On the comprehensibility of functional decomposition: An empirical study

Ewan Tempero
University of Auckland
Auckland, New Zealand
e.tempero@auckland.ac.nz

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

James Finnie-Ansley
University of Auckland
Auckland, New Zealand
james.finnie-ansley@auckland.ac.nz

Andrew Luxton-Reilly
University of Auckland
Auckland, New Zealand
a.luxton-reilly@auckland.ac.nz

Diana Kirk
University of Auckland
Auckland, New Zealand
diana.kirk@auckland.ac.nz

Juho Leinonen
University of Auckland
Auckland, New Zealand
juho.leinonen@auckland.ac.nz

Asma Shakil
University of Auckland
Auckland, New Zealand
asma.shakil@auckland.ac.nz

Robert Sheehan
University of Auckland
Auckland, New Zealand
r.sheehan@auckland.ac.nz

James Tizard
University of Auckland
Auckland, New Zealand
james.tizard@auckland.ac.nz

Yu-Cheng Tu
University of Auckland
Auckland, New Zealand
yu-cheng.tu@auckland.ac.nz

Burkhard C. Wünsche
University of Auckland
Auckland, New Zealand
burkhard@cs.auckland.ac.nz

ABSTRACT

Folk-wisdom in software engineering suggests that small functions that adhere to the principle of single-responsibility have several advantages over longer, monolithic functions, including improvement in code comprehension. Despite this widespread view, empirical research on the impact of functional decomposition on understanding code is sparse, yet it is central to software development practices.

In this study, we investigated the impact of functional decomposition on understanding using a controlled experiment in which participants were tasked with comprehending code for two different functionalities, each implemented as either a single function or multiple functions, and recorded the reading time, code description accuracy, and behaviour question responses.

Despite a carefully constructed empirical study, we find that the influence of function decomposition on code understanding is inconclusive, suggesting that functional decomposition does not *universally* enhance code comprehensibility, and context-aware guidelines for code structuring may promote better comprehensibility. Our negative result contributes to the ongoing refinement of software engineering best practices for creating more maintainable software.

ICPC '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0586-1/24/04…$15.00
https://doi.org/10.1145/3643916.3644432

CCS CONCEPTS

• **Software and its engineering → Abstraction, modeling and modularity**; *Maintaining software.*

KEYWORDS

Functional decomposition, Comprehensibility, Controlled experiment, Program comprehension

ACM Reference Format:
Ewan Tempero, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, Diana Kirk, Juho Leinonen, Asma Shakil, Robert Sheehan, James Tizard, Yu-Cheng Tu, and Burkhard C. Wünsche. 2024. On the comprehensibility of functional decomposition: An empirical study. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3643916.3644432

1 INTRODUCTION

Decomposition is regarded as beneficial to software quality. One decomposition mechanism is functions. This raises the question of what benefits accrue from decomposing code into multiple functions. In this paper, we present the results of a study meant to be a step on the way to answering this question. This study is a controlled experiment whose independent variable is whether functionality is implemented as a single function or decomposed into multiple functions, and the dependent variable is comprehensibility.

One attribute of software quality is *comprehensibility*, that is, how easy it is to understand the code. Fowler [16] says of functions "In our experience, the programs that live best and longest are those with short functions." . He goes on to say "Since the early days of programming, people have realized that the longer a function is, the *more difficult it is to understand*" (emphasis ours). While

Fowler refers to function size, he is really talking about how code is decomposed into functions, suggesting choosing a decomposition that results in small(er) functions over long(er) functions.

Other respected commentators advocate small functions over long functions. Martin is very adamant: "The first rule of functions is they should be small. The second rule of functions is that *they should be smaller than that*" [20].

Yet when we present such advice to our students, we get questions such as "But won't it be hard to keep track of all the functions?" This seems a plausible concern. If the functionality is the same, then the smaller the functions, the more there will be, and there will be the problem of understanding how the functions interact to provide the functionality.

The general belief is that the benefits of multiple small functions outweigh the potential disadvantages. For example, A blog post by *Sam Koblenski*[1] notes the disadvantage "*That may be counter-intuitive because you would think having so many functions with only a handful of code in them would force you to jump around the code a lot more, searching for the implementation of this or that function* but goes on to discuss how small functions will help with code understanding, for example, saying "*Finding that code block is actually easier when you don't have to sift through hundreds of lines of irrelevant code in the same function.*"

Despite the general belief that decomposing code into small functions improves comprehensibility, there is little evidence to support this (see Section 2). We decided to investigate these claims, starting with the simplest comparison between choices of decomposition: none at all (a single function) versus multiple functions.

We designed a between-subjects experiment following a similar methodology used by past program comprehension studies (e.g. [19]). Our results did not support the belief that multiple functions are more comprehensible than a single function. Such decomposition may improve comprehensibility in some cases, but it is not universally the case. This suggests that the consequences of decomposition are quite complex and so deserve in-depth study.

The rest of this paper is organised as follows. In the next section, we present relevant background material and related research. In Section 3 we detail the methodology we used. We present the results of our study in Section 4 and discuss them in Section 5. We present our conclusions in Section 6. The materials used in our experiment can be found in our replication package [40].

2 BACKGROUND AND RELATED WORK

The importance of program comprehension. Program comprehension is an essential aspect for both the maintenance and enhancement of an existing code-base. Much previous research has highlighted the importance of comprehension activities, finding that they consume a significant proportion of development time [8], [14], [18], [23], [45]. Zelkowitz et al.'s early work (1978) claimed that program comprehension accounts for more than half the time spent maintaining software [45]. Zelkowitz's claim was later supported by work from Corbi [8], and Fjeldstad and Hamlen [14]. Minelli et al.'s and Xia et al.'s more recent studies (2015, 2017) were also in line with Zelkowitz [23], [44]. For example, Minelli investigated IDE interactions from 18 developers, observing over 700 working hours, and found the developers spent (on average) 70% of their time on program comprehension activities. Ko et al. also found that comprehension consumed a significant proportion of software maintenance time. They used controlled experiments, with 10 participants performing debugging tasks, finding the developers used approximately 35% of their time on program comprehension [18].

The factors that affect program comprehension. A number of previous studies have investigated the factors that impact program comprehension. Both Siegmund et al. and Xia et al. investigated the relationships between programmer experience and program comprehension [34], [44]. Xia et al. found that senior developers spend a significantly smaller proportion of their time on program comprehension, compared to junior developers. Looking at the impact of code design, Teasley found that naming style within programs has an impact on comprehension [38]. Lawrie et al. showed that full word identifiers (class, method, and variable names) lead to the best comprehension, compared to single letters, and abbreviations [19].

Controlled experiments in program comprehension. There has been much research in program comprehension that has used controlled experiments. For example, Wyrich et al. found 95 studies that measured bottom-up code comprehension using human participants [43]. However such experiments must be performed with care as they are difficult to do well [11, 35, 36].

Modularity and Comprehension - Wisdom. The association between modular structure and ease of understanding has long been accepted by the software community as a given. In 1972, Parnas described modularisation as "a mechanism for improving the flexibility and comprehensibility of a system" [27]. The Boehm et al. product quality model cites *Conciseness* as an attribute of *Understandability* and states that a requirement of conciseness is that programs should neither be "excessively fragmented into modules, ..., functions and sub-routines" nor that "the same sequence of code is repeated in numerous places, rather than defining a subroutine ..." [3]. This suggests a relationship between aspects of modularity (modules, functions and sub-routines) and understandability. Meyer related the design of modules to understandability [21].

A draft of item I-0140 in the US National Institute of Standards and Technology (NIST) Public Interpretations Database has the title "Modularity is for Understandability, ..." [25]. Many popular online programming sites note the link between modularity and understandability. A post in the developer insights category in the TINY organisation states that one of the advantages of modularity is that "Code is easier to read" [22]. Caitlin Lee from Medium states that modular code is "Easier to understand each module and their purpose" [6].

Characterising Modularity. There have been several definitions of what 'modularity' is. The ISO/IEC Standard for Systems and software Quality Requirements and Engineering (SQuaRE) defines modularity as the "degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components" [15]. McCall et al. consider modularity to be synonymous with *structuredness* and the authors provide several definitions from different authors. These address aspects of combining modules and how a program

[1] https://sam-koblenski.blogspot.com/2014/01/functions-should-be-short-and-sweet-but.html

is organised. Boehm at al. relate *Conciseness*, a pre-requisite of *Understandability*, to the existence of functions and hints on the tension between too many and too few functions [3].

Parnas used the definition of modularity provided by Gauthier and Pont [17], which referred to "separate, distinct program modules", where the modules are "well-defined". It also refers to the consequences of a "good modularisation", such as modules being able to be tested independently and limiting the scope of what needs to be understood when debugging.

Booch defines it as "the property of a system that has been decomposed into a set of cohesive and loosely-coupled modules." [4, p57]. Berard defines it as "the extent to which a larger system is broken into smaller, easily integrated, easily maintained, easily tested, easily reused, components;" [2, p334]. Pfleeger suggest that "In a modular design, the components have clearly defined inputs and outputs and each component has a clearly stated purpose." [28, p207].

The various programming languages and paradigms define the term 'module' in different ways. The Python Foundation defines module as "a file containing Python definitions and statements" [30]. Findlay and Watt describe a module in Pascal as "either a single subprogram or a related group of subprograms" [12]. The Java documentation describes a module as a "set of packages" where "The members of a package are classes and interfaces" [26].

Modules are hierarchical and describe code structures that range from a single function or subroutine to a large program containing many packages, files and/or classes. In this study, we focus on single functions.

Modularity and Comprehension - Studies. We found few studies that formally investigate the links between modular code and understanding. We overview these below.

Alardawi and Agil investigated the effects of class structure on program comprehension [1]. They carried out three controlled experiments on 211 first year programming students from three different institutions. Most of the participants had no previous experience in object-oriented programming but had some experience with Java (one institution) or Visual Basic (two institutions). The programs presented to students were versions of a simple program that required no domain knowledge. Participants were divided into two matched groups and given a version of either a class-based or a non-class-based version and some questions to establish comprehension. Visual Basic was used for two of the experiments and Java for the third. Responses were timed. Statistical analysis (Mann-Whitney U non-parametric test) showed that students given the class-based versions performed better in two experiments (one Visual Basic and one Java) and no effect was observed for the third.

For this experiment, modularity is characterised by class structure. The size of the program is not given but is described as "larger compared to those used in prior related studies" but "still small compared to most OO applications" [1]. Our interest is in modularity as characterised by the use of functions and so our goals are different to those of Alardawi and Agil.

Tempero at al. conducted an experiment to investigate how modularity affected students' ability to understand and change code [39]. The motivation for the study was the observation that students struggled to produce designs that were modular, "despite the fact that the assessment criteria included a requirement for good modularity". Forty fourth year software engineering and postgraduate computer science students were asked to modify the output of a small Java program that analysed an input data file and produced a list of module names. Each student was assigned either the code with lower class dependence with respect to the task (higher modularity) or with higher class dependence (lower modularity). Students' attempts at running the supplied unit tests were logged to ascertain success in changing the program and students were asked questions about the program to assess their understanding. The authors found that the high modularity design was associated with greater success in changing the code but tended to be associated with lower understanding, indicating a "tension between understandability and modifiability" [39]. For this study, the programs comprised multiple modules, with modularity characterised by dependencies among modules. For our study, programs are smaller, with modularity characterised by the use of functions.

Several studies have been carried out to understand the difference in the mental representations created by novice programmers when shown programs in the procedural and object-oriented styles [32, 41, 42]. Ramalingam and Wiedenbeck carried out a study to determine the difference in the mental representations created by novice programmers when shown programs in the procedural and object-oriented styles [32]. 75 students in an introductory programming course were shown brief C++ program segments, three written in the procedural style and three in the OO style. They then answered questions relating to operations, control flow, data flow, state and function. Programs fitted on one page, with the OO program containing one class with no complex features, for example, polymorphism and inheritance. The imperative version contained a main function only. They found that students made a lower number of errors in the procedural version. Wiedenbeck et al. extended the study to larger programs and found no significant difference between the number of questions answered correctly but found differences in the types of question answered correctly [42].

Although the goal of these studies was focused on mental models, the Ramalingen and Wiedenbeck study relates to ours, in that the programs used were small and essentially differed in the use of functions.

Sellitto et al. investigated the effects of refactoring on Program Comprehension. The study was based on 156 open source projects and Scalabrino et al.'s eight readability metrics used for the analysis [33]. This and other similar studies include other aspects of readability and are more general than ours, which focuses on the use of functions.

3 METHODOLOGY

The overall goal of our research project is to understand how choice of functions affects comprehensibility of code. Specifically, in the study reported in this paper, our research question is:

RQ: *Is code providing some functionality organised as multiple functions easier to understand than code providing the same functionality as a single function?*

In order to discuss our methodology, we need to refer to the *functionality* being implemented, the implementations are described

in terms of the *functions* it contains. In order to reduce the confusion, we will use the following terminology.

We will use *operation* to indicate which functionality we are referring to, so no two operations provide the same functionality. An operation has multiple implementations. We will refer to different implementations as a *version*. We will use *SFV* to refer to the implementation that consists of a single function, that is, the *single-function version*. *MFV* will refer to the multiple-function version.

3.1 Overview

The general methodology we have adopted is as follows:

(1) Identify some operations.
(2) Create an SFV and a MFV for each operation.
(3) Have human participants perform comprehension tasks on the different versions for the operations and measure their performance.

There are many variables that need to considered:

- What operation to choose. In particular, the size of the operation may be important. Some effects may not be visible in small amounts of code, however larger amounts of code will take more time, risking participant fatigue or incompletion.
- Operation familiarity. If the operation is well-known, the participant may be able to successfully guess answers to questions independently of how the code is presented. [19]
- Number of operations. The nature of the operation may have an effect. This suggests having more than one operation. However with a limited participant pool, having too many versions means the sample size per version may not be big enough for any effect to be visible.
- The nature of the operation. The essential complexity of different tasks may be different [5, Chap.16]. It would be reasonable to expect that operations with different levels of essential complexity will require different amounts of effort to understand independent of the nature of the implementation.
- The nature of the implementation. There are many ways to implement the same operation, and even limiting in one dimension (e.g. having a single function), there are still many possibilities. It may be that some details have a bigger effect than how the code is decomposed. For example, it is generally believed that choice of identifier name effects comprehensibility. We need to ensure that the only variable that varies is the decomposition. We discuss this further in Section 3.2.
- The choice of SFV and MFV for the same operation. We need to ensure that the difference between the SFV and MFV is only in the decomposition. We discuss this further in 3.2.
- The choice of decomposition. It may be that some decompositions are harder to understand than others. We discuss this further in 3.2
- Recruitment. What participants we have is impacted by how we recruit. For example, are the participants students or professionals, are they volunteers or not, or are they rewarded for their time.
- Study environment. Are the tasks in the experiment performed in-person or online? Are the participants known or anonymous? Is the environment they use familiar to them (e.g. their standard IDE) or not?
- As the experiment involves human participants, we must have ethics approval. There are a number of requirements that must be met to acquire this.
- Operationalisation of dependent variable. In this case the variable is comprehensibility. As a construct [31], it is not subject to direct measurement, so we have to ensure what we do measure provides information for our variable.
- Choice of tasks. This is related to the operationalisation. What we refer to as comprehension has many facets, and different tasks may involve different facets [11] and allow for different kinds of measurements.
- Experimental design. Is it within subject or between subject? Within subject potentially gives more data but risks a learning effect, and takes more time.

Avoiding the risk for all variables is not feasible. There are a number of potential threats to validity that must be addressed (see, e.g. [11, 35]). So, as with any experimental design, trade-offs must be made to minimise such threats. As this was our first study in this area there were many unknowns, so our philosophy was to minimise the cost.

We followed, where possible, the checklist provided by Siegmund and Schumann ([35, Appendix]) and the advice of Feitelson [11]. Below we discuss the decisions we made to mitigate various threats, and revisit possible threats to validity in Section 5.1.

We choose to maximise internal validity over external validity [36] as the cost of addressing both would be too high. Thus we limited our study to a single language (Java) and limited our population to students (see Section 3.4). We ran our study in a regularly-scheduled class in a relevant course in order to have a fairly uniform sample with respect to domain knowledge, education, and programming language.

The study was presented to the class as an exercise similar to previous class exercises to reduce evaluation apprehension and the Hawthorne effect. Two of the authors and a teaching assistant were in the room during the study to ensure process conformance, however the data submitted by the participants was anonymous (see Section 3.5) to further reduce evaluation apprehension.

The tasks carried out by participants were designed to take 30 minutes in total (see Section 3.2) to avoid fatigue. The class was 50 minutes long so there should have been no concerns regarding time pressure. We did not identify the hypotheses under study, or even that different members of the class would see different treatments. This reduced possible Rosenthal effects.

We had several different tasks to reduce mono-operation bias and used two metrics for the main task to reduce mono-method bias (Section 3.3).

3.2 Artefacts

As mentioned above, how the code is presented is a key factor in ensuring that only the existence of the decomposition is the variable. These are the decisions we took to address this requirement.

We wanted to have at least two operations of different kinds. We wanted something large enough to be non-trivial, yet small enough that the whole study could be completed in about 30 minutes.

We developed several candidates with the original intention of using them all, but concluded it would be difficult to get a large-enough sample for each version so in the end chose to use only two operations. These were:

Days Since *Determines if a date is valid and if so determines the number of days of the given date since the 31st of December 1899.*
This is "computational" in nature, with a somewhat (but not completely) simple input, and a single output. We decided against a larger task such as computing the difference between two arbitrary dates, as the implementation would be too different in size to our other choice. The SFV we used is shown in Figure 2.

Weather *Report the average rainfall and humidity, and number of days with rain, from the valid data in the supplied data*
This is a version of the so-called "rainfall problem" [37]. This is different in nature to **Days Since** as it requires processing a collection of data. The requirements were extended from the original rainfall problem to give an operation of sufficient size. The SFV we used is shown in Figure 1.

One possible confounding factor is that one version has information that the other does not. In particular, the version with multiple functions will have extra information in the form of the function names. In order to mitigate this threat, we included comments in the SFV that provided the same information as the function name in the MFV. For example, in the MFV for Days Since there is a function named `isValidYear`. Matching this in the SFV is the comment shown on line 10 of Figure 2.

Space limits means we cannot provide the full MFV for the two operations, but we show the top level functions for each in Figures 3 and 4.

Another concern was the choice of identifier names more generally, not just for function names. This means using the same names in both versions for local variables where possible. This includes using actual parameter names as the formal parameter names.

There are usually many ways to implement a given function, and so we had to ensure that our particular choice did not favour the SFV over the MFV or vice versa. For example, **Weather** could be implemented as a single loop that combines the removal of invalid data and the computation of averages and number of rainy days, or it could be implemented as multiple passes through the data (e.g. see [13]). We were concerned that decomposing the single loop design into multiple functions would lead to something that looked quite different, whereas there seemed a natural decomposition of the multiple-pass design.

The choice of decomposition was also a factor, in particular how big the resulting functions should be. Martin suggests functions should not be more than 4 lines [20, p34]. We preferred a decomposition that had a similar structure to the SFV.

Different people in the team created candidates, both SFV and MFV. Then, once the **Weather** and **Days Since** were chosen, their implementations went through iterations with the team reviewing and the original author responding to the reviews.

We limited the availability of tools. In fact, we presented the code as images rather than text, so that participants could not easily copy it into an IDE or similar tool (see Section 3.5).

```
1  public class WeatherSFV {
2    private static int RAINFALL_POS = 0;
3    private static int HUMIDITY_POS = 1;
4    private static int END_OF_DATA = -999;
5
6    public double[] processData(List<List<Integer>> originalData) {
7      // Truncate the data to everything before the end of data
8      List<List<Integer>> dataUpToEnd = new ArrayList<List<Integer>>();
9      for (List<Integer> dailyData: originalData) {
10       if (dailyData.get(RAINFALL_POS) != END_OF_DATA &&
11           dailyData.get(HUMIDITY_POS) != END_OF_DATA) {
12         dataUpToEnd.add(dailyData);
13       } else {
14         break;
15       }
16     }
17     // Remove all invalid values
18     List<List<Integer>> validData = new ArrayList<List<Integer>>();
19     for (List<Integer> dailyData: dataUpToEnd) {
20       if (dailyData.get(RAINFALL_POS) >= 0 &&
21           dailyData.get(HUMIDITY_POS) >= 0) {
22         validData.add(dailyData);
23       }
24     }
25     // Determine average rainfall
26     double averageRainfall = 0.0;
27     if (validData.size() != 0) {
28       double sumRain = 0;
29       for (List<Integer> dailyData: validData) {
30         sumRain = sumRain + dailyData.get(RAINFALL_POS);
31       }
32       averageRainfall = sumRain/validData.size();
33     }
34     // Determine average humidity
35     double averageHumidity = 0.0;
36     if (validData.size() != 0) {
37       double sumHumidity = 0;
38       for (List<Integer> dailyData: validData) {
39         sumHumidity = sumHumidity + dailyData.get(HUMIDITY_POS);
40       }
41       averageHumidity = sumHumidity/validData.size();
42     }
43     // Determine number of rainy days
44     int rainyDays = 0;
45     for (List<Integer> dailyData: validData) {
46       if (dailyData.get(RAINFALL_POS) > 0) {
47         rainyDays = rainyDays + 1;
48       }
49     }
50     double[] result = { averageRainfall, averageHumidity, rainyDays };
51     return result;
52   }
53 }
```

Figure 1: Weather Single. Code has been slightly modified for presentation

3.3 Instrument

The instrument that was used presented participants with a set of tasks delivered via Qualtrics[2], an on-line survey delivery system (see also Section 3.5). The tasks are organised into sections as described below.

Demographics. This section asked questions about the programming courses participants had taken, how much programming experience they had in general, and their experience and confidence in programming in Java. This was to allow us to identify those

[2] qualtrics.com

```
public class DaysSingle {
  private final int JANUARY = 0;
  private final int FEBRUARY = 1;
  // (Remaining constants MARCH to DECEMBER omitted for presentation)
  private final int[] LENGTH_OF_MONTH =
    {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  private final int ZERO_YEAR = 1900;

  public String main(int day, int month, int year) {
    // Is the year valid?
    boolean isValidYear = year >= ZERO_YEAR;
    // Is the month valid?
    boolean isValidMonth = JANUARY <= month && month <= DECEMBER;
    int monthLength = -1;
    if (isValidMonth) {
      monthLength = LENGTH_OF_MONTH[month];
    }
    // Is the year a leap year?
    boolean isLeapYear = !(year % 4 != 0 ||
      (year % 100 == 0 && year % 400 != 0));
    if (month == FEBRUARY && isLeapYear) {
      monthLength += 1;
    }
    // Is the day in the month valid?
    boolean isValidDay = day > 0 && day <= monthLength;
    // Is the given date valid?
    if (isValidDay && isValidMonth && isValidYear) {
      int days = 0;
      // Days before the year.
      for (int aYear = ZERO_YEAR; aYear < year; aYear++) {
        days += 365;
        isLeapYear = !(aYear % 4 != 0 ||
            (aYear % 100 == 0 && aYear % 400 != 0));
        if (isLeapYear) {
          days += 1;
        }
      }
      // Days before the month
      for (int aMonth = JANUARY; aMonth < month; aMonth++) {
        days += LENGTH_OF_MONTH[aMonth];
        isLeapYear = !(year % 4 != 0 ||
            (year % 100 == 0 && year % 400 != 0));
        if (aMonth == FEBRUARY && isLeapYear) {
          days += 1;
        }
      }
      days += day;
      return String.valueOf(days);
    } else {
      return "Invalid date";
    }
  }
}
```

Figure 2: Days Since Single. Code has been slightly modified for presentation

whose responses were less to do with the treatment and more to do with their lack of capability.

Pretest. The pretest section served two purposes. It provided a more detailed assessment of the capability of the participants. It asked simple questions testing the participants' understanding of basic Java concepts that are used in the task implementations, such as use of arrays and the enhanced for-loop, by asking tracing questions. Figure 5 shows one of the questions asked.

Warmup. The warmup section gave the participants a chance to practice, so that any time they spent figuring out what they were supposed to do was done here rather than in the main study. This

```
public static double[] processData(List<List<Integer>> originalData) {
  List<List<Integer>> dataUpToEnd = truncateToEndOfData(originalData);
  List<List<Integer>> validData = getValidData(dataUpToEnd);

  double averageRainfall = averageData(validData, RAINFALL_POS);
  double averageHumidity = averageData(validData, HUMIDITY_POS);
  double rainyDays = countPositive(validData, RAINFALL_POS);

  double[] result = { averageRainfall, averageHumidity, rainyDays };
  return result;
}
```

Figure 3: The top-level function for Weather.

```
public String main(int day, int month, int year) {
  if (isValidDate(day, month, year)) {
    return String.valueOf(totalDays(day, month, year));
  } else {
    return "Invalid date";
  }
}
```

Figure 4: The top-level function for Days Since.

```
int[] list = { 1, -1, 2, -2, 3, -3 };
int count = 0;
for (int value: list) {
  if (value > 0) {
    count = count + 1;
  }
}
System.out.println(count);
```

Q: What value is printed when the code above is executed?
(a) 0 (b) 1 (c) 2 (d) 3 (e) 4 (f) Unsure

Figure 5: An example Pretest question

Instructions: Study the code presented below. You will be asked a question about this code on the following page. Once you leave this page you will not be able to return, so spend as much time as you need to understand this code. You may assume there are no errors in the code.

Figure 6: Instructions for the "reading" task

was a very abbreviated example of what they would see in the reading and behaviour sections (see below). This section was the same for all participants.

Reading. The reading section was the first part of the main study. Like similar previous studies (e.g. [19]), participants were first shown some code (e.g. Figures 1 or 2) and asked to spend time reading it to the point that they felt they understood what it did. Figure 6 shows the text of the instructions participants received for all versions.

Write 2-3 sentences explaining what the code you have just studied on the previous page does. Your description should be at a high-level but still be complete.

Figure 7: Explain in Plain English (EiPE) question text

- Given the [**Weather** code]. Suppose you had to change to the code to ignore values larger than 100. Which of the following sets of lines would you need to modify (one or more lines) to achieve this change?
  (a) 11–18 (b) 21–26 (c) 29–36 (d) 39–45 (e) 49–54
- Given the [**Days Since** code]. What is the value returned for main(1, JANUARY, 1900)? Note that 1900 is not a leap year.
  (a) 1 (b) 365 (c) 2 (d) 0 (e) Invalid date

Figure 8: Example "behaviour" questions.

Participants then advanced to the next page and were asked to provide a short description of the operation, that is, to "explain in plain english" (EiPE). They were not able to return to the page with the code. They were also asked to rate their confidence of their answer. Figure 7 shows the text of the question for all versions.

We measured the time the participants spent reading the code before moving on, and assessed the correctness of their explanation.

Behaviour. The behaviour section consisted of a set of questions (5 for **Weather**, 6 for **Days Since**) asking for a more detailed understanding of the code, by showing how it behaved with different inputs, what inputs were needed to produce specific outputs, or what part of the code would need to change to change behaviour. For each of these questions, participants were given access to the code. Figure 8 gives two examples of the questions asked.

Final. In the final section, participants were asked to rate how easy the version they saw was to understand, and they were presented with the other version and asked which of the two versions they thought was easier to understand.

3.4 Participants

We recruited our participants from a postgraduate course at The University of Auckland: "Creating Maintainable Software". The course provides an in-depth look into the development and research of maintainable software. Much of our initial material comes from Martin's *Clean Code* [20], and we critique and expand on it during the course. It uses Java as the primary programming language.

The course consists of three one-hour lectures each week over 12 weeks. A typical week will consist of two lectures presenting new content, covering topics including program comprehension, alterability, testability, dependency injection, SOLID principles, code smells, and design quality. As part of the comprehension discussion, we examine choice of identifier names (including names of methods), formatting code for comprehensibility, and choice of functions.

The third class in the week is a "discussion" session, where students discuss questions provided by the instructors in groups and post their responses to the course discussion boards. The class ends with a debrief where the whole class discusses their responses.

The experiment was conducted in the week 5 discussion class. The lectures earlier in that week had discussed advice on choosing functions that result in comprehensible code. The students did not know the nature of the experiment, in particular they did not know what the research questions were nor the nature of the treatments.

The students had 35 minutes to complete the experiment. This was followed by the usual discussion class debrief session, where the students were asked to comment on the experimental design, including speculating on what the research questions were, and what risks different aspects of the experiment were meant to mitigate.

3.5 Delivery

We used Qualtrics to administer the experiment. Qualtrics is an on-line survey platform that has different question types such as multiple choice, text entry, and matrix table. We organised each section of the experiment into blocks of questions. Each block appears on a separate page except for the warmup, reading, and behaviour sections where we explicitly display each question on a separate page.

We set up the survey in a way that all participants must answer all questions on a page before moving to the next page. Participants are unable to go back to the previous page after clicking the next button. This is to prevent participants to see the code when explaining their understanding of the code during the reading section. To prevent participants to simply copy and paste the code to any editors, we create images of the source code using Carbon[3] using the standard VSCode syntax colouring.

For each page of the survey, we use a timing question that is hidden from the participants. The timing question records how long participants view one page of the survey. It records information such as the total number of seconds that the participant spent on the page before clicking the next button. This allows us to see how long each participant spends on reading code and answering questions.

We use the randomiser feature that allows us to randomly present question blocks to participants. After the warmup section, participants will be randomly presented one of the four versions. We also utilise the branch logic feature in Qualtrics to display the appropriate code comparison questions depending on which of the four versions of code participants get to answer.

3.6 Analysis

The primary measurements used in our analysis are: the reading time spent by participants, the correctness of their explanations, and the correctness of the answers to the behaviour questions.

To evaluate correctness of explanations (EiPE), a suite of codes was created identifying different aspects of participant responses to the **Days Since** and **Weather** versions. Codes that identified correct explanations (e.g. mentioning the sentinel for the **Weather** problem, or the number of days for the **Days Since** problem) were

[3] https://carbon.now.sh

given a binary score of +1 or 0; codes identifying erroneous or incomplete explanations were given a binary score of -1 or 0. Codes were assigned until agreed upon by the researchers. Codes were then averaged for each participant.

The behaviour questions were all multiple choice questions. Participant responses were given a score of 0 for an incorrect response and a 1 for a correct response for each question. These scores were then averaged for each participant.

3.7 Experimental Design

With the context as presented, we can now present our experiment. Our independent variable is whether an implementation of an operation is presented as a single function (SFV) or decomposed into multiple functions (MFV).

The main dependent variables of interest are the time participants spent reading the code before advancing to the task of explaining, the accuracy of the explanation, and the correctness of the answers to the behaviour questions.

We used a between-subjects design, that is, each participant received only one treatment.

Our general hypothesis is that the MFV version is "easier to understand" than the SFV version. This was operationalised as:

- $\mathbf{H}_0^{\text{time}}$**:** The null hypothesis for reading time is that the time taken by participants does not depend on the version.
- $\mathbf{H}_1^{\text{time}}$**:** The alternative hypothesis is that less time is taken by participants with the MFV version than those with the SFV version.
- $\mathbf{H}_0^{\text{EiPE}}$**:** The null hypothesis for the "Explanation in Plain English" is that the correctness of the explanation does not depend on the version.
- $\mathbf{H}_1^{\text{EiPE}}$**:** The alternative hypothesis is that the explanations by participants with the MFV version are more correct than those with the SFV version.
- $\mathbf{H}_0^{\text{Beh}}$**:** The null hypothesis for the behaviour understanding is that the participant's scores to the behaviour questions do not depend on the version.
- $\mathbf{H}_1^{\text{Beh}}$**:** The alternative hypothesis is that the participants with the MFV version get higher score than those with the SFV version.

Note that these are all 1-way hypotheses. We use an alpha value of 0.05 as the threshold for statistical significance.

4 RESULTS

We had 64 complete all of the tasks, equally divided between the two operations and two versions per operation (16 each). For our analysis, we excluded those who scored below 50% for the Pretest, meaning 4 were excluded. That left us with the following group sizes: **Weather** SFV (14); **Weather** MFV (15); **Days Since** SFV (15); **Days Since**MFV (16).

For the code reading time, our hypothesis (H_1^{time}) is that participants will spend less time reading the MFVs compared to the SFVs. The code reading data is shown in Figure 9. For the **Days Since** problem, participants spent an average of 109 and 110 seconds reading the MFV and SFV treatments respectively. However, for the **Weather** problem, this relationship is reversed with participants spending an average of 244 and 184 seconds for the MFV and SFV

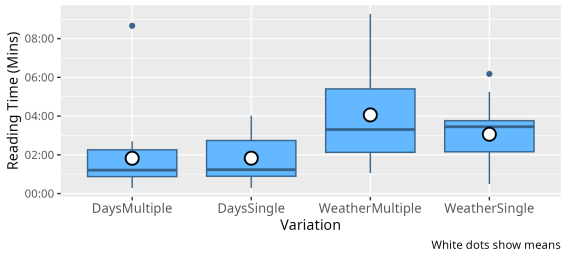

Figure 9: Reading Time by Version

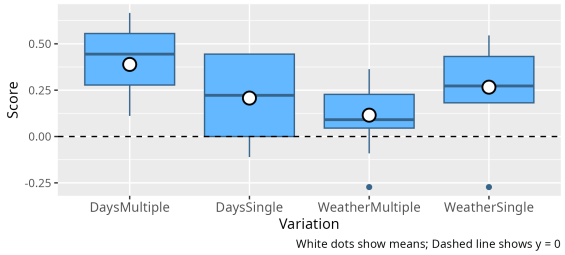

Figure 10: Explain in Plain English by Version

treatments. The differences according to a 1-tailed Mann-Whitney are not significant in either the **Days Since** (W=107, p=0.31) or **Weather** (W=120, p=0.75) cases.

For the explain in plain English question, our hypothesis (H_1^{EiPE}) is that participant explanations will be more correct for the MFVs compared to the SFVs. The explain in plain English data is shown in Figure 10. Participants scored an average of 0.39 and 0.21 for the MFV and SFV **Days Since** operation respectively and 0.12 and 0.27 for the MFV and SFV **Weather** versions. The differences according to a 1-tailed Mann-Whitney are significant for the **Days Since** case (W=171, p=0.02) but are insignificant for the **Weather** case (W=49.5, p=0.99).

For the behaviour questions, our hypothesis (H_1^{Beh}) is participants will score higher on the behaviour questions for the MFVs compared to the SFVs. The scores for the behaviour questions are shown in Figure 11. Participants scored an average of 0.71 and 0.61 for the MFV and SFV **Days Since** treatments respectively and 0.63 and 0.89 for the MFV and SFV **Weather** treatments. The differences according to a 1-tailed Mann-Whitney are not significant in either the **Days Since** (W=151, p=0.11) or **Weather** (W=44, p=1.00) cases.

In fact, by all measures, participants looking at **Weather** performed better with the SFV, whereas participants looking at **Days Since** performed better with the MFV. That is, the performance was determined by the operation, not the decomposition. It should be noted that most of the differences are not statistically significant. We discuss the results in detail below.

Figure 12 shows the results of participants' confidence in their EiPE responses. These results suggest participants were less confidence when they had the MFV of the operation.

We asked participants to give their opinion on the comprehensibility of the version they saw. Figure 13 shows the self-reported comprehensibility of the code by version as proportions. The results

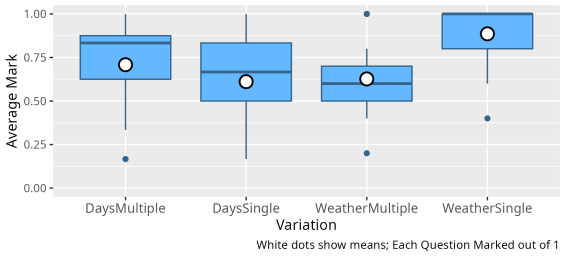

Figure 11: Behaviour by Version

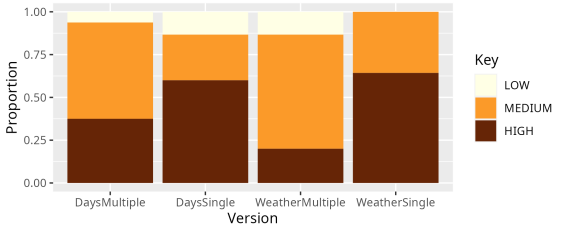

Figure 12: Confidence of EiPE by Version

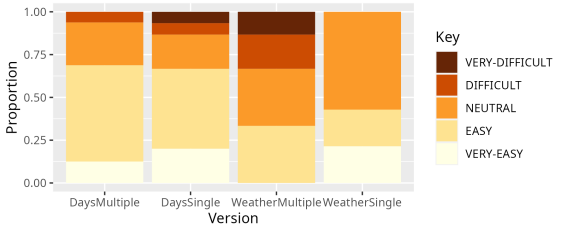

Figure 13: Self-reported Comprehensibility by Version

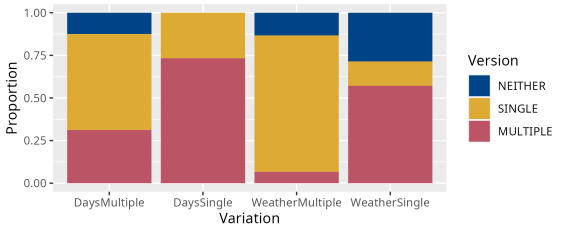

Figure 14: Version Preference by Version Seen

suggest that participants thought for **Weather** that the MFV was harder to understand, but for **Days Since** it was the SFV that was harder.

As the final task, participants were shown the other version for the operation they had worked on (i.e if they had the SFV then they were shown the MFV) and asked which of the two versions they would have preferred to work with. Figure 14 shows, for the version they saw in the main tasks, which version they would have preferred to have worked with as a proportion. The results suggest that participants preferred the version they did not work with.

5 DISCUSSION

It is generally believed that smaller functions are better than larger functions, but there has been little research as to *how* they are better. One school of thought is that smaller functions improve *comprehensibility*. Our results do not support this school of thought.

As discussed in Section 3, there are several factors to control and, consequently, several possible threats to validity, especially internal validity. The first question is whether our results are meaningful since few of our tests were statistically significant.

However, we do note that the relationships between the measurements we made were consistent: for **Weather** the SFV was always the best and for **Days Since** the MFV was the best. This was true for multiple measures of comprehensibility—the time taken to read the code (Figure 9), the accuracy of the explanations of code (Figure 10), the correctness of the code behaviour tasks (Figure 11), and the perception of the participants of the comprehensibility (Figure 13). That the relationship held for all measures provides "triangulation", which is important to provide confidence in the results.

There was some consistency in the confidence participants had for their EiPE answers (Figure 12). Participants were less confident in their answers for the MFV; however, at least for **Days Since** they performed better for that version.

One explanation for our results is that our decomposition was wrong in some way. We put considerable effort into developing the code we used, spending several meetings scrutinising the choices made in the different versions. We are acutely aware that objective measures as to what are good choices are limited, so assessing our choices has a subjective element to it. Nevertheless, we are confident that, even if our choices were not the absolute best, they were good enough. Even with hindsight, there are no glaring problems.

Another explanation is that the benefits of decomposition are only evident when dealing with implementations bigger than those used in our study. This has implications for educators who typically use small examples to demonstrate programming principles and best practices (such as decomposition). It might be that at small scales there are no obvious benefits to such practices and this may explain why students are not convinced by claims of such benefits.

This does not explain why there appears to be some benefit in some cases. This could be due to the essential complexity being a bigger factor than the decomposition at the scale we are dealing with.

It could also be that we are focusing on the wrong thing. The key insight by Parnas is that the decomposition should be based on design decisions that are likely to change, not through functional decomposition—our treatment variable [27]. It may be that the claims made about the benefit of small functions comes from how those functions are distributed across modules and how they hide design decisions that might change, rather than simply making smaller functions. That said, there is some evidence that even in this case the benefit is not due to improved comprehensibility [39].

Work in computing education may also provide some insight into our results. Cognitive Load Theory (CLT) describes how understanding is reduced when the processing required by a task exceeds the limited capacity of working memory [29]. Conceptually modeling a computer program requires mental effort to understand each module. But it also requires effort to model the interactivity

between different modules. This suggests that code comprised of many small modules that interact may prove difficult to understand if the code is not organised in ways that activate existing hierarchical schema (i.e., when the model used by the author to organise the code is not familiar to the reader).

Subgoal labels are names given to functional steps in a solution that allows a reader to "chunk" the information and reduce cognitive load, and play a similar role to the modularisation of code for comprehension purposes. Worked examples that explain a programming solution to a given problem are more effective when they use subgoal labels because the labels influence how learners mentally represent a problem [7]. Although subgoals improve performance in some contexts, in other cases there was no observable improvement [24]. These findings from computing education suggest that there are likely characteristics of code that impact the mental models formed by a reader, and subsequently comprehensibility. However, relatively few studies have measured the cognitive load imposed by different design decisions, and concrete evidence for the impact of modularisation choices on cognitive load remains an area for future work [9].

We asked participants which version (SFV or MFV) they thought would take less effort to understand (Figure 14) and asked them to comment on why they picked the version they did. The comments are perhaps illustrative of the complexity in assessing comprehensibility. For **Weather** a participant who first saw the SFV, thought the MFV would take less effort, commenting "*[The MVF] has more functions; their names give more information to help to understand.*" A participant who first saw the MFV preferred the SFV, saying "*Th[sic] comments in [the SFV] make the overall code easier to comprehend.*" We saw similar comments for **Days Since**.

5.1 Threats to Validity

Internal validity. We believe that the most significant difference between a SFV and a MFV for the same operation in our experiment is the decomposition. If there is a difference that impacts comprehension more than this decomposition, then it is not obvious what it is—but it would be extremely interesting to identify. Of particular interest is whether our use of comments in the SFV made a significant difference. Another question is whether the relatively small size of the SFV is a factor.

It is possible that some students may not have taken the study seriously and their performance may not reflect their actual understanding. However, as participants were randomly allocated to the conditions, the impact of lack of effort or of experience should be minimised. The results from the Pretest section indicate that the less capable participants were spread across the treatments, and these were removed from the analysis.

External validity. While use of students is often considered a threat to external validity, there are situations where it is appropriate (see, e.g. [10]). In our case, if our findings only apply to students then that suggests some thought should be given to how functional decomposition is taught. In fact, we speculate experienced developers would be less affected by the choice of functional decomposition.

The code was written in Java, but the difference in syntax between most languages at the function level is quite small so there is no reason to believe the choice of language is a factor. Nevertheless it a factor that needs further exploration.

Construct validity. We followed previous research in the measurements we made. Exactly what the relationship is between what we measured and our dependent variable, comprehensibility, is a matter for debate but we believe if there is an effect on comprehensibility due to the use of decomposition, the measurements we made were appropriate to detect it.

The time data came from the Qualtrics platform and from past uses of it we have no reason to doubt that data. The evaluation of the EiPE responses was mostly done by one person but checked by two others. The codes could have been scored in different ways (e.g. different weights to different codes). We tried some different options and found no difference. The evaluation of behaviour scores was also performed by one person and checked by two.

Conclusion validity. Due to the non-normality of most of our data, we had to use less-powerful non-parametric statistical tests. This means there is the possibility that a statistical relationship does exist in our data that we have not been able to identify. If such a relationship exists, it seems unlikely to affect our result.

6 CONCLUSIONS

We presented the results of a between-subjects study investigating the impact of decomposition on comprehensibility of code. Specifically, we presented participants with implementations of some functionality (which we refer to as an "operation") that was either organised as a single function, or decomposed into multiple functions. We had two operations, and two implementations of each (single or multiple functions), making a total of 4 versions. Participants were presented with one of the versions randomly via the on-line survey system Qualtrics, and were asked various questions intended to test their understanding of the code they saw.

The results were that for one operation (**Weather**) participants performed better with the single function version and for the other (**Days Since**) participants performed better with the multiple function version. This suggests that there is not a general relationship between how code is decomposed and comprehensibility.

Of course one experiment is not definitive and more similar studies are needed to confirm or refute our findings. There are many directions to pursue. Are our inconclusive results a consequence of: the essential complexity of the operations; the small amount of code; the order the functions in the multiple-function version are presented; something about the code such as choice of names; our participants; the environment we used; the programming language; or something else? If our findings do prove to be valid, this leaves the open question—what actually is the benefit of small functions?

REFERENCES

[1] Ahmed S Alardawi and Agil M Agil. 2015. Novice comprehension of Object-Oriented OO programs: An empirical study. In *2015 World Congress on Information Technology and Computer Applications (WCITCA)*. 1–4. https://doi.org/10.1109/WCITCA.2015.7367057

[2] Edward V. Berard. 1993. *Essays on object-oriented software engineering (vol. 1)*. Prentice-Hall, Inc.

[3] B. W. Boehm, J. R. Brown, and M. Lipow. 1976. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd International Conference on Software engineering*. 592–605.

[4] Grady Booch. 1994. *Object-Oriented Analysis and Design: with Applications* (2nd ed.). Addison-Wesley.

[5] Frederick P. Brooks, Jr. 1995. *The Mythical Man-Month* (20th anniversary ed.). Addison-Wesley.

[6] Caitlin Jee. 2021. *Modularization in Software Engineering*. Medium. Retrieved October 20th, 2023 from https://medium.com/@caitlinjeespn/modularization-in-software-engineering-1af52807ceed

[7] Richard Catrambone. 1998. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of experimental psychology: General* 127, 4 (1998), 355.

[8] Thomas A Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

[9] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive Load Theory in Computing Education Research: A Review. *ACM Trans. Comput. Educ.* 22, 4, Article 40 (sep 2022), 27 pages. https://doi.org/10.1145/3483843

[10] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering* 23 (2018), 452–489. https://doi.org/10.1007/s10664-017-9523-3

[11] Dror G. Feitelson. 2021. Considerations and Pitfalls in Controlled Experiments on Code Comprehension. In *2021 International Conference on Program Comprehension (ICPC)*. 106–117. https://doi.org/10.1109/ICPC52881.2021.00019

[12] W. Findlay and D.A. Watt. 1981. *Pascal: An Introduction to Methodical Programming*. Pitman Publishing Inc., Massachusetts, USA.

[13] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/2632320.2632346

[14] Richard K Fjeldstad. 1983. Application program maintenance study. *Report to Our Respondents, Proceedings GUIDE* 48 (1983).

[15] International Organization for Standardization. 2011. ISO/IEC 25010:2011: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. https://www.iso.org/standard/35733.html.

[16] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, MA, USA.

[17] Richard Gauthier and Stephen Pont. 1970. *Designing Systems Programs*. Prentice-Hall.

[18] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006), 971–987.

[19] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *14th IEEE international conference on program comprehension (ICPC'06)*. IEEE, 3–12.

[20] Robert C. Martin. 2009. *Clean Code: A handbook of agile software craftmanship*. Prentice Hall.

[21] Bertrand Mayer. 1988. *Object-oriented Software Construction*. Prentice-Hall, Inc., Hertfordshire, UK.

[22] Millie MacDonald. 2023. *Modular programming: beyond the spaghetti mess*. TINY. Retrieved October 20th, 2023 from https://www.tiny.cloud/blog/modular-programming-principle/

[23] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd international conference on program comprehension*. IEEE, 25–35.

[24] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15)*. Association for Computing Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/2787622.2787733

[25] National Institute for Standards and Technology NIST). n.d.. *Modularity is for Understandability, Maintainabilty and Testability*. Retrieved October 20th, 2023 from https://www.niap-ccevs.org/Useful_Links/PUBLIC/0140.html

[26] Oracle. n.d.. *Java Language Specification. Chapter 7: Packages and Modules*. Retrieved October 20th, 2023 from https://docs.oracle.com/javase/specs/jls/se20/html/jls-7.html

[27] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058. https://doi.org/10.1145/361598.361623

[28] Shari L Pfleeger. 1998. *Software Engineering: Theory and Practice*. Prentice Hall.

[29] Jan L Plass, Roxana Moreno, and Roland Brünken. 2010. Cognitive load theory. (2010).

[30] Python Software Foundation. 2023. *The Python Tutorial: Modules*. Retrieved October 20th, 2023 from https://docs.python.org/3/tutorial/modules.html

[31] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *22nd International Conference on Evaluation and Assessment in Software Engineering*. https://doi.org/10.1145/3210459.3210461

[32] Vennila Ramalingam and Susan Wiedenbeck. 1997. An Empirical Study of Novice Program Comprehension in the Imperative and Object-Oriented Styles. In *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers* (Alexandria, Virginia, USA) *(ESP '97)*. Association for Computing Machinery, New York, NY, USA, 124–139. https://doi.org/10.1145/266399.266411

[33] Giulia Sellitto, Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, Fabio Palomba, and Filomena Ferrucci. 2022. Toward Understanding the Impact of Refactoring on Program Comprehension. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 731–742. https://doi.org/10.1109/SANER53432.2022.00090

[34] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19 (2014), 1299–1334.

[35] Janet Siegmund and Jana Schumann. 2015. Confounding Parameters on Program Comprehension: A Literature Survey. *Empirical Softw. Engg.* 20, 4 (aug 2015), 1159–1192. https://doi.org/10.1007/s10664-014-9318-8

[36] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 9–19.

[37] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (sep 1986), 850–858. https://doi.org/10.1145/6592.6594

[38] Barbee E Teasley. 1994. The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies* 40, 5 (1994), 757–770.

[39] Ewan Tempero, Kelly Blincoe, and Danielle Lottridge. 2023. An Experiment on the Effects of Modularity on Code Modification and Understanding. In *Proceedings of the 25th Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE '23)*. Association for Computing Machinery, New York, NY, USA, 105–112. https://doi.org/10.1145/3576123.3576138

[40] Ewan Tempero, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, Diana Kirk, Juho Leinonen, Asma Shakil, Robert Sheehan, James Tizard, Yu-Cheng Tu, and Burkhard Wuensche. 2023. Replication package for "On the comprehensibility of functional decomposition: An empirical study". https://github.com/uoa-cs-pcrg/icpc2024-rene-replication

[41] Susan Wiedenbeck and Vennila Ramalingam. 1999. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies* 51, 1 (1999), 71–87. https://doi.org/10.1006/ijhc.1999.0269

[42] Susan Wiedenbeck, Vennila Ramalingam, Suseela Sarasamma, and CynthiaL Corritore. 1999. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers* 11, 3 (1999), 255–282. https://doi.org/10.1016/S0953-5438(98)00029-0

[43] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2023. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *ACM Comput. Surv.* (oct 2023). https://doi.org/10.1145/3626522 Just Accepted.

[44] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[45] Marvin V Zelkowitz, Alan C Shaw, and John D Gannon. 1979. *Principles of software engineering and design*. Prentice Hall Professional Technical Reference.