java. Casting from int to char will provide similar behavior, but removing this atom is not possible in a simple enough manner, since the confusing behavior happens at large values (above 2^{16}, since Java uses UTF-16 character encoding). `char` is the only primitive type in java that is encoded unsigned, and would also have introduced even more confusion since casting characters to numbers is non-intuitive.

The results for this atom show that this pattern causes confusion. The results show that the transformed variants cause more correct answers. While the confusion is significant, participants are spread over which variant is more confusing. The biggest group answered that neither of the code variants is confusing. An argument for this is: "*I believe that casting to an integer automatically floors the value, while it might be easier to understand by flooring as well, I don't think adding redundant code makes it less confusing. While the second simply casts to an integer, flooring it as well.*" A possible influence on the spread of answers could be that the transformed code variants explain the confusing code, making it easier to comprehend. More research is needed to draw any specific conclusions on this.

28.6% did indicate that the code examples including an atom of confusion are indeed harder to understand. The argument given here is that the transformed code is much more explicit in what is happening. "*You would have to know what happens if a float is casted to an int*" one participant mentions. Explaining with the code what you want to do is for these participants helpful for understanding.

Indentation

obfuscated

```
if (V1 > 0) { }
   V2 = 4;
```

transformed

```
if (V1 > 0) { }
V2 = 4;
```

As mentioned in the discussion of the *Remove Indentation Atom*, this is one of the atoms that is not tested in the experiment of Gopstein et al. [21]. Participants that were shown code examples related to this atom of confusion only answered incorrectly 2 times. Both incorrect answers were given to code examples with the confusing pattern present, but since a total of 57 code examples related to this atom were shown, this atom does not meet significance.

As for the perception of the participants towards this atom, the majority agrees that the variants with the atom present are more confusing. Furthermore, the explanations heavily gear to the intended explanation of this atom. The comments often mention indentation or formatting, and how this affects the clearness of the distinction between different branches.

These results indicate that this atom of confusion is easily recognized, and therefore will not often cause actual misunderstanding. The perception however, shows a strong preference to avoid this atom. It is not recommended to write code this way, since it unnecessarily complicates the code. One participant provides a nice reason for this that "*if you're skimming the code you could misread and think the [line of code] was in the if statement*". Code formatting tools are a very easy way to avoid the confusion targeted in this pattern, and make the code nicer to read, by fixing the indentation to clearly represent the code block depth.

4.4. Recommendations to Educators

Educators can play a big role in helping students detect confusing situations. When programmers encounter confusing patterns like the atoms of confusion, they should be made aware that this is a potentially problematic pattern. Whenever you spot such a pattern, you can take actions to refactor the code and clean up the pattern. This clears up possible misunderstandings and makes the code easier to read, understand, and maintain. To make this possible, a setup similar to the second experiment from this study can be used. Making the comparison between two snippets, with and without the atom of confusion, and explaining where the confusion originates from will help students understand where the misunderstandings originate from. The transformed variants provide a great example of how the code can be less confusing and easier to read and understand.

4.5. Avoiding atoms of confusion

For avoiding some of the patterns, IDEs are very suitable. Most code editors have an option to auto format code. This will, for one, clear up all atoms of confusion related to indentation. The *Remove Indentation Atom*, and *Indentation* are avoided by using this. For the following section, the warnings that the IntelliJ idea will give are inspected. Code following the *Dead, Unreachable, Repeated* pattern is very likely to be flagged as a warning. When the body of an if-statement is unreachable, IntelliJ will warn that `Condition '0 > 2' is always 'false'`. When assigning a variable to itself, warnings `Variable is already assigned to this value` and `Variable 'V1' is assigned to itself` are shown. Writing repeated assignment to the same value will result you the error `The value assigned to 'V1' is never used`. The IDE will also not like some of the *Change of Literal Encoding* atoms. When you assign an int with a prepending 0, you will see a `Octal integer '013'` warning.

The *Omitted Curly Braces* atom takes somewhat more effort to be caught. By default, the auto-formatter (in IntelliJ) will not fully clear this up for you. Settings exist to let the editor enforce using {}-brackets. Whenever they are not yet used, the tool will insert them for you.

An `.editorconfig` file for IntelliJ containing the following lines will always insert braces around the body of for, if, while, and do-while blocks:

```
[*.java]
ij_java_for_brace_force = always
ij_java_if_brace_force = always
ij_java_do_while_brace_force = always
ij_java_while_brace_force = always
```

Discussion Summary

Atom	Significant Confusing Effect	Perceived Confusing
infix operator precedence	👎	👍
post increment decrement	👍	👍
pre increment decrement	👍	~
constant variables	👎	~
remove indentation atom	👍	👍
conditional operator	👎	👍
arithmetic as logic	👎	👍
logic as control flow	👍	👍
repurposed variables	👎	👍
dead unreachable repeated	👎	👍
change of literal encoding	👍	👎
omitted curly braces	👍	👍
type conversion	👍	~
indentation	👎	👍

Table 4.3: Summarizing per atom of confusion. The *confusing effect* column shows whether or not the presence of the atom causes significantly more errors. The *perceived confusing* column shows a thumbs up when the obfuscated variant was perceived more confusing, a tilde when there is no difference in confusion between the variants, and a thumbs down when the transformed variant is more confusing than the obfuscated variant.

Now that we discussed the results for the individual atoms of confusion, we take a look at the takeaways. Table 4.3 shows a summary of the discussion above. For most of the atoms, a majority of participants indicate that they perceive the code examples with the atoms of confusion in them as more confusing. They compared them to equivalent code, but without those patterns. For the atoms of confusion where this is the case, this shows that the variants without the atoms of confusion are perceived less confusing. In these cases, avoiding the atom of confusing and using the provided workaround will help the understanding of the code.

5

Conclusions and Future Work

This thesis researched the effects of atoms of confusion by measuring the reactions and impact of these confusing patterns in Java code among first year students. We did this with the use This is done by first creating a suitable translation in Java for the code examples of the 19 *atoms of confusion* found by Gopstein et al. [21]. With these translated code examples, a two-fold experiment is conducted to find (1:) the effect of these atoms of confusion and (2:) the perception of the participants towards these atoms of confusion.

The results of the experiment are analyzed and discussed in order to answer the following research questions.

1. Which atoms of confusion hinder the comprehensibility of Java programs, and to what extent?
2. How do students perceive confusion in Java programs that include atoms of confusion, as opposed to the translated, confusion-free, Java programs?

To answer these questions, we first compute the odds ratio and associated confidence interval for every atom of confusion. Resulting values provide us with the information that 7 out of the 14 researched atoms of confusion have a confusing effect. To provide more detail, for each of the atoms, we dove into the results and discussed the specifics for the individual atom of confusion.

To answer the second research question, the results of the second part of the experiment are used mainly. For most of the atoms of confusion, more than half of participants agree that including the atom of confusion in a code example makes it more confusing. Three of the atoms of confusion have no consensus between participants' answers, and the *change of literal encoding* is the only atom of confusion where the atom makes the code example less confusing. Table 4.3 shows a summary of the results per researched atom of confusion.

5.1. Future Work

Initially, the plan was to run the experiment online, and to include the C, Java and Python programming languages in the experiment. Due to concerns on the amount of respondents per language, the choice was made to focus solely on Java, and use first year computer science students of TU Delft as the target group. The same experiment can be redone, asking participants for their programming experience in the 3 presented languages. This will provide additional relations to be explored between atoms of confusion and programmer experience.

In the process of this research, whenever anything related to atoms of confusion came by, an option to extend the set of atoms of confusion showed up. Due to the scope of this research, and the added value of being able to compare results to the work of Gopstein et al. [21], the set of atoms of confusion was not increased. However, an expansion of the set of atoms of confusion provides a nice topic for future research. Some potential atoms of confusion are:

- Usage of `break;` in nested loops makes it confusing what code will be executed after the `break;` command.

- The `^` sign can be easily mistaken to be a mathematical power, like `2^8` $= 2^8 = 256$. In a lot of programming languages, however, the `^` represents the logical `XOR` operator. `2^8` will then result in 10 instead of 256. This particular pattern raised a discussion on the gcc forums whether this should raise a warning: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90885

Bibliography

[1] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. Syntax, Predicates, Idioms - What Really Affects Code Complexity? In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, volume 24, pages 66–76. IEEE, may 2017. ISBN 978-1-5386-0535-6. doi: 10.1109/ICPC.2017.39. URL https://doi.org/10.1007/s10664-018-9628-3.

[2] Douglas G Altman. *Practical Statistics for Medical Research.* CRC press, 1991. ISBN 978-0412276309.

[3] John R Anderson. *Cognitive psychology and its implications*. Macmillan, 2005.

[4] Eran Avidan and Dror G. Feitelson. Effects of Variable Names on Comprehension: An Empirical Study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 55–65. IEEE, may 2017. ISBN 978-1-5386-0535-6. doi: 10.1109/ICPC.2017.27. URL https://doi.org/10.1109/ICPC.2017.27.

[5] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, may 2013. ISBN 978-1-4673-3076-3. doi: 10.1109/ICSE.2013.6606617. URL https://doi.org/10.1109/ICSE.2013.6606617.

[6] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications - ETRA '06*, page 125, New York, New York, USA, 2006. ACM Press. ISBN 1595933050. doi: 10.1145/1117309.1117356. URL http://portal.acm.org/citation.cfm?doid=1117309.1117356https://doi.org/10.1145/1117309.1117356.

[7] Moritz Beller, Andy Zaidman, and Andrey Karpov. The Last Line Effect. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 240–243, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2820282.2820317.

[8] Moritz Beller, Andy Zaidman, Andrey Karpov, and Rolf A. Zwaan. The Last Line Effect Explained. *Empirical Software Engineering*, 22(3):1508–1536, dec 2016. ISSN 15737616. doi: 10.1007/s10664-016-9489-6. URL https://doi.org/10.1007/s10664-016-9489-6.

[9] Jeffrey Bonar and Elliot Soloway. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human/Computer Interaction*, 1(2):133–161, jun 1985. doi: 10.1207/s15327051hci0102_3. URL https://doi.org/10.1207/s15327051hci0102{_}3.

[10] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, jun 1983. ISSN 00207373. doi: 10.1016/S0020-7373(83)80031-5. URL https://doi.org/10.1016/S0020-7373(83)80031-5.

[11] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. *Studying the Difference Between Natural and Programming Language Corpora*. Empirical Software Engineering, 2019. ISBN 1066401896. doi: 10.1007/s10664-018-9669-7.

[12] Fernando Castor. Identifying Confusing Code in Swift Programs. In *VI CBSoft Workshop on Visualization, Evolution, and Maintenance*, São Carlos, Brazil, sep 2018. URL https://docs.google.com/a/cin.ufpe.br/viewer?a=v{&}pid=sites{&}srcid=Y2luLnVmcGUuYnJ8Y2FzdG9yfGd4OjY2YTBhNWFjZWYxNmRiMTA.

[13] Jason Cohen. *Best Kept Secrets of Peer Code Review: Modern Approach. Practical Advice.* 2006. ISBN 9781599160672.

[14] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. 1997.

[15] Jonathan J Deeks and Julian PT Higgins. Statistical algorithms in Review Manager 5. *Statistical Methods Group of The Cochrane Collaboration*, 1(August):1–11, 2010. doi: 10.1371/journal.pone.0069930. URL https://doi.org/10.1371/journal.pone.0069930.

[16] J.J. Dolado, M. Harman, M.C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, jul 2003. ISSN 0098-5589. doi: 10.1109/TSE.2003.1214329. URL https://doi.org/10.1109/TSE.2003.1214329.

[17] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. Impacts of Coding Practices on Readability. *Proceedings of the 26th Conference on Program Comprehension - {ICPC} {\textquotesingle}18*, pages 277–285, 2018. doi: 10.1145/3196321.3196342. URL https://doi.org/10.1145/3196321.3196342.

[18] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion Detection in Code Reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2017. doi: 10.1109/icsme.2017.40. URL https://doi.org/10.1109/icsme.2017.40.

[19] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *ACM SIGPLAN Notices*, 48(4 SUPPL.):22–33, 2013. ISSN 15232867. doi: 10.1145/2502508.2502520.

[20] R. L. Glass. Frequently Forgotten Fundamental Facts about Software Engineering. *IEEE Software*, 18(3):0–2, 2001. ISSN 07407459. doi: 10.1109/MS.2001.922739. URL https://doi.org/10.1109/MS.2001.922739.

[21] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, 2017. doi: 10.1145/3106237.3106264. URL https://doi.org/10.1145/3106237.3106264.

[22] Dan Gopstein, Jake Iannacone, Yu Yan, Lois Anne Delong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Existence Experiment Questions Dataset, 2017. URL https://atomsofconfusion.com/2016-snippet-study/questions.

[23] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of Confusing Code in Software Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, pages 281–291, New York, New York, USA, 2018. ACM Press. ISBN 9781450357166. doi: 10.1145/3196398.3196432. URL https://doi.org/10.1145/3196398.3196432.

[24] Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding Object Misconceptions. *{ACM} {SIGCSE} Bulletin*, 29(1):131–134, 1997. doi: 10.1145/268085.268132. URL https://doi.org/10.1145/268085.268132.

[25] Susan Jamieson. Likert Scales: how to (ab)use Them. *Medical Education*, 38(12):1217–1218, dec 2004. ISSN 0308-0110. doi: 10.1111/j.1365-2929.2004.02012.x. URL https://doi.org/10.1111/j.1365-2929.2004.02012.x.

[26] Ahmad Jbara and Dror G. Feitelson. On the Effect of Code Regularity on Comprehension. *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, pages 189–200, 2014. doi: 10.1145/2597008.2597140. URL https://doi.org/10.1145/2597008.2597140.

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't Software Developers use Static Analysis Tools to Find Bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, may 2013. ISBN 978-1-4673-3076-3. doi: 10.1109/ICSE.2013.6606613. URL https://doi.org/10.1109/ICSE.2013.6606613.

[28] Brian W Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley Professional, 1999.

[29] Chris Langhout. Dataset for: Investigating the Perception and Effects of Misunderstandings in Java Code, 2020. URL http://doi.org/10.5281/zenodo.3822523.

[30] Flávio Medeiros, Gabriel Lima, Guilherme Amaral, Sven Apel, Christian Kästner, Márcio Ribeiro, and Rohit Gheyi. An Investigation of Misunderstanding Code Patterns in {C} Open-Source Software Projects. *Empirical Software Engineering*, nov 2018. ISSN 15737616. doi: 10.1007/s10664-018-9666-x. URL https://doi.org/10.1007/s10664-018-9666-x.

[31] Sanjay Misra, Adewole Adewumi, Luis Fernandez-Sanz, and Robertas Damasevicius. A Suite of Object Oriented Cognitive Complexity Metrics. *IEEE Access*, 6:8782–8796, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2791344. URL https://doi.org/10.1109/ACCESS.2018.2791344.

[32] James H Neely. Semantic Priming Effects in Visual Word Recognition: A Selective Review of Current Findings and Theories. In *Basic Processes in Reading*, pages 264–336. 1991. ISBN 9780203052242. URL https://www.taylorfrancis.com/books/e/9780203052242/chapters/10.4324/9780203052242-12.

[33] Wilberto Z. Nunez, Victor J. Marin, and Carlos R. Rivero. ARCC: Assistant for Repetitive Code Comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 999–1003, New York, New York, USA, 2017. ACM Press. ISBN 9781450351058. doi: 10.1145/3106237.3122824. URL https://doi.org/10.1145/3106237.3122824.

[34] Marcello Pagano and Kimberlee Gauvreau. *Principles of biostatistics*. CA: Brooks/Cole, 2nd ed. be edition, 2000. ISBN 9781138593145.

[35] John F. Pane, Chotirat Ann Ratanamahatana, and Brad A. Myers. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human Computer Studies*, 54(2):237–264, feb 2001. ISSN 10715819. doi: 10.1006/ijhc.2000.0410. URL https://linkinghub.elsevier.com/retrieve/pii/S1071581900904105.

[36] Shari Pfleeger. *Software engineering : theory and practice*. Prentice Hall, Upper Saddle River N.J, 2010. ISBN 0136061699.

[37] Andrea Schankin, Annika Berger, Daniel V Holt, Johannes C Hofmeister, Till Riedel, and Michael Beigl. Descriptive compound identifier names improve source code comprehension. In *Proceedings of the 26th Conference on Program Comprehension - ICPC '18*. ACM Press, 2018. doi: 10.1145/3196321.3196332. URL https://doi.org/10.1145/3196321.3196332.

[38] Ivonne Schroter, Jacob Kruger, Janet Siegmund, and Thomas Leich. Comprehending Studies on Program Comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 308–311. IEEE, may 2017. ISBN 978-1-5386-0535-6. doi: 10.1109/ICPC.2017.9. URL http://ieeexplore.ieee.org/document/7961527/.

[39] Michael Lee Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.

[40] Janet Siegmund. Program Comprehension: Past, Present, and Future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 13–20. IEEE, mar 2016. ISBN 978-1-5090-1855-0. doi: 10.1109/SANER.2016.35. URL http://ieeexplore.ieee.org/document/7476769/.

[41] Kristin Fjola Tomasdottir, Mauricio Aniche, and Arie Van Deursen. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering*, 2018. ISSN 19393520. doi: 10.1109/TSE.2018.2871058.

A

Code Examples

Details	Obfuscated	Transformed

example implicit predicate
Obf answer:
true
TF answer:
true
	`class Snippet {`
	`  public static void main(String[] args) {`
	`  int V1 = 10, V2 = 3;`
	`  if ( !(V1 - V2 == 0) ) {`
	`  System.out.println(”true”);`
	`  } else {`
	`  System.out.println(”false”);`
	`  }`
	`  }`
	`}`
		`class Snippet {`
		`  public static void main(String[] args) {`
		`  int V1 = 10, V2 = 3;`
		`  if (V1 - V2 > 5) {`
		`  System.out.println(”true”);`
		`  } else {`
		`  System.out.println(”false”);`
		`  }`
		`  }`
		`}`

4 infix operator precedence
Obf answer:
0
TF answer:
6
	`class Snippet {`
	`  public static void main(String[] args) {`
	`  int V1;`
	`  V1 = 2 - 4 / 2;`
	`  System.out.println(V1);`
	`  }`
	`}`
		`class Snippet {`
		`  public static void main(String[] args) {`
		`  int V1;`
		`  V1 = 3 + (9 / 3);`
		`  System.out.println(V1);`
		`  }`
		`}`

5 infix operator precedence
Obf answer:
true
TF answer:
true
	`class Snippet {`
	`  public static void main(String[] args) {`
	`  if (!greaterThanZero(-2)`
	`  && greaterThanZero(51)) {`
	`  System.out.println(”true”);`
	`  } else {`
	`  System.out.println(”false”);`
	`  }`
	`  }`
	
	`  static boolean greaterThanZero(int v) {`
	`  return v > 0;`
	`  }`
	`}`
		`class Snippet {`
		`  public static void main(String[] args) {`
		`  if (greaterThanZero(1)`
		`  && (!greaterThanZero(0))) {`
		`  System.out.println(”true”);`
		`  } else {`
		`  System.out.println(”false”);`
		`  }`
		`  }`
		
		`  static boolean greaterThanZero(int v) {`
		`  return v > 0;`
		`  }`
		`}`

6 infix operator precedence
Obf answer: true
TF answer: true

```
class Snippet {
  public static void main(String[] args) {
    String line = "The cat is black";

    boolean V1 = line.contains("dog");
    boolean V2 = line.contains("cat");
    boolean V3 = line.contains("black");

    if (V1 && V2 || V3) {
      System.out.println("true");
    } else {
      System.out.println("false");
    }
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    String line = "The cat is black";

    boolean V1 = line.contains("dog");
    boolean V2 = line.contains("cat");
    boolean V3 = line.contains("black");

    if ((V1 && V2) || V3) {
      System.out.println("true");
    } else {
      System.out.println("false");
    }
  }
}
```

---

7 post increment decrement
Obf answer: 3 5
TF answer: 3 5

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 3 + V1++;
    System.out.println(V1 + " " + V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2, V2;
    V2 = V1 + 3;
    V1++;
    System.out.println(V1 + " " + V2);
  }
}
```

---

8 post increment decrement
Obf answer: true 1
TF answer: true 1

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 0;
    if (V1++ == 0) {
      System.out.print("true ");
    } else {
      System.out.print("false ");
    }
    System.out.println(V1);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 0;
    if (V1 == 0) {
      System.out.print("true ");
    } else {
      System.out.print("false ");
    }
    V1++;
    System.out.println(V1);
  }
}
```

---

9 post increment decrement
Obf answer: false 1
TF answer: false 1

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    if (V1-- == 1) {
      System.out.print("true ");
    } else {
      System.out.print("false ");
    }
    System.out.println(V1);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    if (V1 == 1) {
      System.out.print("true ");
    } else {
      System.out.print("false ");
    }
    V1--;
    System.out.println(V1);
  }
}
```

---

10 pre increment decrement
Obf answer: 3 1
TF answer: 6 -1

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = ++V1 - 2;
    System.out.println(V1 + " " + V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 5, V2;
    ++V1;
    V2 = 5 - V1;
    System.out.println(V1 + " " + V2);
  }
}
```

11 pre increment decrement
Obf answer:
true 1
TF answer:
true 1

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    if (--V1 == 1) {
      System.out.print(”true ”);
    } else {
      System.out.print(”false ”);
    }
    System.out.println(V1);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    V1--;
    if (V1 == 1) {
      System.out.print(”true ”);
    } else {
      System.out.print(”false ”);
    }
    System.out.println(V1);
  }
}
```

12 pre increment decrement
Obf answer:
1 4
TF answer:
5 3

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = --V1 + 3;
    System.out.println(V1 + ” ” + V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 6, V2;
    V2 = 9 - V1;
    --V1;
    System.out.println(V1 + ” ” + V2);
  }
}
```

13 constant variables
Obf answer:
5
TF answer:
5

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 3;
    int V2 = V1 + 2;
    System.out.println(V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2 + 3;
    System.out.println(V1);
  }
}
```

14 constant variables
Obf answer:
4
TF answer:
6

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 2 * V1;
    System.out.println(V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 3 * 2;
    System.out.println(V1);
  }
}
```

15 constant variables
Obf answer:
5.0
TF answer:
9.0

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    System.out.println(2.5 * V1);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    System.out.println(2 * 4.5);
  }
}
```

16 remove indentation atom
Obf answer:
7
TF answer:
9

```
class Snippet {
  public static void main(String[] args)
    boolean V1 = (2 != 3);
    boolean V2 = false;
    int V3 = 3;

    if (V1)
      if (V2)
        V3 = V3 + 2;
    else
      V3 = V3 + 4;

    System.out.println(V3);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    boolean V1 = (2 != 3);
    boolean V2 = false;
    int V3 = 5;

    if (V1)
      if (V2)
        V3 = V3 + 2;
      else
        V3 = V3 + 4;

    System.out.println(V3);
  }
}
```

---

17 remove indentation atom
Obf answer:
6
TF answer:
6

```
class Snippet {
  public static void main(String[] args)
    int V1 = 5, V2 = 5;
    while (V2 > 0)
      V2--;
      V1++;
    System.out.println(V1);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 5, V2 = 5;
    while (V2 > 0)
      V2--;
    V1++;
    System.out.println(V1);
  }
}
```

---

18 remove indentation atom
Obf answer:
2 2
TF answer:
5 5

```
class Snippet {
  public static void main(String[] args)
    int V1 = 1, V2 = 2;
    if (V1 > V2)
    V2 = 1;
    V1 = 2;
    System.out.println(V1 + " " + V2);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 5, V2 = 2;
    if (V1 < V2)
      V1 = 2;
    V2 = 5;
    System.out.println(V1 + " " + V2);
  }
}
```

---

25 conditional operator
Obf answer:
1
TF answer:
1

```
class Snippet {
  public static void main(String[] args)
    int V1 = 4;

    int V2 = V1 == 3 ? 2 : 1;

    System.out.println(V2);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 4;
    int V2 = 3;
    int V3;

    if (V1 == 3) {
      V3 = 2;
    } else {
      V3 = 1;
    }

    System.out.println(V3);
  }
}
```

26 conditional operator
Obf answer: 5
TF answer: 2

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 3;
    int V2 = 5;
    int V3 = 2;

    int V4 = V1 == 3 ? V2 : V3;

    System.out.println(V4);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 3;
    int V2 = 5;
    int V3 = 2;

    int V4;
    if (V1 == 2){
      V4 = V2;
    } else{
      V4 = V3;
    }

    System.out.println(V4);
  }
}
```

---

27 conditional operator
Obf answer: 1
TF answer: 1

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 3;
    int V3 = 1;

    int V4 = V1 == 3 ? V2 : V3;

    System.out.println(V4);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 3;
    int V3 = 1;

    int V4;
    if (V1 == 3) {
      V4 = V2;
    } else {
      V4 = V3;
    }

    System.out.println(V4);
  }
}
```

---

28 arithmetic as logic
Obf answer: true
TF answer: true

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 8;

    if ((V1 - 3) * (7 - V1) <= 0) {
      System.out.println(”true”);
    } else {
      System.out.println(”false”);
    }
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 8;

    if (3 <= V1 || V1 >= 7) {
      System.out.println(”true”);
    } else {
      System.out.println(”false”);
    }
  }
}
```

---

29 arithmetic as logic
Obf answer: false
TF answer: false

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;

    if ((V1 - 2) * (6 - V1) > 0) {
      System.out.println(”true”);
    } else {
      System.out.println(”false”);
    }
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;

    if (V1 < 2 || 6 < V1) {
      System.out.println(”true”);
    } else {
      System.out.println(”false”);
    }
  }
}
```

30 arithmetic as logic
Obf answer: true
TF answer: true

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 5;

    if (V1 + 5 != 0) {
      System.out.println("true");
    } else {
      System.out.println("false");
    }
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 5;

    if (V1 != -5) {
      System.out.println("true");
    } else {
      System.out.println("false");
    }
  }
}
```

40 logic as control flow
Obf answer: 4 10
TF answer: 6 8

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 1;
    int V2 = 5;

    if (++V1 > 0 || ++V2 > 0) {
      V1 = V1 * 2;
      V2 = V2 * 2;
    }

    System.out.println(V1 + " " + V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 4;

    if (++V1 > 0) {
      V1 = V1 * 2;
      V2 = V2 * 2;
    } else if (++V2 > 0) {
      V1 = V1 * 2;
      V2 = V2 * 2;
    }

    System.out.println(V1 + " " + V2);
  }
}
```

41 logic as control flow
Obf answer: 1 6
TF answer: 2 7

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 1;
    int V2 = 5;

    boolean _test =
      V1 == V2 && ++V1 > 0 || ++V2 > 0;

    System.out.println(V1 + " " + V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 6;

    if (V1 == V2) {
      ++V1;
    } else {
      ++V2;
    }

    System.out.println(V1 + " " + V2);
  }
}
```

42 logic as control flow
Obf answer:
8 5 5
TF answer:
11 11 10

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 3;
    int V2 = 5;
    int V3 = 0;

    while (V3 < V2 && ++V1 > 0) {
      V3++;
    }

    System.out.println(V1 + " " +
                V2 + " " + V3);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 1;
    int V2 = 11;
    int V3 = 0;

    while (V1 != V2) {
      ++V1;
      if (!(V1 > 0))
        break;

      V3++;
    }

    System.out.println(V1 + " " +
                V2 + " " + V3);
  }
}
```

---

43 repurposed variables
Obf answer:
2 0
TF answer:
4 0

```
class Snippet {
  public static void main(String[] args) {
    int V1[] = new int[5];
    V1[4] = 3;

    while (V1[4] > 0) {
      V1[3 - V1[4]] = V1[4];
      V1[4] = V1[4] - 1;
    }

    System.out.println(V1[1] + " "
                    + V1[4]);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1[] = new int[6];
    int V2 = 5;

    while (V2 > 0) {
      V1[5 - V2] = V2;
      V2 = V2 - 1;
    }

    System.out.println(V1[1] + " " + V2);
  }
}
```

---

44 repurposed variables
Obf answer:
4
TF answer:
1

```
class Snippet {
  public static void main(String[] args) {
    int V3 = 0;

    for (int V1 = 0; V1 < 2; V1++) {
      for (int V2 = 0; V1 < 2; V1++) {
        V3 = 4 * V1 + V2;
      }
    }

    System.out.println(V3);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V3 = 0;

    for (int V1 = 0; V1 < 2; V1++) {
      for (int V2 = 0; V2 < 2; V2++) {
        V3 = 4 * V1 + V2;
        V1 = V2;
      }
    }

    System.out.println(V3);
  }
}
```

45 repurposed variables
Obf answer: 2
TF answer: 4

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 0;
    for (int V2 = 0; V2 < 2; V2++) {
      V1 = (V2 < 1) ? 1 : 0;
      if (V1 > 0) {
        V1 = V2 + 5;
      } else {
        V1 = V1 + 2;
      }
    }
    System.out.println(V1);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 0;
    for (int V2 = 0; V2 < 2; V2++) {
      int V3 = (V2 < 1) ? 1 : 0;
      if (V3 > 0) {
        V1 = V2 + 4;
      } else {
        V1 = V3 + 4;
      }
    }
    System.out.println(V1);
  }
}
```

49 dead unreachable repeated
Obf answer: 2
TF answer: 2

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 1;

    V1 = 3;
    V1 = 2;

    System.out.println(V1);
  }
}
```

```
class Snippet {
{ public static void main(String[] args) {
   int V1 = 1;

   V1 = 2;

   System.out.println(V1);
  }
}
```

50 dead unreachable repeated
Obf answer: 1
TF answer: 1

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 1;

    if (0 > 2) {
      V1 = 3;
    }

    System.out.println(V1);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 1;

    System.out.println(V1);
  }
}
```

51 dead unreachable repeated
Obf answer: 0
TF answer: 0

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 0;

    V1 = V1;

    System.out.println(V1);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 0;

    System.out.println(V1);
  }
}
```

53 change of literal encoding
Obf answer: 11
TF answer: 11

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 013;

    System.out.println(V1);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = Integer.parseInt("13", 8);

    System.out.println(V1);
  }
}
```

54 change of literal encoding
Obf answer:
0
TF answer:
0

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 11 & 32;

    System.out.println(V1);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 0b1100 & 0b0011;

    System.out.println(V1);
  }
}
```

55 omitted curly braces
Obf answer:
3
TF answer:
3

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;

    if (V1 <= 0) V1++; V1++;

    System.out.println(V1);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;

    if (V1 <= 0) {
      V1++;
    }
    V1++;

    System.out.println(V1);
  }
}
```

56 omitted curly braces
Obf answer:
5 3
TF answer:
8 3

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 4;

    int V2 = 0;
    while (V2 < 3) V2++; V1++;

    System.out.println(V1 + " " + V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 7;

    int V2 = 1;
    while (V2 < 3) {
      V2++;
    }
    V1++;

    System.out.println(V1 + " " + V2);
  }
}
```

57 omitted curly braces
Obf answer:
7
TF answer:
8

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 3;

    for (int V2 = 0;
         V2 < 3; V2++) V1++; V1++;

    System.out.println(V1);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 4;

    for (int V2 = 0;
         V2 < 3; V2++) { V1++; } V1++;

    System.out.println(V1);
  }
}
```

58 type conversion
Obf answer:
1
TF answer:
2

```
class Snippet {
  public static void main(String[] args) {
    float V1 = 1.99f;

    int V2 = (int) V1;

    System.out.println(V2);
  }
}
```

```
class Snippet {
  public static void main(String[] args) {
    float V1 = 2.87f;

    int V2 = (int)Math.floor(V1);

    System.out.println(V2);
  }
}
```

60 type conversion
Obf answer: 32
TF answer: 32

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 288;
    byte V2 = (byte)V1;

    System.out.println(V2);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 288;

    byte V2 = (byte) (V1 % 256);

    System.out.println(V2);
  }
}
```

61 indentation
Obf answer: 4
TF answer: 4

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 0;
    int V2 = 2;

    if (V1 > 0) {}
      V2 = 4;

    System.out.println(V2);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 0;
    int V2 = 2;

    if (V1 > 0) {}
    V2 = 4;

    System.out.println(V2);
  }
}
```

62 indentation
Obf answer: 3
TF answer: 10

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 0;
    int V2 = 1;

    if (V1 > 0) {
      V2 = 2;
    }
      V2 = V2 * 3;

    System.out.println(V2);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 0;
    int V2 = 5;

    if (V1 > 0) {
      V2 = 2;
    }
    V2 = V2 * 2;

    System.out.println(V2);
  }
}
```

63 indentation
Obf answer: 7
TF answer: 9

```
class Snippet {
  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 0;
    int V3 = 3;

    if (V1 > 0) {
      if (V2 > 0) {
        V3 = V3 + 2;
    } else {
      V3 = V3 + 4;
    }
    }

    System.out.println(V3);
  }
}
```

```
class Snippet {
{  public static void main(String[] args) {
    int V1 = 2;
    int V2 = 0;
    int V3 = 5;

    if (V1 > 0) {
      if (V2 > 0) {
        V3 = V3 + 2;
      } else {
        V3 = V3 + 4;
      }
    }

    System.out.println(V3);
  }
}
```

Table A.1: Java Code Examples.

B

Ethics Committee Approval

Date 25-06-2019
Contact person Ir. J.B.J. Groot Kormelink, secretary HREC
Telephone +31 152783260
E-mail j.b.j.grootkormelink@tudelft.nl

Human Research Ethics Committee
TU Delft
(http://hrec.tudelft.nl/)

Visiting address
Jaffalaan 5 (building 31)
2628 BX Delft

Postal address
P.O. Box 5015
2600 GA Delft
The Netherlands

Re:

Dear Chris Langhout,

Thank you very much for informing us about the human research by students as part of your course and for referring them to the current human research ethics and GDPR guidelines.

We have no further questions at this this time.
Your application will be stored in accordance with the existing regulations.
In case of any problems, please contact us.

With kind regards,

Joost Groot Kormelink - TPM
Secretary Chair Human Research Ethics Committee TU Delft

C

Screenshots of the Online Survey

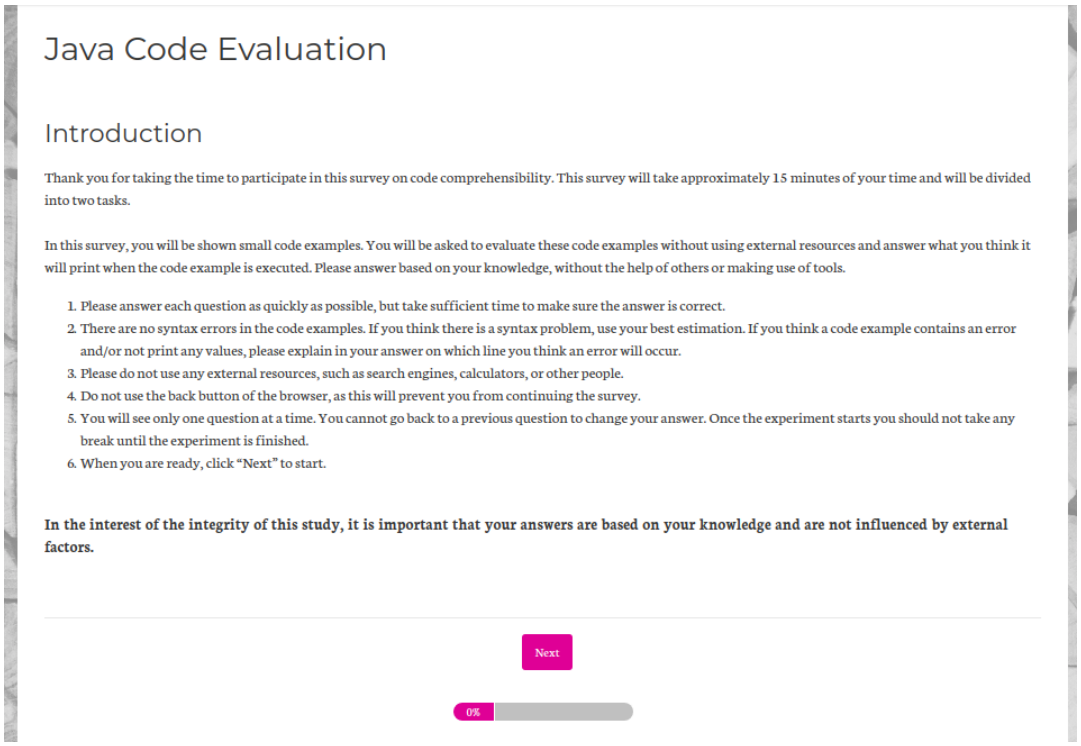

Figure C.1: Survey introduction

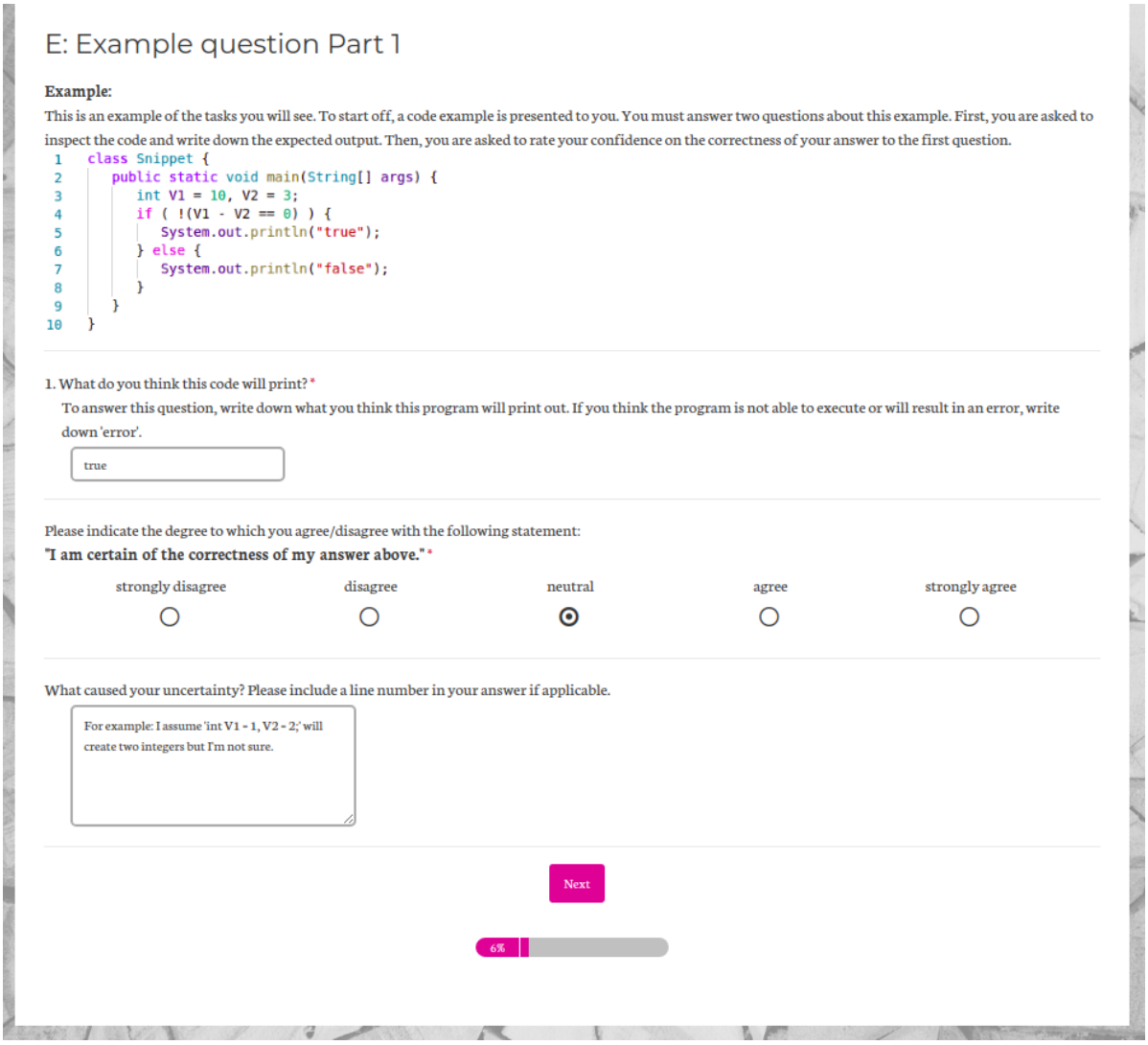

Figure C.2: The example question of part 1

Figure C.3: The page design of part 1, the participant is shown one of the 80 available code questions.

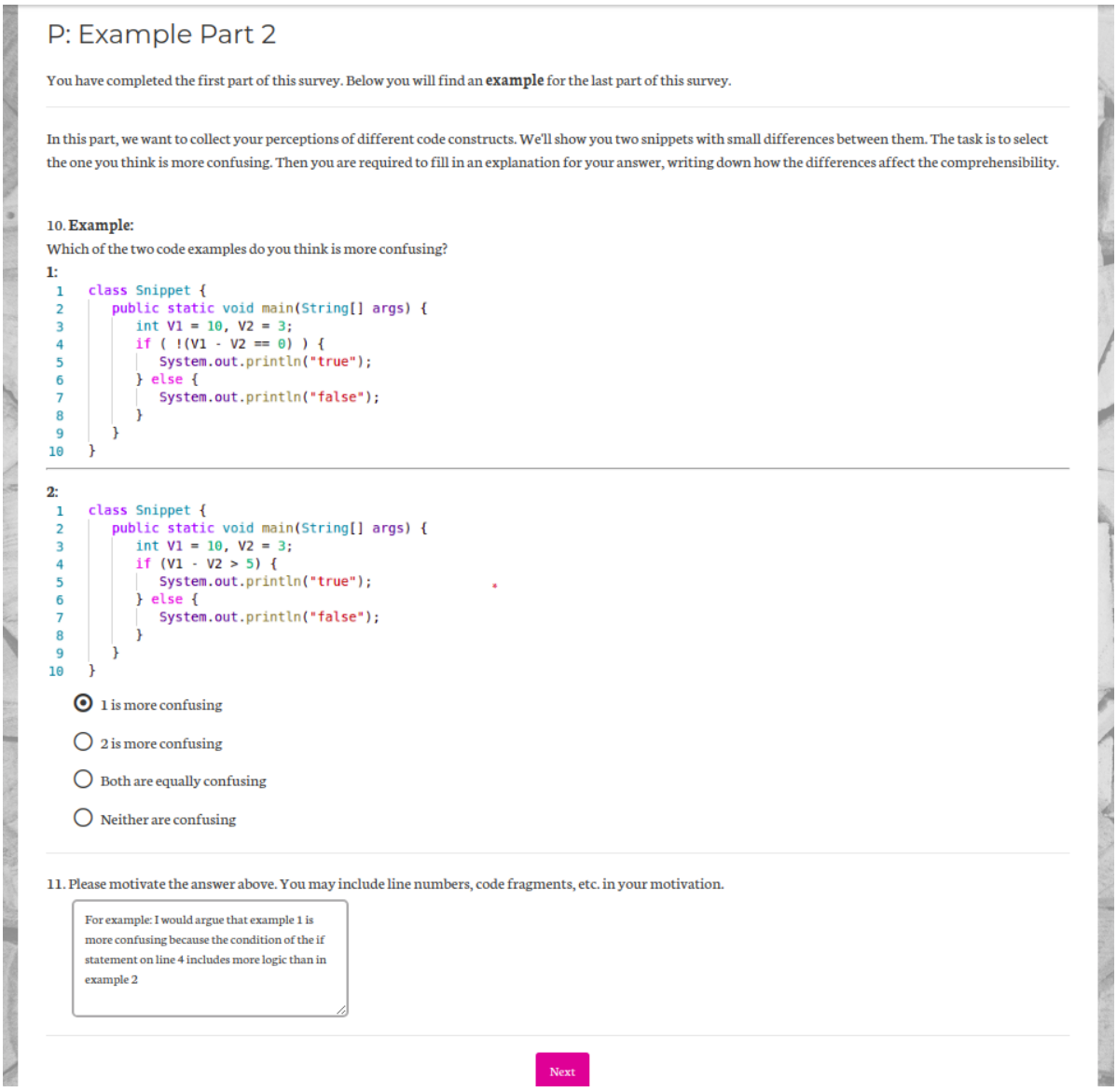

Figure C.4: The example question of part 2

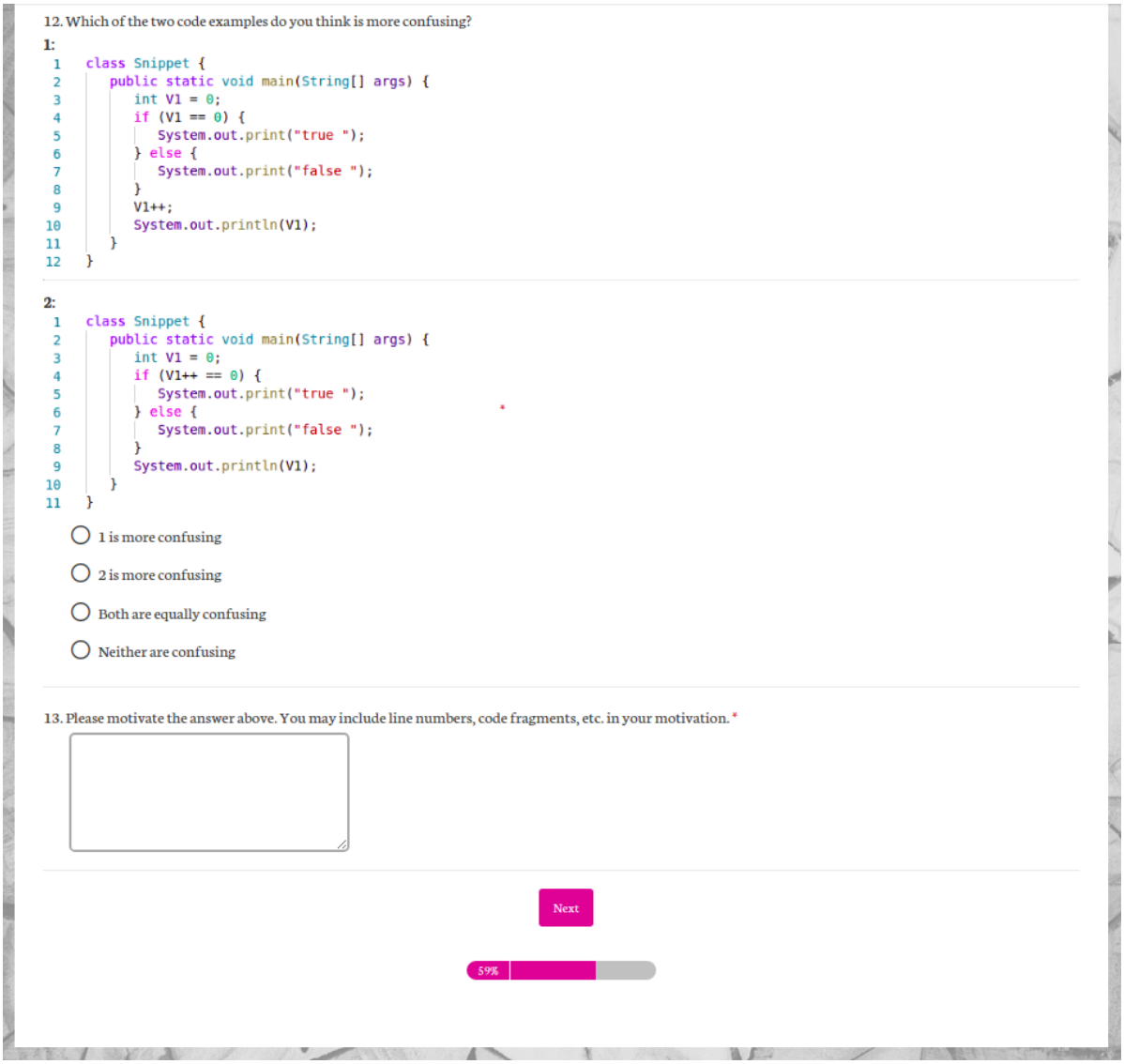

Figure C.5: A page of part 2, asking the participant to compare a randomly selected pair of code examples.

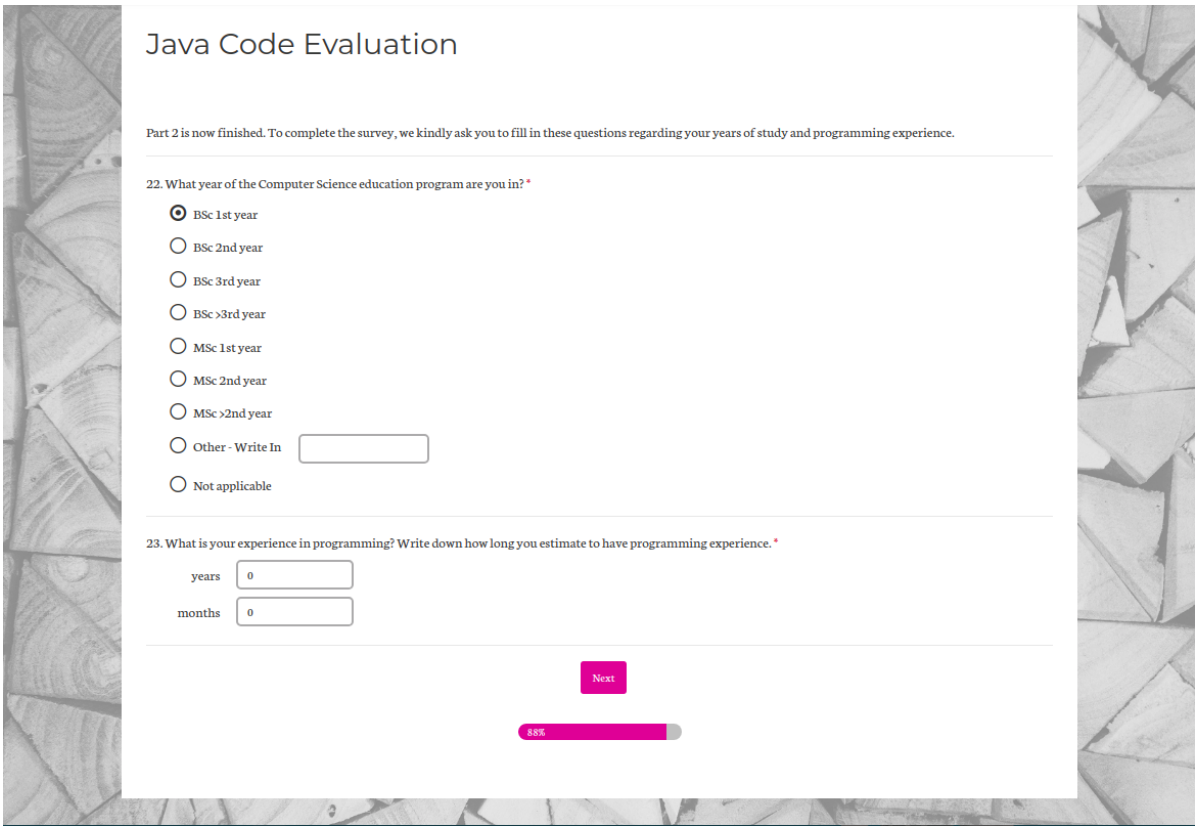

Figure C.6: End of Survey questions

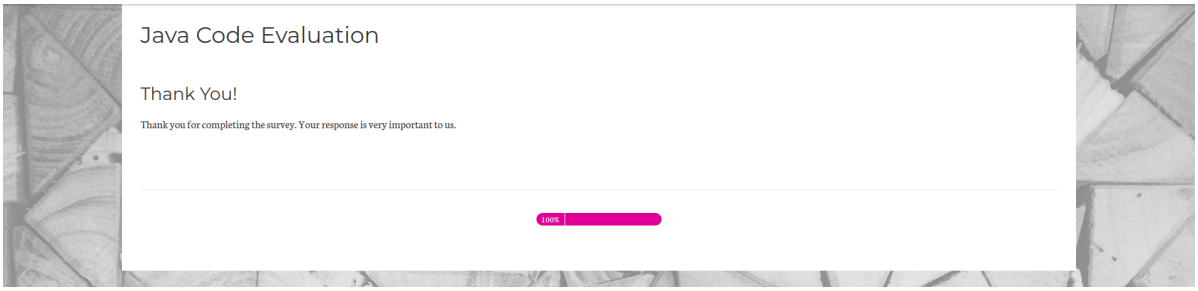

Figure C.7: Thank you page