An Experiment on the Effects of Modularity on Code Modification and Understanding

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

Kelly Blincoe
The University of Auckland
Auckland, New Zealand
kblincoe@acm.org

Danielle Lottridge
The University of Auckland
Auckland, New Zealand
d.lottridge@auckland.ac.nz

ABSTRACT

Good modularity is seen as an important goal in software design. Achieving this goal is claimed to improve, among other things, the understandability and modifiability of a design. Yet, when teaching software design, we see that students limit the amount of modularity that they introduce into their code and cannot see the benefit of further modularity. This could be because they do not understand the benefits, but it could also be that these benefits are limited for inexperienced developers. In order to teach the benefits of modularity we need to understand what, if any, benefits exist for students. We conducted a controlled experiment where 40 students performed a modification task on two different designs, one with higher modularity than the other. Students were better able to successfully complete the task with the design with higher modularity. However, we found a trend where understanding was lower for the high modularity design. These results suggest modularity is beneficial to students, and that understanding of modularity needs to be better supported when teaching software design.

CCS CONCEPTS

• **Software and its engineering → Software usability**.

KEYWORDS

modularity, empirical software engineering, modifiability, code understandability

ACM Reference Format:
Ewan Tempero, Kelly Blincoe, and Danielle Lottridge. 2018. An Experiment on the Effects of Modularity on Code Modification and Understanding. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Modularity is regarded as an important quality attribute of any software design. A modular design is believed to lead to better understandability and modifiability [28]. This means modularity is a topic of significant importance when teaching software design.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

However we have found that students struggle to learn this concept. Our experience is that students struggle to come up with designs that might reasonably be considered "modular". Over multiple years, when fourth year Software Engineering students were asked to implement a relatively small Java application, they produced implementations with poor modularity, despite the fact that the assessment criteria included a requirement for good modularity. When asked to evaluate the modularity of their designs, those that choose designs with poor modularity often reported them as being easier to understand, seemingly in contradiction to the stated benefits of modularity, modifiability and understandability.

While our experience is anecdotal, similar concerns have been reported by others [10]. Cai et al. suggest the problem is due to students not receiving explicit feedback on their designs [8]. While this seems plausible, we wonder whether there is also the issue that students simply are unable to perceive increased modularity past a certain point. This could be because in fact there is no significant benefit to students—the nature of the programming assignments they do means the benefits are too small to see. Or it could be that the benefits are there, but the students are simply not aware of them.

In order to improve how modularity is taught, we need to better understand where the difficulty lies. We conducted a controlled experiment to improve our understanding of how students engage with more or less modular code. We designed a between-subjects experiment to investigate whether higher modularity leads to higher modifiability and better understanding for students.

The rest of our paper is organised as follows. In the next section we discuss the relevant background and related work. We then discuss our motivation in more detail and our experiment methodology in Section 3. Section 4 presents our results. We then discuss the implication our results have on how to teach modularity in Section 5, and finally present our conclusions.

2 RELATED WORK

There has been much research discussing software modularity. Parnas describes the benefits of modular programming include the ability to make substantial changes to one module without needing to change others, and allowing a system to be studied on module at a time [28]. That is, modularity leads to better understandability and modifiability.

While he was not the first to discuss this concept, Parnas founded its study as an academic discipline by suggesting that different criteria result in different modularisations of a design [28]. He argued that choosing modules based on the criteria of information to hide lead to better designs than modules that perform tasks. He argued that, of the two, the information hiding decision was

easier to understand and easier to change. The information hiding principle involves decisions that are consistent with choosing good classes in object-oriented design, and so teaching modularity based on information hiding is essentially teaching good object-oriented design.

Modularity is a quality attribute of software that has an appealing intuition but it is difficult to find a clear definition. Parnas used the definition of modularity provided by Gauthier and Pont[15], which referred to "separate, distinct program module", 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. Unfortunately this does not help with assessing a given design as to how modular it is.

There are been a number of software quality models that attempt to explain the different software quality attributes and their relationships. One of the earliest software quality models was by McCall et al. (generally referred to as "McCall's Quality Model") [24]. It describes modularity as a "quality criteria" associated with the "quality factors" that included maintainability, testability, and interoperability. At about the same time, Boehm et al. proposed their model of software quality ("Boehm's Quality Model") [3, 4]. This model does not mention modularity. There are internationals standards on software quality, including ISO 9126, which does not mention modularity, and ISO 25010, which has modularity as a sub-characteristic of maintainability. The inconsistency between different quality models makes it difficult to determine exactly what is meant by such attributes as modularity.

There are a variety of definitions of modularity. ISO 25010 defines modularity as "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" [13, 4.2.7.1]. Booch defines it as "the property of a system that has been decomposed into a set of cohesive and loosely-coupled modules." [5, 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;" [1, p334]. Pfleeger offers "In a modular design, the components have clearly defined inputs and outputs and each component has a clearly stated purpose." [31, p207].

The difficulty is, none of these provide a direct means to examine a design and determine its modularity. The common theme in all the discussions we have seen is one of "independence" of modules, and that is what we have adopted. However, it cannot be that all modules are completely independent. To provide the requisite functionality, there must be some interaction between modules. Furthermore, the nature of the interaction bears on the degree of independence. Martin argues that depending on abstract modules is preferable to depending on concrete modules [22] (and see also the Dependency Inversion Principle from SOLID [23]). So assessing modularity is difficult. We will address this point further in section 3.

There has been a great deal of research on how to teach object-oriented programming (OOP). Some of the discussion has been on "objects-early" versus "objects-late" (e.g. see Bruce for a summary [6]). Some have focused on language issues (e.g. [19]) or environment support (e.g. [20]), and based on this experience new languages have been design to support teaching OOP [2]. Issues relating to general concepts in object-oriented programming continue to create challenges for teachers [33], as do how they manifest in particular languages [25]. The computer science education community continues to seek more effective methods to teach these concepts [26].

While understanding the concepts associated with object-oriented programming and associated languages is important, creating a good design is more than that. Parnas' discussion was independent of the language used, and in fact was not in terms of object-oriented concepts. There has been comparatively little research in teaching good object-oriented design. There is evidence that students struggle to develop modular designs. Cai et al. conducted a study in which students were given detailed object-oriented designs in UML [7]. Despite the design having good modularity, the implementations developed by the students largely did not. Of the 85 usable student submissions, 74% contained dependencies inconsistent with good modularity. The authors used design structure matrices (DSMs) to assess modularity. A DSM provide avisualisation of the dependency structure of a design. It can be used to indicate dependencies that are considered inappropriate for good modularisation.

In later work, Cai et al. speculated that the difficulty students face is that the real benefits of modular designs is in their evolution[8]. As students rarely revisit code they write for assignments, they do not experience the consequences of design decisions they have made. They proposed the use of a tool that provided direct feedback on DSMs. The authors hypothesised that having such a tool would enable students to produce more modular designs. While the results were not significant, they did suggest that even with the tool, capable students still introduced unnecessary dependencies in their implementations.

The research described above is interesting in two ways. The first is, students struggled to produce modular *implementations* despite having been given modular *designs*. This suggests that students would find it even more difficult if they have to come up with the designs as well. Second, the feedback given to the students was in terms of *dependencies*. While quality attributes such as modifiability and understandability are likely affected by the dependencies that exist in a design, assessing dependencies is nevertheless an indirect view of modifiability and understandability. In contrast, in this study, we directly assess modifiability and understandability of software designs with varying levels of modularity.

Controlled experiments are frequently used in software engineering research. Many studies use students as some or all of the participants, as it is often easier (and cheaper) to recruit them [9, 34]. Most such studies are using students as proxies for all software developers, rather than the students being the target population. This has raised external validity concerns with such studies, however at least in some cases students seem to be representative of all developers [32].

Use of controlled experiments in computer science education is also fairly common. Most focus on determining whether an intervention helps with learning (e.g. [11]).

There have been studies examining the impact of how some characteristic of software design impacts developer performance. For example Sjøberg et al. hired professional developers to perform maintenance tasks on four functionally-equivalent implementations with different code smells [35]. To our knowledge there have

been no studies, whether on software developers in general, or students in particular, that directly compare how modularity affects understandability and modifiability.

Understanding the quality of code also includes understanding the effect of code design on developers. Psychological constructs have been used to better understand how developers work with code. A "mental representation" refers to how developers represent their understanding of the code base [29], for example, how a developer understands the behaviour of the specific object, method or function of the software system. During programming tasks, developers accumulate transient representations, referred to "Task Knowledge" [30], which they maintain long enough to complete the task [29]. Information foraging has been used to describe how programmers navigate when they modify and debug code [21], where the programmer is the predator, the software bug is the prey, the scents are any cues that may lead to the bug. Code design may influence the salience of scent information.

Cognitive load has been used for decades to understand individual differences in task performance. Cognitive load describes the interaction between task demands and the person's capabilities, and can be thought of as a multidimensional construct including mental load, mental effort, and performance [27]. These and related emotional constructs have been associated with software engineering. Graziotin et al. found a relationship between unhappiness, productivity and performance [16]. Cognitive depletion may result from being "overloaded" from acute workload and extended engagement [14]. Cognitively depleted individuals may abandon their tasks [14]. The NASA task load index (NASA TLX) introduced in 1998 [18], is a subjective, multidimensional assessment tool to assess perceived mental workload. The NASA-TLX has become widespread internationally as a standard in government, industry and academic research [17]. The NASA TLX is divided into six sub-scales: mental demand, physical demand, temporal demand, performance, effort and frustration. Thus, in attempting to understand the effect of modularity on developers, we seek to understand performance as well as multidimensional subjective effort.

3 METHODOLOGY

Our overall goal is to better understand how students respond to modularity in order to develop better ways to teach this concept. Modularity is presented as having benefits to software quality, in particular modifiability and understandability. Thus, we ask: *in a software engineering student population*:

(1) does modularity impact modifiability of code?
(2) does modularity impact understandability of code?

We use a controlled experiment to answer these questions. Our design is a between-subjects experiment. Our participants were asked to perform the same modification task on a design, with each participant receiving either a "low modularity" or a "high modularity" design. That is, "modularity" is the independent variable. The dependent variables are the correctness of the result from completing the task and a rating by participants of how understandable they perceived the design they worked with.

Participants completed the experiment in an office room at our organisation. There were two desks with two standard computers. The desks faced toward different walls of the room. There was a maximum of two participants that could partake in the study for every session. The methods used in this research were approved by the ethics committee at our institution.

3.1 Participants

Forty computer science or software engineering students completed the experiment. Participants were recruited from students enrolled in the 4th year of a software engineering program or enrolled in a postgraduate computer science program. 10 participants were women, and were near evenly distributed across conditions, with four women who were assigned to the low modularity and six assigned to the high modularity condition.

3.2 Code Modification Task

We selected two designs to use for the Modification task, one from the high modularity group and one from the low modularity group (Figures 2 and 1 respectively). These were selected to reflect high or low modularity for the particular modification task.

Both designs provided the same functionality, confirmed through unit tests. The functionality was to process a file containing metadata about classes, interfaces, enums, and annotations (collectively referred to as "modules") in a Java project. The result was a report showing the dependencies between modules in the alphabetical order of their fully-qualified names, i.e., ordered based on the packages a module is located in.

The programs used in this study were implementations of a system typical for a first assignment in a introductory course on object-oriented programming. This meant it was small enough to be done in a reasonable amount of time but complex enough to allow students to make use of features typically found in an object-oriented language. The programs used were in fact prior submissions by students in such a course and were implemented in Java.

The system is meant to provide different forms of analysis of a data file. It takes the name of the file with the data and a *query* as input, and it produces output corresponding to applying the query to data in the file. The details of the data and the possible queries are not important here, except that the data includes names of Java modules (including the package the modules belong to) and one query produced a list of *fully qualified names* in alphabetical order. So, for the input `x.y.A`, `m.n.M`, and `x.m.B`, the output would be (on separate lines):

```
m.n.M
x.m.B
x.y.A
```

The task the participants had to perform was to change the order that these names were displayed to be in alphabetical order of the *simple name* (the name of the module without the package name). For the example, this order would be:

```
x.y.A
x.m.B
m.n.M
```

As discussed in Section 2, it is difficult to reliably assess the modularity of a design. As we noted there, a common theme in

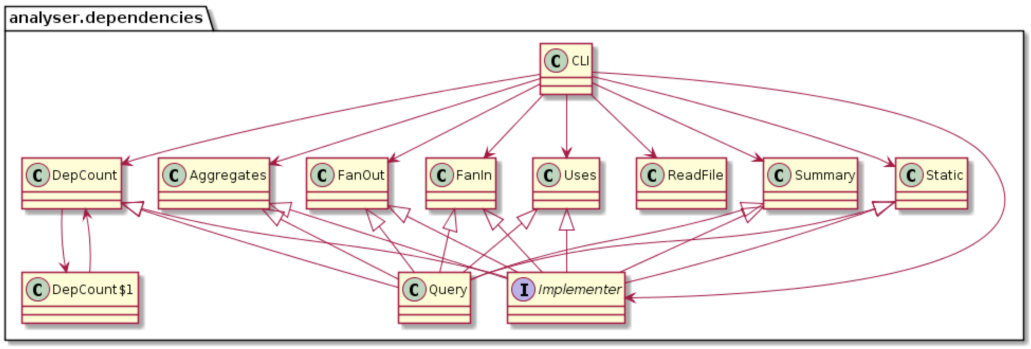

Figure 1: UML class diagram of design with low modularity.

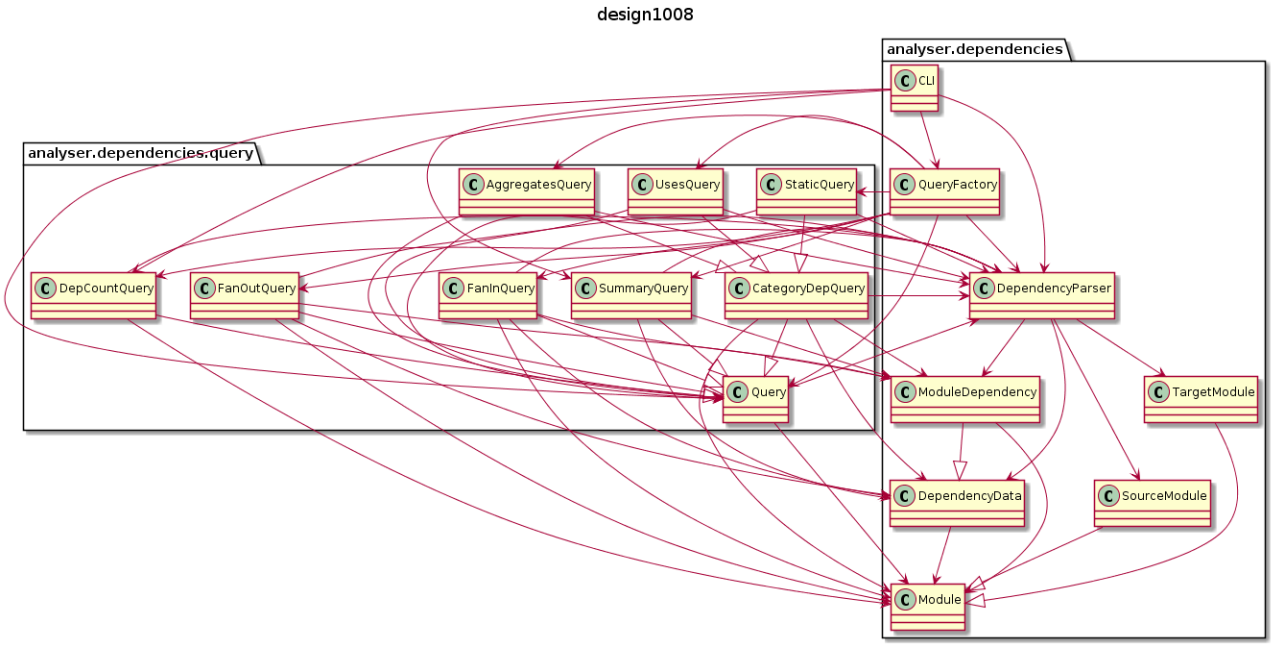

Figure 2: UML class diagram of design with high modularity.

discussions on modularity is independence between modules. However this alone is not a sufficient basis to assess modularity, as there has to be some interaction between modules in order for the requisite functionality to be provided. We resolved this in two ways. First, we only need to know the *relative* modularity of two designs, that is, which is "more modular." Second, we assessed a design with respect to the task we wanted participants to perform.

Measurement is ultimately about comparison, in this case comparing the modularity of two designs. If we had a reliable means to measure modularity, we could do so with the two designs and use the measurements to determine which is "more modular." However all measurement does is provide a means to determine the *empirical relationship* of two things (Fenton and Pfleeger, referring to the *representation condition* [12]). Measurement is not required to determine this—it can be determine through direct observation. We provide the details of this direct observation below.

Discussions regarding modularity often focus on the benefits of having good modularity, such as resulting in "easily maintained, easily tested, easily reused" [1, p334]. Such benefits apply to specific tasks. It is therefore reasonable that some tasks may benefit more than others, and so we reason that it is appropriate to compare the modularity of two designs with respect to a specific task.

Based on the rationale above, we chose the two designs for our study based on how independent the relevant components in the designs were with respect to the task, in this case changing the order that results are displayed.

The two designs chosen are shown in Figures 2 and 1. Figure 2 shows the design with "high" modularity, at least with respect

to the ordering of classes, referred to as "modules" in the requirements specification. It does so by representing declaring an abstract class `Module` that implements the `Comparable` interface. Doing so consists of a 1-line implementation of the `compareTo()` method. This means that how modules are ordered is determined by this one method.

Figure 1 shows the design with "low" modularity. It uses a 10-line implementation of the `Comparator` interface that is an inner class implemented as a parameter in a method invocation and in a method that is 65 lines long. We argue that because this implementation is an actual parameter to a method call contained in another method means the code that supports ordering of modules in this design is less independent than the equivalent in Figure 2, and so the modularity is lower.

There are clearly other differences between the two designs. The low modularity design appears simpler, with fewer modules with few dependencies and a single level of inheritance. The high modularity design has many more dependencies (not just due to the larger number of modules), and more levels of inheritances (e.g. `UsesQuery` extends `CategoryDepQuery`, which in turn, extends `Query`). As we have no way to reliably and objectively assess the modularity of a design ([8]), we had to rely on our manual, and possibly subjective, assessment of the modularity of the two designs only with respect to the task. We discuss this further in Section 5.1.

In both cases, the modification task requires finding the relevant sections of code and making the changes. Our reasoning was that, in the case of the high modularity design, the fact that `Module` implemented the `Comparable` interface was very visible (being near the top of the file) and the code that needs to be changed was very short (one line). For the low modularity design, we felt that the fact that the relevant code is buried in the middle of a long method would make it difficult to find. Further, as the code is much longer (10 lines), it would be more difficult to change.

Participants completed the task using Eclipse Photon edition, which is an Integrated Development Environment (IDE). Through this IDE, the participants could verify the functionality of their modification through a JUnit Test Suite that was provided. The JUnit Test Suite consisted of seven JUnit Test Cases that checked the output of the Java program for the given inputs.

Participants had access to coding resources. In the introduction of the code base and modification task, participants were given a link to a Java reference tutorial with the exact text "Optional resource: https://docs.oracle.com/javase/tutorial/".

A system-level logger developed in C# was created to track activities during sessions. In the instructions, participants were informed that they had 30 minutes to complete the task. Our system displayed a pop-up notifications on the amount of time left at the 20, 28 and 30-minute marks.

3.3 Dependent Variables

A key outcome for this research is on whether participants were able to complete the modification task in the allotted time period. We term whether participants completed the task as "functional correctness". To do so, every time a participant executed the JUnit Test Suite, the time and the number of passed and failed test cases were logged. The logs of the JUnit Test Suite were used to determine if the participant's modification has "passed" or "failed" the programming test cases, i.e., was the participant able to complete the requested modification. This variable was then converted into a binary variable of "passed" or "did not pass". We also assessed understanding and perceived task difficulty.

We administered four questions to test participants' understanding of the code base:

(1) Which class(es) are responsible for displaying the output?
(2) Which method(s) contains the implementation for displaying the output?
(3) Which class(es) are responsible for ordering the output?
(4) Which method(s) contain the implementation for ordering the output?

Participants' answers to these questions were manually assessed for correctness by two honours software engineering students, who also created model answers for each question. Participants were given a score of 1 for each correctly answered question, and a score of 0 for incorrectly answered questions. If the participant's response did not match the model answers, their response may still be marked correct because they may have modified their code as for the answer to be correct. For each answer that did not match the model answer, we checked whether the participants' responses were correctly aligned with their revised code. If the answer was aligned with their revised code we marked it as correct. The correctness of the answers could be determined objectively—either the participants provided the name of the correct class/method or they did not. If there was any uncertainty, the assessors consulted each other and reached a consensus.

The NASA TLX [18] was used to assess the perceived workload for the code modification task. Scales were shown as continuous sliders that registered 21 values between the anchors "very low" and "very high". Participants completed the scale at the end of the experiment.

4 RESULTS

In this section we present differences in performance, understanding and self-reported mental load.

To investigate modifiability, we conducted a χ^2 (Chi-square) test, a non-parametric test designed to analyse group differences, with condition and functional correctness. We observed that achieving functional correctness was significantly associated with modularity, where those in the high modularity condition achieved a higher rate of success compared to those in the low modularity condition $\chi^2(1, N = 40)$ p=.038 (Figure 3).

To investigate understandability, we conducted a t-test with condition and the number of questions that were answered correctly. We observed a statistical trend between score and modularity, where those in the high modularity condition exhibited poorer understanding compared to those in the low modularity condition (mean low modularity = 3, mean high modularity = 2.4, p=.07).

In the low modularity condition, 11 participants were able to achieve fully accurate understanding (all questions correct), whereas only 3 of those also achieved functional completeness. In the high modularity condition, only 7 participants were able to achieve full understanding, and all of them achieved functional completeness.

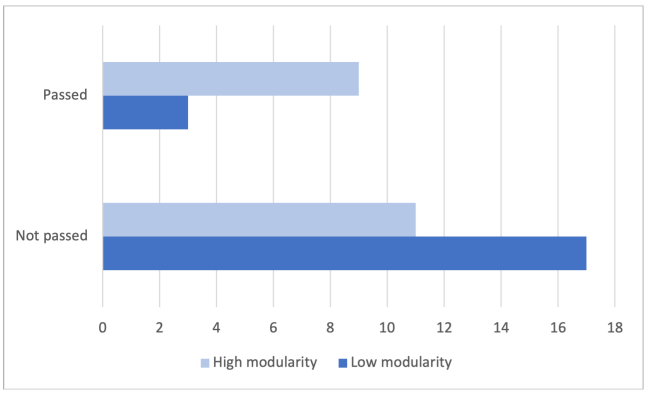

Figure 3: Number of participants whose modification passed or failed test cases, i.e., who achieved functional correctness, in the high or low modularity conditions. Nine out of twenty participants in the high modularity condition were able to complete the modification compared to only three out of twenty participants in the low modularity condition.

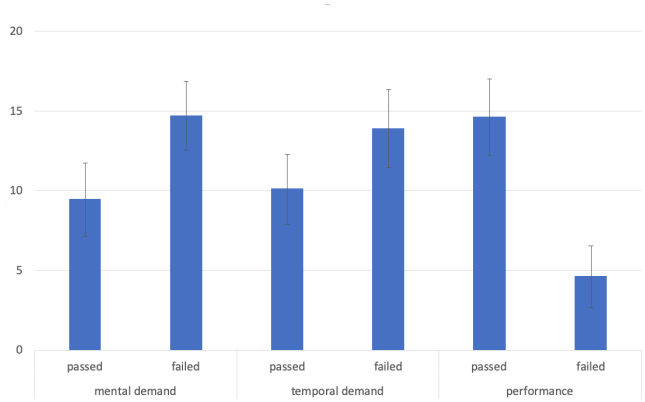

Figure 4: Average NASA TLX scores and standard deviations for participants whose modification passed or failed test cases, on a 21 point scale. Participants who were able to complete the modification task rated the task's mental demand and temporal demands as lower and rated their performance as higher.

In the high modularity condition, there were two participants who achieved functional completeness with incomplete understanding (one student with 0 questions correct, and another with 2/4 questions correct).

We conducted a MANCOVA on the effects of high or low modularity on the NASA-TLX indices including mental demand, temporal demand, performance, effort, and frustration, and we included the functional correctness pass/fail as a covariate. We observed a significant effect for functional correctness (p=.002), suggesting that the experience of either completing or not completing the task was more salient to perceived load than the code design. We further investigate the univariate analyses and found that functional correctness had a significant effects on mental demand ($F[1, 37]=8.262$, $p=.007$; , temporal demand($F[1, 37]=4.993$, $p=.032$; , and performance($F[1, 37]=23.193$, $p<.001$ (Figure 4).

5 DISCUSSION

Modularity has been referred to for decades with variations on the construct and a gap in evidence on the effects on programmers. We conducted an experiment to investigate the causal impact of modularity on software engineering performance. The results indicate that designs with high modularity code enable participants to achieve successful modification compared to designs with low modularity. However, there was a trend where participants' understanding of the high modularity code was lower than of the low modularity code. Our results indicate that modularity may present a tension between understandability and modifiability. Understandability of code with high modularity may require the ability for abstraction.

Modularity affected the complexity of the modification task. It is fascinating that high modularity led to more successful modification as well as to a poorer understanding of the code. The most was found by those who understood the high modularity design as they were all able to modify it successfully. It is interesting that another 2 participants were able able to successfully modify the high modularity condition without a fully accurate understanding. In the low modularity condition, the correct understanding did not transfer to being able to successfully modify the code.

We expect that the results for modifiability will hold with developers but we anticipate that results for understanding may change with experience. In our sample with students, modularity affected their ability to quickly develop a good understanding of the code. In the case of the low modularity design, there were fewer classes and methods, hence the task may be inherently easier for the simple reason that there are fewer things to choose from. In the case of low modularity, students did not have to understand relationships between abstractions. We believe that more expertise and experience may lead to more ease in understanding relationships between abstractions. Specifically, more experienced developers may be more familiar with abstracts, and hence hold more mental schemas of abstractions, meaning that perceiving cues for abstraction can then match a mental model of how the sections of code fit together.

Our research has implications for curriculum design. For example, students are rarely presented with examples of the same program with designs that vary in modularity. In our experience of presenting such exemplars, most students are able to rank the designs based on modularity, but a portion are not able to. When asked for explanations on better or worse modularity, students explain their choices with justifications that are unrelated to modularity. Students are also rarely given the opportunity to experience the benefits of modularity. One of the main benefits of modularity occurs when code must be modified periodically. Students' assignments are typically 'one-off's' where they never revisit code produced earlier in their education. This type of revisiting is difficult to include in curriculum because it means that each student would be starting off at a different point, introducing inequalities in the outcomes.

5.1 Threats to Validity

One potential threat is confounding factors. Distractions can impact participant performance, and so we tried to provide a relatively quiet space for them to work in. Familiarity with the technologies and tools used in another possible confounding factor. The experimental artefacts were written in Java and the IDE used was Eclipse, which the participants should be familiar since they are used in both our Computer Science and Software Engineering programmes. Nevertheless, there could be other confounding factors that have not been considered in our analysis.

The modularity of the artefacts was assessed by one of the researchers, with the rationale given above. It is possible there are other characteristics of these artefacts that affect participants' performance. For example, the low modularity design seems simpler, and so may be easier to work with. In fact this characteristic strengthens our result. Participants were more successful for the apparently more complex, but more modular, design.

Because the high modularity design has more modules than the low modularity design, it is possible those participants with the low modularity condition had a higher chance of guessing answers to the questions correctly. Since the lower modularity design had 11 classes, the chance of guessing correctly would still be quite low, and so we believe that the level of modularity better explains our results than does number of classes.

We used task success as our measure of modifiability. Where this choice may not be appropriate is if someone achieved success but took a significant amount of time to complete the tasks. As we had a limit on the time available for the task, we believe this is not a factor. We used the correctness of the answers to the questions to measure the level of understanding. We acknowledge that the questions were at quite a high level, and so may not reflect the participants' true understanding.

All of our participants were in undergraduate programmes with a strong programming component (e.g. Computer Science, Software Engineering). This may limit the generalisability of our results. However our programmes are developed according to international curricula, and so we believe our results are generally applicable. Future research can replicate our study to validate whether our results generalise outside of our organisation.

There is the question as to whether the difference in the two designs are really differences in modularity or due to some other quality attribute. Given the lack of agreement in proposed definition of modularity, it is possible that others would disagree with our assessment. Nevertheless, we believe the differences can be best explained in terms of difference in modularity.

Finally, we performed a single study with only two different designs, so we cannot claim that our results generalise beyond our study. To generalise further would require many more studies. Our results suggest that such studies are justified.

6 CONCLUSION

To investigate how modularity impacts how students modify and understand code, we conducted an experiment where 40 computer science or software engineering students worked through a 30-minute Java task. Participants were randomly allocated to modify a design high in modularity or low in modularity. We found that those in the high modularity condition where better able to successfully modify the code. We found a trend in the opposite direction for understanding, where students in the high modularity condition tended to have worse understanding of the code.

Our results have implications for how modularity is taught in tertiary curriculum. In order to demonstrate the benefit of modularity on modifiability, we should expose students to this through appropriate activities.

In science, one study provides only an indication of what may be true. More evidence is needed. We call upon the Computer Science Education community to conduct more such students to confirm (or refute) our results.

ACKNOWLEDGMENTS

The authors would like to those who participated in this study. We also thank the reviewers for their very useful feedback.

REFERENCES

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

[2] Andrew P. Black, Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. Seeking Grace: A New Object-Oriented Language for Novices. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 129–134. https://doi.org/10.1145/2445196.2445240

[3] Barry W Boehm, John R Brown, and Myron Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*. 592–605.

[4] B. W. Boehm, J. R. Brown, M. Lipow, G. J. MacLeod, and M. J. Merritt. 1978. *Characteristics of Software Quality.* Elsevier North-Holland.

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

[6] Kim B. Bruce. 2004. Controversy on How to Teach CS 1: A Discussion on the SIGCSE-Members Mailing List. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (Leeds, United Kingdom) *(ITiCSE-WGR '04)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/1044550.1041652

[7] Yuanfang Cai, Daniel Iannuzzi, and Sunny Wong. 2011. Leveraging design structure matrices in software design education. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE T)*. 179–188. https://doi.org/10.1109/CSEET.2011.5876085

[8] Yuanfang Cai, Rick Kazman, Ciera Jaspan, and Jonathan Aldrich. 2013. Introducing tool-supported architecture review into software design education. In *2013 26th International Conference on Software Engineering Education and Training (CSEET)*. 70–79. https://doi.org/10.1109/CSEET.2013.6595238

[9] Marian Daun, Carolin Hübscher, and Thorsten Weyer. 2017. Controlled Experiments with Student Participants in Software Engineering: Preliminary Results from a Systematic Mapping Study. arXiv:1708.04662 [cs.SE]

[10] Pablo Anderson de L. Lima, Gustavo da C. C. Franco Fraga, Eudisley G. dos Anjos, and Danielle Rousy D. da Silva. 2015. Systematic Mapping Studies in Modularity in IT Courses. In *Computational Science and Its Applications – ICCSA 2015*, Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Marina L. Gavrilova, Ana Maria Alves Coutinho Rocha, Carmelo Torre, David Taniar, and Bernady O. Apduhan (Eds.). Springer International Publishing, Cham, 132–146.

[11] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. *On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors.* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3411764.3445696

[12] Norman E Fenton and Shari L Pfleeger. 1998. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA. http://portal.acm.org/citation.cfm?id=580949

[13] International Organization for Standardization. [n. d.]. 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.

[14] Lyndsey Franklin, Kristina Lerman, and Nathan Hodas. 2017. Will Break for Productivity: Generalized Symptoms of Cognitive Depletion. *arXiv preprint arXiv:1706.01521* (2017).

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

[16] Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. 2017. Unhappy developers: Bad for themselves, bad for process, and bad for software product. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 362–364.
[17] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50. Sage Publications Sage CA: Los Angeles, CA, 904–908.
[18] Sandra G Hart and Lowell E Staveland. 1988. *Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research*. Human Mental Workload, Vol. 52. North-Holland, Amsterdam ;. 139–183 pages.
[19] Michael Kölling. 1999. The Problem of Teaching Object-Oriented Programming, Part I: Languages. *JOOP* 11 (04 1999), 8–15.
[20] M. Kolling. 1999. The problem of teaching object-oriented programming, Part 2: Environments. *Journal of Object-Oriented Programming* 11, 9 (1999), 6–12. http://bluej.kent.ac.uk/papers/1999-09-JOOP2-environments.pdf
[21] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. 2010. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 39, 2 (2010), 197–215.
[22] Robert C. Martin. 1995. Object Oriented Design Quality Metrics: an analysis of dependencies. *C++ Report* (Sep/Oct 1995).
[23] Robert C. Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
[24] J. McCall, P. A. Richards, and Gene F. Walters. 1977. Factors in software quality: concept and definitions of software quality. *Nat'l Tech. Information Service* 1, 2, 3 (1977).
[25] Craig S. Miller and Amber Settle. 2016. Some Trouble with Transparency: An Analysis of Student Errors with Object-Oriented Python. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 133–141. https://doi.org/10.1145/2960310.2960327
[26] Nathan Mills, Allen Wang, and Nasser Giacaman. 2021. Visual Analogy for Understanding Polymorphism Types. In *ACE '21: 23rd Australasian Computing Education Conference, Auckland, New Zealand (and virtually), 2-5February, 2021*, Claudia Szabo and Judy Sheard (Eds.). ACM, 48–57. https://doi.org/10.1145/3441636.3442304
[27] Fred GWC Paas and Jeroen JG Van Merriënboer. 1994. Instructional control of cognitive load in the training of complex cognitive tasks. *Educational psychology review* 6, 4 (1994), 351–371.
[28] D. 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
[29] Chris Parnin. 2010. A cognitive neuroscience perspective on memory for programming tasks. *Programming Interest Group* (2010), 27.
[30] Chris Parnin and Spencer Rugaber. 2011. Resumption strategies for interrupted programming tasks. *Software Quality Journal* 19, 1 (2011), 5–34.
[31] Shari L Pfleeger. 1998. *Software Engineering: Theory and Practice*. Prentice Hall.
[32] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 666–676.
[33] R. Shmallo and N. Ragonis. 2021. Understanding the "this" reference in object oriented programming: Misconceptions, conceptions, and teaching recommendations. *Educ Inf Technol* (2021). https://doi.org/10.1007/s10639-020-10265-6
[34] Dag Ingar Kondrup Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. 2005. A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering* 31, 9 (September 2005), 733–753. https://doi.ieeecomputersociety.org/10.1109/TSE.2005.97
[35] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering* 39, 8 (Aug. 2013), 1144–1156. https://doi.org/10.1109/TSE.2012.89

7 APPENDIX: TASK DESCRIPTION PROVIDED TO PARTICIPANTS

The Java program analyses data sets and displays a set of results after analysing the data. This program processes a file containing data about dependencies in code. The program will display a list of modules that match certain queries. In Eclipse, you'll find the source code of one design. Currently, the program displays a list of modules in the alphabetical order of their fully-qualified names (i.e., full file name including directory). For example, for modules x.y.A, m.n.M, x.m.B, the current output is:
m.n.M
x.m.B
x.y.A

Your task is to modify the code so that the modules are displayed in the alphabetical order of their simple class names (i.e., last segment of file name). For example, for modules x.y.A, m.n.M, x.m.B, the modified output would be:
x.y.A
x.m.B
m.n.M

Optional resource: https://docs.oracle.com/javase/tutorial/