SIGCSE: G: Empirical Assessment of Software Documentation Strategies: A Randomized Controlled Trial

Scott Kolodziej
Texas A&M University
College Station, Texas
scottk@tamu.edu

ABSTRACT

Source code documentation is an important part of teaching students how to be effective programmers. But what evidence do we have to support what good documentation looks like? This study utilizes a randomized controlled trial to experimentally compare several different types of documentation, including traditional comments, self-documenting naming, and an automatic documentation generator. The results of this experiment show that the relationship between documentation and source code understanding is more complex than simply "more is better," and poorly documented code may even lead to a more correct understanding of the source code.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Documentation**; *Software development techniques*; Automatic programming;

KEYWORDS

Software documentation, empirical study, randomized controlled trial, automatic documentation generation, self-documenting code

ACM Reference Format:
Scott Kolodziej. 2019. SIGCSE: G: Empirical Assessment of Software Documentation Strategies: A Randomized Controlled Trial. In *Proceedings of ACM Student Research Competition Grand Finals.* ACM, New York, NY, USA, 5 pages.

1 PROBLEM AND MOTIVATION

When discussing source code documentation, a question invariably arises: What does good documentation look like? High-quality comments and good variable names are standard recommendations, but what experimental evidence exists to support this standard?

Historically, technical limitations in programming languages such as line length limits have restricted the amount and type of in-source documentation available. However, as these limitations have been removed, the available options for documentation have rapidly increased and diversified. Coding standards frequently describe how source code should be documented [2, 5–7], and popular reference books provide detailed arguments for and against certain forms of documentation [9, 10].

ACM Student Research Competition Grand Finals, 2019,

Anecdote, case studies, and thought experiments are often provided as evidence in these standards and references, and recent advances in data mining source code repositories can provide hints to good programming practices. However, well-designed experiments may provide significant insight and further test these hypotheses and practices [3, 19]. This randomized controlled trial compares several different documentation styles under laboratory conditions to determine their effect on a programmer's ability to understand what a program does.

2 BACKGROUND AND RELATED WORK

Several previous studies have examined the effects of software documentation on program comprehension, but few have focused specifically on source code comments and variable names rather than external documentation formats. Fewer still can be classified as controlled experiments [15]. One experiment by Sheppard et al. found that descriptive variable names had no statistically significant impact on the ability of a programmer to recreate functionally equivalent code from memory [14]. However, the same study commented that as participants wrote their functionally equivalent code, they generally used meaningful variable names, even when rewriting source code that had less meaningful names.

Another study by Woodfield et al. demonstrated the efficacy of comments when interpreting source code written in Fortran 77 [22], with similar studies conducted by Tenny using PL/1 [18] and Takang et al. using Modula 2 [17] showing similar trends. While these studies lend strong evidence to the claim that comments improve comprehension of source code, their validity may not be generalizable to today's modern programming languages.

More recently, Endrikat et al. conducted a controlled experiment to study the effect of static typing and external (i.e. not in-source) documentation of an existing API had on development time [4]. While the beneficial effect of static typing over dynamic typing was statistically significant and substantial, the effect of additional API documentation was not statistically significant. A similar study on software maintenance by Tryggeseth showed that external documentation of a large codebase with no existing comments decreased the time programmers took to modify the system by 21.5%, supporting the efficacy of external documentation in the absence of comments [20].

Developers tend to be more willing to change code than to change accompanying comments, which can lead to misleading documentation and future bugs [8]. This may imply that self-documenting code is a better documentation strategy than comments alone. An observational study based on open-source software repositories indicated that more verbose comments may result in more change-prone code, again challenging the efficacy of comments [1].

Automatic documentation generation is a recent development that uses machine learning to document code based on established examples of well-documented code [16]. The resulting documentation often uses a mixture of variable renaming and additional comments. One such generator, JSNice, uses this approach to generate documentation for source code written in JavaScript [12].

In this work, we compare the effect of comments, self-documenting naming, and automatic documentation generation in a modern programming language (C++). Specifically, we measure the effects of these documentation styles on programmer response time and accuracy when determining what a short code snippet does. Additionally, this study is an instance of a randomized controlled trial, meaning that the experimental categories are randomly assigned to avoid bias and a control category (representing no documentation) is present.

3 APPROACH AND UNIQUENESS

In this experiment, three forms of documentation were compared:

- *Commenting*, non-functioning descriptive text in the source code (see Listing 1).
- *Self-documenting naming*, descriptively naming variables and functions to imply further documentation unnecessary (see Listing 2).
- *Automatic generation*, using a machine learning-based tool to generate both comments and descriptive names (see Listing 3).

```
// Sum all elements in the array a
double s = 0;
int n = a.size(); // length of the array
for (int i = 0; i < n; i++) {
  s += a[i];
}

// Compute the arithmetic mean
double x = s / n;
```

Listing 1: Example of Commented Code

```
double arraySum = 0;
int numElements = array.size();
for (int index = 0; index < numElements; index++) {
  arraySum += array[index];
}

double mean = arraySum / numElements;
```

Listing 2: Example of Self-Documenting Code

```
/** @type {number} */
double s = 0;
int sl = a.size();
/** @type {number} */
int i = 0;
for (; i < sl; i++) {
    s = s + a[i];
}
/** @type {number} */
double v1 = s / sl;
```

Listing 3: Example of Generated Documentation

Because commenting and self-documenting naming are not mutually exclusive, five experimental categories were created:

(A) *No comments with poor naming*. This category served as the control. No comments were provided, and all variable and function names were random alphabetic characters.
(B) *No comments with self-documenting naming*. No comments were provided, but variables and functions were descriptively named.
(C) *Descriptive comments with poor naming*. Comments were included in the source code, but variables and functions were named with random alphabetic characters.
(D) *Descriptive comments with self-documenting naming*. Both comments and descriptive names were included, otherwise referred to as *full documentation*.
(E) *Automatic generation*. Comments and names generated using JSNice.

3.1 Experimental Design and Methods

Five code snippets, ranging between 23 and 96 lines of code, were written in C++ and documented in each of the five styles. The snippets were created with the goal of being representative samples of software source code, with each snippet including 1-2 concepts selected from object-oriented programming, algorithms, data structures, and recursion.

As no C++-based documentation generator currently exists, the JavaScript-based JSNice was used [12]. Thus, the code snippets were translated from C++ to equivalent JavaScript, stripped of all forms of documentation, submitted to JSNice, and the resulting source code was translated back to C++.

Twenty undergraduate students majoring in computer science or computer engineering were recruited who had completed an introductory course in C++. After informed consent, each participant was given a brief demographic survey (to account for programming experience and year in major), a practice question to familiarize themselves with the user interface, and a test to assess their understanding of the documented code snippets. Participants were asked to respond to three multiple-choice or fill-in-the-blank questions per snippet, with the code still visible.

To avoid experimental bias, the test contained all five code snippets in a randomized order and randomized documentation style such that each participant was presented with all five snippets and all five documentation styles. This allowed each participant to serve as their own control. This randomization was accomplished using a 5×5 Latin square experimental design (see Figure 1).

The experiment was conducted under controlled conditions with privacy dividers and no outside assistance. Pen and paper was provided for all participants, and the test was administered on a laptop with a wireless mouse. While no time limit was imposed, all participants completed the experiment within one hour.

4 RESULTS AND CONTRIBUTIONS

The results of the experiment demonstrate that several trade-offs may exist in choosing the most effective documentation strategy. Figure 2 shows the effect of the different documentation types on the response time of the participant, while Figure 3 shows the effect of the documentation types on the correctness of the participant's response. Generally, the addition of comments or self-documenting naming decreases response time compared to the control group,

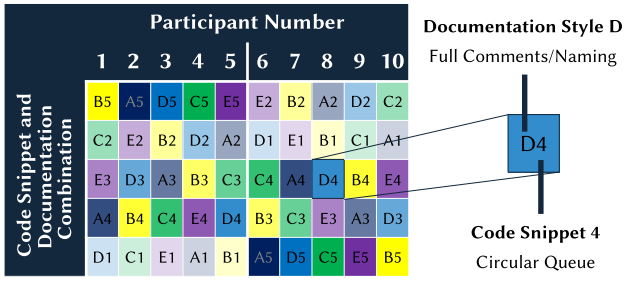

Figure 1: Latin Square Experimental Design. Four 5×5 Latin squares (two shown) were used to randomize the assignment of documentation/snippet permutations to participants. Each participant was presented with one of each of the five snippets, and one of each of the five documentation styles. The order of the snippets was randomized every five participants.

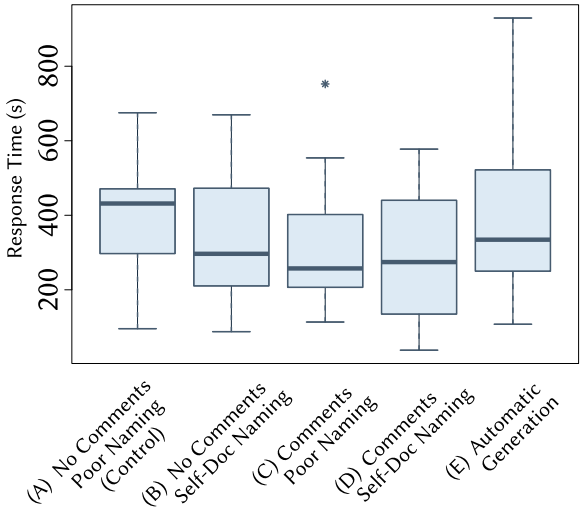

Figure 2: Effect of Documentation Style on Response Time. Raw data box plots showing median, interquartile range, and outliers of the response times for each documentation style.

but also decreases response accuracy. Automatically generated documentation yields results similar to no documentation at all.

4.1 Statistical Analyses

Additional statistical analysis using generalized linear models allowed us to control for several factors:

- *Code snippet.* Our analysis controlled for certain snippets being inherently more or less difficult to understand.
- *Participant.* We controlled for variability in each participant's inherent ability, including their average response time and accuracy.
- *Time.* When analyzing response accuracy, responses were binned into two minute intervals, allowing us to consider correctness given an approximately constant response time.

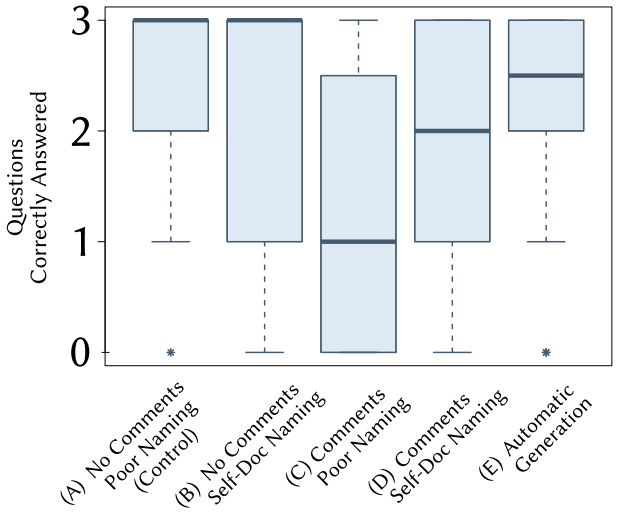

Figure 3: Effect of Documentation Style on Response Accuracy. Raw data box plots showing median, interquartile range, and outliers of the response accuracy for each documentation style.

- *Accuracy.* When analyzing response time, responses were binned by the number of questions correctly answered, allowing us to consider response time given a constant response accuracy.
- *Snippet and Documentation Style Ordering.* While these factors were thoroughly randomized, we also determined that there were no substantial learning effects or attrition over the duration of the test.

To determine which documentation styles are statistically different from one another, we used Tukey's honest significant difference (HSD) test to compare all pairs of means [21]. For the response time model, we applied a square root transformation to response time, as residual variability increased at higher values of time (i.e. most participants finished in approximately the same amount of time, but a sizable fraction took significantly longer to complete the test). For the response accuracy model, we used a generalized linear model with a binomial link function to account for the discrete response variable (correct or incorrect for each question). These results are reported as an odds ratio, e.g. documentation style A has 1.3 times higher odds of eliciting a correct response than documentation style B. A threshold of $\alpha = 0.05$ was used for determining statistical significance.

As the documentation styles form two orthogonal factors when neglecting the automatic generation case, the categories can be grouped as follows:

- All cases without comments (A $\cup$ B) against all cases with comments (C $\cup$ D).
- All cases without self-documenting naming (A $\cup$ C) against all cases with self-documenting naming (B $\cup$ D).

This grouping allows us to increase our statistical power, as we are only comparing two cases with approximately twice as many data points in each category.

An additional analysis was used to determine which documentation styles are statistically indistinguishable, in contrast to the previous analyses to determine which documentation styles are

Documentation Category Pair	$\Delta\sqrt{t}$	95% Confidence Interval	Approx. Δt	p-value
A – B	1.43	[-1.29, 4.16]	50s	0.582
A – C	2.09	[-0.60, 4.78]	75s	0.201
A – D	3.17	[0.52, 5.83]	117s	**0.011***
A – E	0.14	[-2.51, 2.80]	5s	1.000
B – C	0.65	[-2.07, 3.38]	22s	0.962
B – D	1.74	[-0.95, 4.43]	62s	0.376
E – B	1.29	[-1.40, 3.98]	45s	0.665
C – D	1.09	[-1.57, 3.74]	38s	0.781
E – C	1.94	[-0.71, 4.60]	69s	0.253
E – D	3.03	[0.41, 5.65]	111s	**0.015***
(A ∪ B) – (C ∪ D)	0.22	[1.54, 0.15]	7s	0.987
(A ∪ C) – (B ∪ D)	3.17	[1.43, 2.22]	117s	0.059

Table 1: Statistics for Response Time Analysis. The leftmost column describes the two documentation styles being compared, with the last two rows comparing two groups of documentation styles. The first category listed in each pair corresponds to a longer response time. The difference between means is reported in square root time, along with the 95% confidence interval for the difference between the square root means. An approximate untransformed value based on the minimum mean time of 284.6 seconds is listed, along with a p-value for the square root means being the same ($H_0 : \mu_A = \mu_B$), so a statistically significantly small p-value of $p < 0.05$ implies that the means are different.

Documentation Category Pair	Odds Ratio	p-value
A / B	1.30	0.995
A / C	10.03	**0.011***
A / D	4.09	0.164
A / E	3.14	0.468
B / C	7.71	**0.005****
B / D	3.14	0.257
B / E	2.41	0.561
C / D	2.45	0.435
C / E	3.19	0.215
E / D	1.30	0.990
(A ∪ B) / (C ∪ D)	31.52	**0.0004*****
(B ∪ D) / (A ∪ C)	1.88	0.669

Table 2: Statistics for Response Accuracy. The leftmost column describes the two documentation styles being compared, with the last two rows comparing two groups of documentation styles. The first category listed in each pair has better odds of eliciting a correct answer, and the odds ratio is reported in the middle column. The p-value for equal odds ($H_0 : \mu_A = \mu_B$) is given in the last column; a small p-value of $p < 0.05$ implies that the odds of eliciting a correct answer are statistically significantly different.

statistically *different*. When considering response time, a threshold value of $\epsilon = 20$ seconds was used (i.e. controlled response time *averages* that are within 20 seconds of each other are considered the same). However, the results of this analysis indicated that none of the documentation styles were statistically indistinguishable at this threshold.

4.2 Conclusions

The numerical results of these analyses for both response time and response accuracy are shown in Tables 1 and 2, respectively. The key conclusions and associated p-values from the analysis are listed as follows:

(1) The no documentation (control) style (A) has 10 times higher odds of eliciting a correct answer than (C) comments alone ($p = 0.011$), even when controlling for response time. When grouped together, all cases without comments (A ∪ B) have 31 times higher odds of eliciting a correct answer than all cases with comments (C ∪ D), with a p-value of $p = 0.0004$.

One interpretation of this result is that documentation can provide a false sense of comprehension on the part of the programmer, leading them to believe that they understand the code through the documentation without taking time to truly determine how the program functions. It should also be noted that the documentation was not meant to be misleading in any way, and that this phenomenon was visible across all code snippets.

This result calls into question traditional wisdom regarding the benefits of comments as documentation [11], but it cannot be ignored given both its magnitude and strong statistical significance. Further study to determine under what conditions comments are most beneficial (or detrimental) is warranted. If substantiated, this trend could potentially be exploited for use in debugging: an editor could hide comments to help a programmer focus entirely on the mechanics of the code and improve their comprehension of it.

(2) Descriptive naming alone (B) has almost 8 times higher odds of eliciting a correct answer than comments alone (C) after controlling for response time ($p = 0.005$).

This result most directly answers the question whether descriptive, "self-documenting" naming is more or less effective than comments: indeed, naming was the more beneficial factor in our analysis, at least with respect to correctness of understanding.

(3) The no documentation (A) and automatically generated (E) documentation cases take substantially more time (mean times >100 seconds apart) to interpret than full documentation (D) after controlling for response accuracy ($p = 0.011$ and $p = 0.016$, respectively).

While these two categories of documentation yielded more correct answers, the time participants took to read and interpret the code was disproportionately greater. This suggests that participants resort to deciphering and mentally executing the code itself in the absence of documentation.

(4) Documentation styles that elicit more correct responses generally take longer to interpret; the inverse is also true. In other words, there is a weak inverse relationship between response time and accuracy.

This is a surprising result, as one would expect documentation styles that are difficult to interpret or unhelpful would lead to both increased response times and fewer correct responses, with the inverse true for more helpful documentation styles. Instead, our data suggest that a trade-off exists, and that a lack of good in-source documentation may lead to a better understanding of the functionality of the code at the cost of time.

4.3 Threats to External Validity

As with any empirical study, we list below several reasons the results of this study may not generalize to other contexts without further study or verification:

- *Student developers versus professionals.* The experiment sampled exclusively from a student population, which may or may not generalize well to professional developers [13].
- *Programming language-specific effects.* This experiment was conducted in C++ as undergraduate computing majors at Texas A&M University are required to complete an introductory course in C++. It is unknown whether these results will generalize to other programming languages. It is interesting to note that our results are substantially different from previous studies done in Fortran and PL/I [18, 22], but the precise cause for this difference warrants further investigation.
- *Code snippets versus large codebases.* Each code snippet in this experiment was an isolated and self-contained program, but whether these results will generalize to larger codebases is unknown.

4.4 Future Directions

Moving forward, replication of this experiment via a second trial of similar size could help to greatly increase the statistical significance of the results and further define the effects of each documentation strategy relative to one another. Additionally, further studies are planned to help better understand the detrimental effect that comments had on participant accuracy and distinguish it from scenarios where comments are more beneficial. For example, previous studies have suggested that comments may be beneficial for software maintenance tasks [14], but also that they risk becoming outdated in that same context [1, 8]. The location and type of comments can also vary greatly, with several broad categories including header comments (describing what an entire class or function does), multi-line block comments, and single line comments. In our study and in the majority of the existing literature, no firm distinction is made regarding the type of comments used, so future work in this area to differentiate the efficacy of each would be beneficial.

REFERENCES

[1] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2017. *Empirical Analysis of Words in Comments Written for Java Methods.* IEEE. 375–379 pages. https://doi.org/10.1109/SEAA.2017.23

[2] Apple, Inc. 2018. WebKit Code Style Guidelines. Retrieved from https://webkit.org/code-style-guidelines/. Accessed: 2019-04-14.

[3] Victor R Basili. 2007. The role of controlled experiments in software engineering research. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions.* Springer, 33–37.

[4] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How do API documentation and static typing affect API usability?. In *Proceedings of the 36th International Conference on Software Engineering.* ACM, 632–642.

[5] Free Software Foundation, Inc. 1992. GNU Coding Standards. Retrieved from https://www.gnu.org/prep/standards/standards.html. Accessed: 2019-04-14.

[6] Google. 2019. Google C++ Style Guide. Retrieved from http://google.github.io/styleguide/cppguide.html. Accessed: 2019-04-14.

[7] Google. 2019. Google Developer Documentation Style Guide. Retrieved from https://developers.google.com/style/. Accessed: 2019-04-14.

[8] Walid M Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software* 85, 10 (2012), 2293–2304.

[9] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship.* Pearson Education.

[10] Steve McConnell. 2004. *Code Complete.* Pearson Education.

[11] Jef Raskin. 2005. Comments are more important than code. *Queue* 3, 2 (2005), 64–65.

[12] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 111–124.

[13] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 666–676.

[14] S.B. Sheppard, Curtis, Milliman, and Love. 1979. Modern Coding Practices and Programmer Performance. *Computer* 12, 12 (1979), 41–49. https://doi.org/10.1109/MC.1979.1658575

[15] D. I. K. Sjoberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. . Liborg, and A. C. Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering* 31, 9 (Sep. 2005), 733–753. https://doi.org/10.1109/TSE.2005.97

[16] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10).* ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/1858996.1859006

[17] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.* 4, 3 (1996), 143–167. http://compscinet.dcs.kcl.ac.uk/JP/jp040302.abs.html

[18] Ted Tenny. 1988. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1271–1279.

[19] Walter F Tichy. 1998. Should computer scientists experiment more? *Computer* 31, 5 (1998), 32–40.

[20] Eirik Tryggeseth. 1997. Report from an experiment: Impact of documentation on maintenance. *Empirical Software Engineering* 2, 2 (1997), 201–207.

[21] John W. Tukey. 1949. Comparing Individual Means in the Analysis of Variance. *Biometrics* 5, 2 (1949), 99–114. http://www.jstor.org/stable/3001913

[22] Scott N Woodfield, Hubert E Dunsmore, and Vincent Yun Shen. 1981. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering.* IEEE Press, 215–223.