Reflective Statement

This essay was definitely much harder than the previous two, mainly because it needed to be concise yet detailed, and these ideas are almost opposite. I needed to write specific but not wordy, make my essay readable but not over-explain, successfully convey a very complicated idea. I tried to write as directly as I could, yet I still wrote over 3000 words. This is very hard because it's a "balancing act" — you can go too far in one direction, then with a slight modification, go too far in the other direction. I still believe that often, it *is* possible to write about a complicated idea relatively clearly (on the other hand, it's also possible to write about a simple idea very obscurely). However, even after finishing the essay, I'm still not sure I've done so.

The Unit 2 memo really helped me create this essay. When I initially started thinking about the essay, before I wrote the memo, I had a hard time choosing a specific topic. I initially chose a topic which was very broad and complicated: proposing a large, detailed experiment which would test many programming paradigms against many types of programs. This would have been much harder to explain concisely — it would've had much more details, yet also would've been less specific. Even I wasn't really sure how the experiment would actually work, and I also wasn't convinced it was a good idea myself. Fortunately, while writing the memo, I found a much better topic: determining memory management's effect on code clarity. This topic is much more specific (one paradigm, against one concept), and over time, I *did* form a clear idea how the experiment would run, and why my audience would want to run it. Still, explaining these ideas was a challenge, and while doing so, I realized the topic isn't as simple as I hoped. In particular, I realized that the *real* reason that the experiment is important is because of *partial* memory management's effect on code clarity — while this still falls under "memory management's effect on code clarity", so I referred to it as such, I had to make sure my essay focused on the "partial" aspect, particularly when it explained its motive. But overall, I believe this topic was a relatively good choice, much better than the other topics I had in mind. My Unit 2 memo was how I settled on this topic, so looking back, although I'm not sure it needed to be an entire unit, I'm glad it was its own assignment.

Unlike my Unit 1 essay, I significantly changed this essay after my initial draft. But also, unlike my Unit 2 essay, I didn't need to start from scratch. When I wrote the draft, I already had my topic in mind, from Unit 2. However, I didn't have much time to write the draft, and it turned out harder than I expected. I did manage to completely finish every section, so when I submitted the draft I had a "full" essay. But I didn't spend much time on the sections, and some of them needed significant work. In particular, I wrote the conclusion right before the draft was due, and that showed — I didn't go "above and beyond" in any way, and I ended up completely rewriting my conclusion in the final paper.

I worked on my draft over spring break, so I made a lot of changes even before I got my revisions. I quickly realized that my organization was bad — in particular, I put my experiment right in the middle of the essay. So I significantly re-organized the sections, split paragraphs, and added more headers. I also tried to make my essay much less wordy, looked for more sources to back up my claims, and significantly re-

worked my introduction and background to make my proposal and motive clearer. When I got my peer reviews and instructor feedback, they showed that I needed to make many changes from my initial draft. Fortunately, I already made some of those changes, such as the re-organization and better introduction. But there were other changes I didn't notice. Most importantly, I over-explained a lot — the instructor feedback *and both* peer reviews mentioned this. I honestly didn't realize how much I over-explained until I looked at the feedback, mainly because I was worried that with such advanced material, I would make my essay too confusing. However, once I got this feedback, I realized some of my paragraphs *did* explain ideas that a scholar should know about. In particular, the paragraph where I explained "failure is a necessary part of research" was completely unnecessary, any researcher should know that, so I removed it. Overall, I believe I applied most of the general feedback, although some feedback I had already applied in prior revisions (such as the re-organization), and other feedback was no longer relevant because of those revisions (mostly minor grammar corrections).

One big change I made was choosing a more specific audience — Winthrop High School. While none of my feedback directly suggested changing the audience, my instructor feedback in particular signaled some problems. I found that I had trouble specifically speaking to and appealing to my audience (in ways I wouldn't speak to other audiences), and appealing to their specific wants and needs. I figured this was because my audience was too broad — I was trying to speak to *all* college professors. I realized that if I made my audience more narrow, I could better speak to them specifically, because they would have clearer, more precise wants and needs. Furthermore, I realized that Winthrop High School had some really unique reasons how it would benefit from the experiment. I won't go into too much detail, because I mention these reasons in my actual paper. But, for example, the school really is trying to modernize itself, and my experiment uses a lot of new technology, so it would help achieve this goal. There were many schools I could've chose from, particularly Northeastern. But I know Winthrop High School really well, and because of their unique position, they would benefit from this experiment more than the rest. (Northeastern, in particular, would've been a bad audience because we already have a well-established computer science program).

Overall, this essay took a lot of effort, but I believe it will help me in the future. This was one of my first experiences writing to a scholarly audience. But I doubt it will be the last, because I'm currently doing a research co-op, and plan to go to graduate school in the future. In fact, I've submitted an abstract and poster to RISE, and I believe this essay even helped me write those (although RISE doesn't seem as scholarly as other poster sessions — I believe it's supposed to be an "introductory" session designed to ease students into future research). Similarly, this was an experience where I needed to *read* from a scholarly audience — I needed to read papers to find sources. This was *not* my first experience reading scholarly papers, but if I'm going to graduate school, I'll need to read many more papers and find many more sources. Furthermore, this is a persuasive essay, and I believe I have a hard time persuading others. Particularly important, this was an essay where I try to persuade *professional scholars*, people who are "higher-up", and I'll need to persuade my bosses and managers in the future. In my case, I might even need to persuade my general audience — instructors — because if I'm doing research, a professor will be my boss (advisor). In fact, while I don't think I'll

make a proposal for this particular experiment, I'll probably need to make proposals to instructors in the future, and perhaps as part of my research, I might also investigate memory management's effect on code clarity. Ultimately, this essay was not as interesting as I had hoped, but I stayed motivated writing it, because I know that it will significantly help me in the future.

Context Note

This document is a proposal for a specific experiment, which would help determine memory management's effect on code clarity. The document's audience consists of the math teachers and upper-level faculty of Winthrop High School, a school in my hometown which I attended. I would distribute the document by e-mailing or directly handing it to them. The purpose is to convince the faculty to run the experiment, and describe the experiment with an outline, so they ultimately do perform it.

While a computer runs a program, it needs to keep track of which data is allocated (used) and which is free (unused), so it can reuse the space of the free memory. This is called memory management. Some programming languages do this completely automatically (with "garbage collectors"), while some require the developer to write code and keep track of the data themselves. It's easier for developers to understand "clear" code and figure out what it does. If memory management clarifies code, that means it makes the other parts of the code (like what it does) easier to figure out — if it obscures code, that means it makes them harder.

Proposal for an Experiment to Investigate (Partial) Memory Management's Effect on Code Clarity

For Winthrop High School

Overview

Memory management's effect on code clarity, particularly in high-level languages, is an area which should be investigated more. This proposal presents one possible investigation: a specific experiment which would replace Winthrop High School's introductory programming class. The first section covers why memory management's effect should be investigated. The second section covers how memory management's effect should be investigated. The third section explains how the investigation would particularly benefit Winthrop High School. The fourth section is the conclusion, it summarizes the above sections. A detailed outline of the experiment is provided in the appendix.

1. Why investigate?

There is not much research on how memory management affects code clarity, particularly "partial" memory management implemented in recent high-level languages. As of now, many professionals believe that manual memory management obscures code, and most papers simply assume it. However, there is a possibility that some forms of memory management clarify code. Ultimately, the experiment proposed here is significant because it would test this possibility.

1.1. Why memory management seems to obscure code

In the past, manual memory management was only implemented in "low-level" languages, like C and Fortran. In these languages, incorrect memory management code causes "memory corruption" errors which are extremely hard to track down, as well as exploits such as the LBNL Traceroute Heap Corruption Vulnerability [11]. Practically all "high-level" languages, such as Java and Haskell, used to have garbage collectors. Because the programs with memory management had many other downsides, memory management *correlated* with obscure code.

However, recent high-level languages (like Rust and Swift) introduce "partial" memory management: the developer must write some explicit memory-managing code, but the compiler handles the rest. Furthermore, these languages have checks, so that memory management bugs are significantly easier to track down (in particular, Rust will highlight these bugs at compile-time) [5, 15].

Unfortunately, there is not much research on how "partial" memory management affects code clarity. Articles which reference memory management mostly conclude that it obscures code, but as their evidence, they use the older languages and deficiencies which partial memory management has addressed. For example, "A Large Scale Study of Programming Languages and Code Quality in Github" [6] analyzes how many factors affect code clarity, including memory management, by determining whether languages with each factor lead to more bugs. It categorizes each language based on its features — functional vs. procedural paradigm, static vs. dynamic compilation, strong vs. weak typing, and (automatic) managed vs. (explicit) unmanaged memory. However, it only

uses C, C++, and Objective C to determine explicit memory management's effect, and as a result, the only category with unmanaged memory also has a procedural paradigm, static compilation, and weak typing. Similarly, "Automatic vs. Explicit Memory Management: Settling the Performance Debate," [7] states that "Garbage collection frees the programmer from the burden of memory management and improves modularity, while preventing accidental memory overwrites (“dangling pointers”) and security violations [36, 45]." However, Rust has an integrated package manager which encourages modularity, and a compiler which essentially prevents memory overwrites and security violations [5].

1.2.How memory management could clarify code

There is reason to believe the hypothesis that some aspects of memory management, when implemented in a high-level language, actually clarify code. These aspects seem to model more general concepts, which when revealed by explicit memory management, could help a developer see and modify their program's structure. For example, Rust's "borrowing" could model linear types — another article, "System Programming in Rust: Beyond Safety" [5], has also speculated this. Furthermore, allocation could model relevance (allocated objects are relevant, while freed objects are "forgotten"), and references could model dependencies (when one object has a strong reference to another, it depends on the other) or parent-child relationships (parents have strong references to children, but children have weak references to parents).

Research on general design patterns and code clarity, in particular "The role of language paradigms in teaching programming" [12], support this hypothesis. Explicit and implicit memory managing can be considered programming paradigms, as they are both different ways to express a program with the same semantics. In this paper, four academics each present an example of other paradigms and concepts, then describe their effects:

- Joe Armstrong states that the real world is concurrent, and procedural paradigms do not model concurrency well, so they do not produce clear code in programs which simulate or interact with the real world. Similarly, memory management might produce clearer code in programs which require what it models (linear types, relevance, dependencies, or parent-child relationships; a concrete example of the latter is management software for a hierarchical company).
- Boris Magnusson states that the object-oriented paradigm is a particularly good first paradigm for computer science students to learn, because it teaches them how to structure programs. Similarly, memory management might also teach students structure, by teaching them the other high-level concepts mentioned above.
- Matthew Flatt argues that even non-developers could benefit by learning how to program, because programming as a whole teaches more general concepts which apply outside of computers. Similarly, even if students never use memory management in their jobs (because of languages with garbage collectors), they could still benefit by learning the more general concepts which memory management models.

- Lastly, Peter Van Roy, the moderator, summarizes that there is no "ideal" paradigm, each paradigm has benefits and drawbacks. Memory management (even partial) does have drawbacks, but perhaps it has benefits as well.

Everything stated above in this section is speculation — there is no proof that memory management reveals more general concepts, or that those concepts clarify code. But there is no proof otherwise, and that is why memory management's effect on code clarity should be investigated.

2. How to investigate

2.1. General structure of the investigation

The investigation should be very similar to previous investigations on code clarity. Any new, unique investigation is risky and not credible. However, this especially applies to an investigation on partial memory management's effect on code clarity, because there are no prior results to reinforce the findings.

In particular, "Fundamental concepts of CS1: procedural vs. object oriented paradigm – a case study" [14], provides a good foundation for this investigation to be derived from. The study takes an existing course, divides it into two sections, teaches each section according to a different paradigm, gives both sections the same exams, then compares the exam averages to see what type of reasoning each paradigm supports. The case study produced results: both groups performed the same in all questions, except the object-oriented group performed significantly better in a question involving reasoning about ADTs. These results seem accurate, because they align with the commonly-held belief that object-oriented programming helps developers model structures [12]. Furthermore, these results must have been meaningful, because the study got published by the ACM, a prestigious organization. The study might have even influenced the college which performed it, as they stopped teaching procedural programming afterward.

2.2. Key details of the investigation

While the case study above is a good starting point for an investigation into memory management, there are a few key details which should be different. These details will make the experiment easier to perform, and the results more meaningful and accurate:

- Students should be assigned to both sections randomly, so that each section has a diverse, unbiased sample. The case study above did not assign students randomly, but that was simply because the college was open enrollment [14].
- The experiment should use Swift because it is good for teaching, but it also allows "optional" memory management. Swift is good for teaching because it is designed to be easy to learn, and there are existing tutorials and classes which the course can get its material from [15]. Furthermore, many companies use it and similar languages, so it will prepare the students for future careers — according to Stack Overflow's 2018 Developer Survey, 35.1% of all mobile developers use Xcode, and 8.3% of all developers use Swift [3]. At the same time, Swift allows "optional" memory management: while one is *technically* required to use its explicit memory management capabilities, in most scenarios (especially for beginners), omitting them

simply leads to wasted, excess memory (this is mentioned in the official Swift documentation [4]).

- The experiment's results should be derived from homework, quizzes and tests. The case study above only looked at a single final exam, but with multiple assignments, the results will be more accurate and present changes over time. Furthermore, the assignments should be completed in an IDE, and auto-graded with tests. The exam in the case study above was completed with pen and paper and graded manually. However, automated tests would be easier to implement, as noted in "An automatic grading scheme for simple programming exercises", especially because more assignments are being graded [8]. Moreover, multiple studies ("Computer versus paper--does it make any difference in test performance?" [10] and "Paper vs. Computer-based Exams: A Study of Errors in Recursive Binary Tree Algorithms" [13] are referenced here) have shown that when students complete assignments in an IDE, it does not significantly impact their results.

3. How Winthrop High School would benefit

Winthrop High School is a particularly good location for this experiment. Most schools do not have the resources to perform the experiment — according to Code.org, in 2018 only 35% of schools had computer science curriculums, and this is partially because of lack of funding [16]. Other schools, such as Northeastern, already have a well-established computer science program. However, Winthrop High School has only recently acquired a new building, with upgraded technology, and its "Introduction to Computer Science" class was just launched this school year. Therefore, the school has the ability to perform the experiment, without significantly altering itself or breaking long-standing traditions.

This experiment provides an opportunity for Winthrop High School to contribute novel research. There have been many studies on code clarity and how students learn to code (see [6, 9, 12, 14]), but few focus on how either is affected by memory management, and practically none include partial memory management. Therefore, this experiment will fill a gap in society's knowledge, and (even if it reinforces that memory management obscures code), it will get the school noticed. This will particularly benefit Winthrop High School, because it is underrated. It has made many improvements in recent years, but mostly due to low test scores, many still have a bad perception of it — for example, U.S. News gives it a "College Readiness Index" of just 36.3 out of 100 [1]. However, once people take a closer look at Winthrop High School, they will discover the recent improvements it has made.

This experiment will help modernize Winthrop High school and "jump start" its computer science program. Students have Chromebooks and classrooms have Smartboards, but many assignments and exams are still on paper, because most faculty simply do not know how to integrate the new technology. This experiment uses new tools like Swift and Xcode, and it is entirely digital. Furthermore, if the experiment concludes that memory management improves code clarity, contradicting prior beliefs, Winthrop High School will be the first to benefit from this knowledge. In this way, the experiment could transform Winthrop High School's computer science curriculum from a work in progress, into a solid program which stands out from the competition.

4. Conclusion

Professionals have believed that memory management obscures code ever since garbage collectors were introduced. But that could be simply because when garbage collectors were introduced, memory management was only implemented unsafely in low-level languages. High level languages with “partial” memory management are all recent, and so far, few (if any) people have really looked into how this memory management affects code clarity. Therefore, a new investigation would be informative, but it would also be risky. To make the investigation more informative and less risky, it should draw on prior similar experiments along with current practices.

Winthrop High School is transforming itself to fit into today’s computer-oriented society, with new technology. Because it uses new technology, this experiment would significantly help this transformation. It would also bring new attention to the school, so that outsiders would see the transformation's resulting progress. Few other schools are as suited to perform the experiment as Winthrop High School.

Overall, this is a major, rare opportunity. It is a great time to investigate memory management's effect on code clarity, and Winthrop High School is well-suited to perform the investigation. Therefore, even though it is a little risky, the school should take a chance and perform the experiment proposed below.

A. Experiment outline

The proposed experiment is derived from the case study suggested above: "Fundamental concepts of CS1: procedural vs. object oriented paradigm – a case study" [14]. The experiment is held in Winthrop High School's introductory computer science class, which teaches students basic computer science skills in Swift. In the experiment, the class is split into two sections, and students are randomly assigned to each.

A.1. Commonalities between the groups

Both sections learn the programming language Swift. Students participate in lectures with assigned readings, and they complete quizzes, homework, and exams (including a midterm and final). The assignments consist of simple exercises, like "write a function to compute the Fibonacci number". Students complete them on a computer with Xcode, and each exercise is a "Playground" file, which contains the assignment's instructions, and (for some assignments) starter code like helper functions. Students can run their code as many times as they want (within a time limit for quizzes and exams), and they can (and should be encouraged to) write their own tests before they submit them, but no tests are ever provided to the student.

The material is derived from *Beginning Swift Programming* [14], except for one important detail: for most of the exercises, a technically correct solution involves using memory management "hints", like weak references, but a solution without any hints would also run, it would simply waste memory. The tests for grading the assignments should require different hints so that they do not leak memory — some tests should not leak memory even without any hints, (if possible) some should only require particular hints, and some should require the program to be completely leak-proof.

When the students submit the assignments, a computer will auto-grade them by running the code against a series of tests (not provided to the student), designed to cover all cases. Each test is an expression which, when appended to the student's code, should evaluate to true. Instructors will briefly look over the assignments to disqualify cheaters, but besides that, the grading will be entirely based on the tests. Instructors can officially (on transcripts) grade students however they want, but each grade measured in the experiment's analysis is the number of passing tests, divided by the number of total tests.

A.2. Differences between the groups

One section (the experimental group) explicitly teaches memory management in the lectures and readings. The other section (the control group) omits the memory management lectures and readings, and replaces them with more review. Both sections complete most of the same assignments. However, the memory management section completes a few assignments which *require* memory management hints (they semantically do not work without them), while the other section instead completes extra review assignments.

Additionally, even for the same assignments, the auto-grader for the memory management section not only checks that each test evaluates to true, it also checks that afterward, there are no leftover allocations. If there are any, the test counts as half credit to the student (e.g. on their transcript), although it still counts as *full* credit in the final analysis.

A.3. How to analyze the results

At the end of the semester, all of the grades for the quizzes and tests which both students had to complete will be analyzed. In particular, the analysis will look at the average scores among students for each question, and note any significant differences between the groups. The analysis will look at any additional "odd" statistics and try to explain them. However, this outline explicitly omits more details on how the data will be analyzed, because there are many possible trends, and many ways which one could interpret them.

Works Cited

[1] "Winthrop Sr High in Winthrop, MA." U.S. News. https://www.usnews.com/education/best-high-schools/massachusetts/districts/winthrop/winthrop-sr-high-9568 (accessed March 16, 2019, 2019).

[2] "2018 State of Computer Science Education," 2018. Accessed: March 17, 2019. [Online]. Available: https://advocacy.code.org

[3] "Automatic Reference Counting — The Swift Programming Language (Swift 5)." Apple Inc. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html (accessed March 16, 2019, 2019).

[4] "Stack Overflow Developer Survey 2018." Stack Overflow. https://insights.stackoverflow.com/survey/2018/#technology (accessed March 17, 2019, 2019).

[5] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari, and L. Ryzhyk, "System Programming in Rust: Beyond Safety," *SIGOPS Oper. Syst. Rev.,* vol. 51, no. 1, pp. 94-99, 2017, doi: 10.1145/3139645.3139660.

[6] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in GitHub," *Commun. ACM,* vol. 60, no. 10, pp. 91-100, 2017, doi: 10.1145/3126905.

[7] M. Hertz and E. D. Berger, "Automatic vs. Explicit Memory Management: Settling the Performance Debate," ed. Vancouver, BC, Canada: OOPSLA, ACM, 2004.

[8] J. B. Hext and J. W. Winings, "An automatic grading scheme for simple programming exercises," *Commun. ACM,* vol. 12, no. 5, pp. 272-275, 1969, doi: 10.1145/362946.362981.

[9] J.-P. Kr, #228, mer, M. Hennings, J. Brandt, and J. Borchers, "An empirical study of programming paradigms for animation," presented at the Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering, Austin, Texas, 2016.

[10] Y. Karay, S. K. Schauber, C. Stosch, and K. Schuttpelz-Brauns, "Computer versus paper--does it make any difference in test performance?," (in eng), *Teach Learn Med,* vol. 27, no. 1, pp. 57-62, 2015, doi: 10.1080/10401334.2014.979175.

[11] P. Savola. Lbnl traceroute heap corruption vulnerability. http://www.securityfocus.com/bid/1739 (accessed Mar. 9, 2019)

[12] P. V. Roy, J. Armstrong, M. Flatt, and B. Magnusson, "The role of language paradigms in teaching programming," *SIGCSE Bull.,* vol. 35, no. 1, pp. 269-270, 2003, doi: 10.1145/792548.611908.

[13] S. Grissom, L. Murphy, Ren, #233, e. McCauley, and S. Fitzgerald, "Paper vs. Computer-based Exams: A Study of Errors in Recursive Binary Tree Algorithms,"

presented at the Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, Tennessee, USA, 2016.

[14] T. Vilner, E. Zur, and J. Gal-Ezer, "Fundamental concepts of CS1: procedural vs. object oriented paradigm — a case study," *SIGCSE Bull.,* vol. 39, no. 3, pp. 171-175, 2007, doi: 10.1145/1269900.1268835.

[15] W.-M. Lee, *Beginning Swift Programming*. Wrox Press Ltd., 2014, p. 288.