Crafting Design Documents in First-Year CS Courses

Shannon Ernst
Oregon State University
ernstsh@oregonstate.edu

Jennifer Parham-Mocello
Oregon State University
parhammj@oregonstate.edu

Abstract

Computer science, at its core, is about solving problems. The "Carry out the Plan" portion of problem solving is often examined and emphasized in CS 1 and CS 2, forgetting to emphasize the other important aspects of the problem-solving process. This study focuses on the "Devise a Plan" or design step of problem solving. Four terms of design data (2,797 designs) are examined to answer the question of whether syllabus detail impacts the crafting of design documents and what are the students' attitudes toward design. The results show that syllabus detail does impact the way students design and that students do value design when asked in a survey. These insights have implications of when and how design is taught and opens questions to how best assess design.

1. Introduction

Computer science, at its core, is about solving problems. As educators in computer science, the general goal should be to teach students a systematic approach for solving problems (Raadt, 2004). Often the focus of solving the problem, especially in the initial stages of computer science, is on programming in a particular language. In early, first-year computer science classes, the emphasis is placed on how to structure the code and how to write syntax, and this emphasis largely misses the bulk of the problem-solving processes. As defined by George Polya in *How to Solve It* (1957), there are four steps used to solve problems: 1) Understand the Problem, 2) Devise a Plan, 3) Carry out the Plan, and 4) Look Back. In many freshman computer science courses, the step to "Carry Out the Plan" typically involves writing the code/program for an assignment, and the grade for the assignment is based on this step, which removes focus from the other three steps in the problem-solving process. However, one recent study used Polya's problem-solving steps to teach software development and found that students exposed to these steps were more capable to solve problems and did so in a shorter amount of time (Allison and Joo, 2014). Grounded in Polya's theoretical framework for solving problems, the action research in this paper uses student engagement and attitudes in the first-year courses to evolve a strategy and rubric for teaching the craft of creating design documents prior to coding and before learning formal software engineering practices.

2. Related Work

As Ginat et. al. point out, there is an abundant amount of literature on how students understand programming and computer models, but there are few studies on how students perform task analysis and utilize patterns (2013). The Ginat et. al. (2013) study looks at students' utilization of patterns; whereas, the study presented in this paper focuses on students' ability to recognize and perform tasks presented in a design document template and grading rubric. Instead, we are asking them to recognize the goals of the problem by listing the requirements and assumptions, as well as "put it all together" in a design plan, prior to coding and thinking about syntax. According to Soloway (1986), we are asking for the explanation using stepwise refinement, but we are also asking for students to combine these goals/plans with rules without using a specific programming language.

> *"To facilitate the transfer of knowledge from "Computer Science 100" to other problem-solving activities, students must be taught explicitly that programming is a design discipline, and as such the output of the programming process is not a program per se, but rather an artifact that performs some desired function."*

We are interested in Soloway's *plan composition*, but unlike the Fisler *et. al.* (2016) study, we are interested in the crafting of the plan composition prior to coding, rather than identifying the plan composition presented in code. At this time, we do not investigate students' plan composition used in the design documents because the focus here is on establishing a curriculum that promotes planning and design early in the first-year CS classes.

Other related work tends to revolve around a multi-institutional study published in 2004 where students were asked to design a super alarm clock (Chen et. al., 2005), (Eckerdal, 2006), (Fincher,

2004), (Thomas, 2014). The study was conducted at 21 institutions, across four countries and garnered 314 participants with half the students being early in their computer science program and the other half preparing to graduate. The designs were collected in a controlled and timed setting, and the results from these studies show that design is a skill that is acquired and improved over time. Though these studies have a wider breadth in regard to the institutions and backgrounds of the students, the research presented in this paper does not collect data in a controlled, timed environment because the authors' want to investigate the craft of creating design documents in one's own natural environment. In addition, the authors in this research study seek to teach problem-solving techniques and skills to first-year computer science students, rather than allowing them to acquire these skills. The common thread of the multi-institutional studies is they assume the design of a solution is constrained to UML diagrams or formalized notation. Whereas, this study investigates unofficial notation wherein students and professionals express their thinking in a variety of ways which can still address the key points of design and yield viable software solutions. Our research aligns a little more closely with a recent study analyzing novice student design strategies using recorded sessions of students creating design documents (Yeh, 2018); whereas, we are more interested in what students do on their own outside of a recorded environment over the course of multiple assignments.

Work published in the programming language community focuses on design recipes. These design recipes come from *How to Design Programs* and is referred to as *Program by Design* (Felleisen, 2001). Recent research used the *Program by Design* method in an introductory class and focused heavily on the link between design and code with the belief that there are patterns students should employ each time they design a function (Sperber and Crestani, 2012), (Ramsey, 2014). The work examined in the presented study does not follow a set step by step design process nor does it examine the links to code in as much depth as work found in relation to *Program by Design*. Rather, the students in the presented study are encouraged to engage in whatever form of representation best lets them learn, whether that is pictures, text or some combination. The idea behind this freedom of choice is to not bog students down with learning formal notation for design so early in the process to the point where they may not see or examine different ways of solving the problem.

There is also work on test driven design as an alternative to recipes and other systematic approaches (Janzen and Saiedian, 2008), (Proulx, 2009). It was found that students who engage in test driven design see more benefits when programming however many students are reluctant to adopt test driven design, instead preferring the test last approach (Janzen and Saiedian, 2008). In the presented study, a testing table is submitted as part of the design before the programming assignment is submitted, attempting to encourage students to reflect before and after implementation. The design in this study is not marketed under a test first policy though.

The classification system used in this study was based largely on a combination of Polya's steps and the rubric established by Thomas et al. (2014). Other rubrics to assess design quality exist, such as Castro and Fisler's SOLO Taxonomy (2017). Their rubric, though very detailed, was more closely coupled with how the code was structured and interacted rather than the designs represented in this paper which do not rely on examining the student's code structure (Castro and Fisler, 2017). Their study was also very small with only 15 participants. The presented study continues to examine the correlation between designs and grades and the inclusion of design attributes, outlined by the authors, but over a much larger data set. There are many more studies that focus on the detection of student patterns used in a solution (Ginat, 2009), (Ginat and Menashe, 2015), and use the SOLO taxonomy to classify the quality of the patterns or building blocks used by the students in an algorithmic design or actual code written in a programming language (Ginat and Menashe, 2015), (Izu et. al., 2016). However, this research does not focus on detecting patterns or classifying the learning based on these patterns. However, this research study is interested in determining a way to get first-year students to engage in multiple problem-solving steps prior to writing code, instead of skipping steps or engaging in them after code writing occurs.

3. Motivation

Systematic problem-solving prior to coding has been the primary motivation for this research, and before this study was conducted, first-year students at the host university had been required to submit

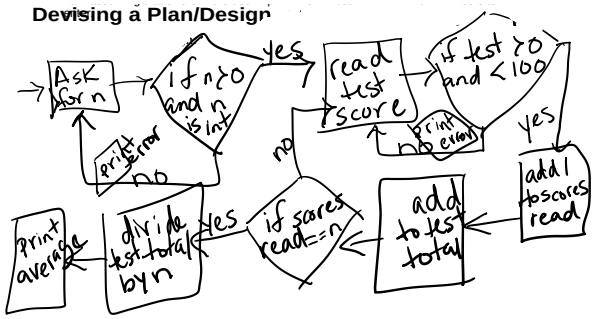
designs with their programming assignments for the past 4 years. These designs were not meant to be formal UML write ups and were not expected to be correct. The purpose of the design was to encourage students to think about the problem before they began programming the solution. At that time, students had access to a description of George Polya's problem-solving steps (see Table 1) and an example design document for what to include given a specific problem statement (see Figure 1). The example design document provided both a flowchart and pseudocode as a means for devising a plan, but the students were not required to do both. However, the students were shown that the example design document does not include code, and they were provided many more test cases to show good test coverage for good and bad test cases.

The students were reminded at the beginning of each assignment PDF to submit with their code a design which included the Understand the Problem, Devising a Plan, and Testing steps. Though data was not being collected at this point, anecdotally, teaching assistants (TAs) grading the assignment reported that students were either not submitting designs, submitting partial designs missing some of Polya's steps, or submitting low-quality designs with all the steps. It was believed that this was largely due to the low point value associated with the design as part of the assignment, as well as students writing a design after they had already programmed the assignment. Many students supported this hypothesis by admitting to their lack of engagement or creating the design document after writing the program. It was clear that students were not seeing a benefit in designing as they did not engage properly with the activity and received little enforcement or guidance on the process. This led to mandatory recitations in the first-year CS courses focusing on design concepts and practices.

Understanding the Problem:
This problem is asking me to read an unsigned whole number value, n, from the user, and then read n unsigned real numbers, which represent test scores, from the user. These scores need to be between 0 and 100, as well as a valid real number. If the user doesn't enter a valid number or a number in the range, then an error message is printed, and the user is prompted to enter a new number without taking away from the n valid numbers the user is entering. After the user enters n valid real numbers in the range of 0-100, then the average is calculated and printed to the screen.

I am assuming the number of tests is an unsigned whole number.
I am assuming the test scores can be unsigned real numbers, instead of just integers.
I am assuming that errors in the user input does not count against the n numbers to enter.

Devising a Plan/Design



Declare n
Prompt user for n
Read n from user
While (n<0) or (n not an int)
  Print error mssg.
  Promt user for n
  Read n
For n test scores
  Prompt user for score
  Read score
  While (score > 100) or (score < 0)
    Print error mssg.
    Promt user for score
    Read score
  Add score to totalscores
  Increment test scores read
Calculate average by totalscores/n
Print test average

Based on planned testing (if n ≥ 0)

don't want to divide by 0

Testing:

Value	Expected	Actual
n = 0	Nothing, just exit	Yes
n = -1	Error message and re-prompt the user for a good n value	Yes
n = 1.5	Error message and re-prompt the user for a good n value	Yes
n = 1	Prompt user for 1 test score	yes

Figure 1 - Example Design Template

Understanding the Problem	In your own words, explain what YOU think the problem is asking you to do. In this section, document your uncertainties about the problem and anything else that you feel was unclear or vague. This is to ensure that YOUR understanding matches MY understanding of the problem.

Devising a Plan/Design	At a minimum, provide an algorithm/pseudocode you designed to help solve the problem. In addition, include pictures/flowcharts you used to help you devise your plan, as well as any other design decisions you made such as how to manage your time, how to decompose the problem, where to start first, etc. You can scan any handwritten work and attach it to the document as needed.
Testing	Report any checking/self-reflection you did while solving the problem. For instance, how did you make sense of the output from the implementation? This includes things such as using a calculator to make sure the output is correct, testing to make sure your code executes correctly and behaves the way you expect under specific circumstances, using external sources of information such as the internet to make sense of the results, etc. In addition, you will provide us a test plan!

Table 1- Polya's problem-solving steps with detailed descriptions

4. Research Method

Having a recitation with the course provided a framework for discussing and enforcing design prior to coding, more feedback for design, and encouraged engagement with the activity by requiring a peer-reviewed design document submitted on Canvas (an online learning management system) one week prior to the assignment's due date. The student's recitation peer leader would grade and comment on the design to encourage the student to improve their designs over time, and students conducted peer reviews of the designs in their recitation section through Canvas to be exposed to alternative ways of thinking, to constructively provide additional comments, and to enforce taking designs seriously.

With a structured method for examining and enforcing student design in the first-year courses, CS 1 and CS 2, a study was conducted to see what students did when required to provide design documents based Polya's problem-solving steps, and how instructor guidance in a syllabus can change the behaviour of the students. The study presented in this paper examines 2,797 designs over four 10-week terms with the following research questions and motivations:

RQ1: Does the inclusion of categories differ with instructor guidance in the syllabus?

RQ2: What are student attitudes toward guidance on designs and their value of design?

4.1 Course Structure

Four first-year CS courses from spring 2016 – spring 2017 (referred to as Classes A, B, C, and D respectively) were part of this study. All courses were taught by the same instructor, providing consistency in the implementation of the recitations. The recitations count as 20% of their total course grade and have their own syllabus to provide grading requirements for submitted designs and peer reviews, in addition to the explanation of Polya's steps presented in Table 1 and the example design document shown in Figure 1 above. The recitation syllabus evolved each term of the study to address peer leader grading confusions and promote student engagement with the activity by adding increased clarification and point values to the syllabus (see Table 2). The changes are influenced by McCracken et al. (1999), who found that students do not inherently know what design is and need to be taught or given clear definitions of design. Even though the students were given a thorough explanation of what was required and an example design document, the researchers believe that increasing clarity of expectations in the recitation syllabus (or grading rubric) leads to better designs.

Class	Designs	% of Grade	Expectations of Design According to the Syllabus
Class A: Spring 2016	5	25%	"Recitation will focus heavily on design concepts. To enhance learning, every student will have to submit their design to Canvas for the current assignment during the first week of the period in which the assignment is assigned… The design does not need to be correct but show good faith effort to creating quality design for the current assignment."
Class B: Fall 2016	4	40%	"The design does not need to be correct but show good faith effort to creating quality design for the current assignment, and it MUST address 1) Understanding the Problem (2 pts), 2) Flowchart and/or Pseudocode (4 pts), and 3) Test Cases (4

			pts). You will receive one point for each area by turning in the work. The remaining points for each area will be based on how thorough and complete each section is. For example, restating the problem for the design area 1) Understanding the Problem will only get you one point. You must describe and justify your understanding of what the problem is asking to receive full credit."
Class C: Winter 2017	4	40%	"The design does not need to be correct but show good faith effort to creating quality design for the current assignment and it MUST address 1) Understanding the Problem (2 pts), 2) Flowchart and/or Pseudocode (must contain function details and header info) (4 pts), 3) Test Cases (must contain good, bad, and edges cases) (4 pts).…"
Class D: Spring 2017	5	40%	"The design does not need to be correct but show good faith effort to creating quality design for the current assignment, and it MUST address 1) Understanding the Problem (2 pts), 2) Flowchart and/or Pseudocode (must contain function details and header info) (4 pts), and 3) Test Cases (must contain good, bad, and edges) (4 pts). By default, you will receive one point for each area addressed in the design (up to 3 points for just turning in something!). The remaining points for each area will be based on how thorough and complete each section is. For example, restating the problem for the design area 1) Understanding the Problem will only get you one point. You must describe and justify your understanding of what the problem is asking you to receive full credit, i.e. both points. For test cases, you MUST have good (1 pt), bad (1 pt), and edge (1 pt) cases to receive full testing credit, and your design needs to include details for the logic in the functions (1 pt), as well as information about the pre/post conditions and return values (1 pt), and the relationship among the functions/classes (1 pt) for full design credit."

Table 2 - Recitation Syllabus/Grading Rubric Changes Over 4 Courses (A-D)

Classes A and D were required to turn in designs for 5 assignments coving the following topics.

- 2-D arrays and Files
- Classes
- Inheritance
- Polymorphism
- Linked Lists

Classes B and C were required to turn in designs for 4 assignments coving the following topics.

- Repetition
- Functions
- 1-D Arrays
- 2-D Arrays

Even though the assignment topics differ in classes A and D from B and C, the research questions in this paper are addressing the inclusion of features in the design document based on the change in instructor guidance given in the recitation syllabus.

4.2 Participants

The researchers obtained consent from students to examine their designs in 4 first-year CS courses from spring 2016 – spring 2017 (referred to as Classes A, B, C, and D respectively). Table 3 shows the consenting population differences across courses. Each course had slightly different consent rates and grade distributions. It is important to note for analysis purposes that most of the participants in this study are "above average" students.

Courses	**A**	**B**	**C**	**D**

Consent Rate	172/279 (62%)	50/120 (42%)	213/418 (51%)	177/278 (64%)
A	51% (87)	50% (25)	50% (106)	51% (90)
B	31 % (54)	22% (11)	27% (58)	24% (43)
C	13% (22)	18% (9)	14% (29)	14% (25)
D	1% (2)	6% (3)	5% (11)	5% (8)
F	4 % (7)	4% (2)	4% (9)	5% (8)

Table 3 - Demographic Details of Students in Each Class (A-D)

4.3 Classification Categories

Based on Thomas et al. (2014), the researchers developed six categories for classifying what students include in their design documents to evaluate the quality of design (see Table 4). Eighty-six random designs, 10% of the data collected from spring 2016, were used to determine a suitable classification for quantifying the design documents based on an inter-rater reliability (IRR) greater than 80%. Though the rubric proposed by Thomas et al. (2014) had six categories with 0 to 5 representing the level of quality (ranging from informal to expert), the research questions in this study can be answered using a binary value to represent if the design contains certain features. Note that *Code Present* is seen as a negative feature, while the rest of the categories are positive or neutral. Program design in the context of this study should not be code specific.

Category	Definition
Understanding the problem (UP)	Framed the design and states what the design solves for.
Relationship Among Parts (RAP)	Provided text or a picture explaining how each function and/or classes relate(s) to each other
Logic (L)	Provided details for each function and/or class, expect pseudocode or code, with an emphasis on functionality details (.h doesn't count)
Code Present (CP)	Specified specific syntax.
Diagram/Picture (DP)	Drew a diagram, picture, or flowchart to represent idea
Testing (T)	Provided a test plan

Table 4 - Design rubric with >80% inter rater reliability

5 Results

RQ1: Does the inclusion of categories differ with instructor guidance in the syllabus?

Syllabus changes seem to correlate with increased inclusion of some design components by the students. A Kruskal Wallis test was run comparing each design in each term to determine if there were differences in inclusion rate between the terms because the data was not normally distributed. A Pairwise Wilcox test was then run to determine which terms of the four differed significantly, if at all. The hypothesis for RQ1 is that increased detail in the design syllabus will lead to more inclusion of the categories.

Understanding the Problem and *Testing* were included at a higher rate each term, as shown by Figure 2. The inclusion of Understanding the Problem was found to be significantly different each term for each design with a p-value < 0.05. For testing, there were no significant differences in inclusion for the first three designs in courses A and B, but all other classes and designs did show significant differences. This can be directly linked to changes in the syllabus. Recall from Table 2 that the key syllabus change from class A to B was the enumeration of what was required in the design, namely Understanding the Problem, Flowchart and/or Pseudocode, and Testing point values and an increased overall weight of the designs to the recitation grade. The key difference between courses B, C, and D were details for the Flowchart and/or Pseudocode and Testing, which profoundly affected the rate of inclusion for these categories. These results seem to align with other research on motivating student performance using rubrics (McCracken et. al., 1999) (Jonsson and Svingby, 2007). However, there is a plateau effect in

classes C and D that is likely due to many students who take class C also take class D in their first-year, and they may already know what is expected of their designs.

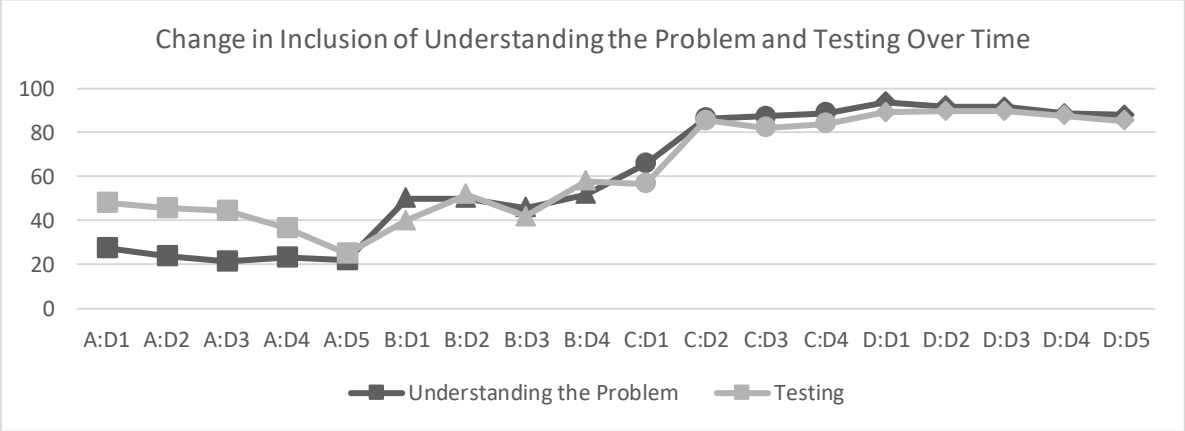


Figure 2: Change in Percent Inclusion of Categories Over Time

A place where refinement of the syllabus may still need to occur is in the *Relationship Among Parts* and the *Logic* components. There was no significant difference across the designs or terms for the inclusion of *Logic*, except for design 3 where A and B, A and C, and B and D all significantly differed from one another. The *Relationship Among Parts* category showed a variety of significant difference across the designs and terms. Figure 3 shows the percent inclusion for *Relationship Among Parts* over time. Note that even without running statistics, the graph is rather sporadic. When the Kruskal Wallis and the Pairwise Wilcox tests were run, it was discovered that there was no difference between A and B on design 1 or C and D on design 1. This could be because the design was the first one of the term and every student was starting in the same place with their understanding of the *Relationship Among Parts*. Course B design 2 was statistically different from all of the other terms. This is potentially due to the increased number of nonmajors in this course engaging with the *Relationship Among Parts*. Design 3 saw a statistically significant difference each term, starting after course B. Design 4 and design 5 saw significant differences each term.

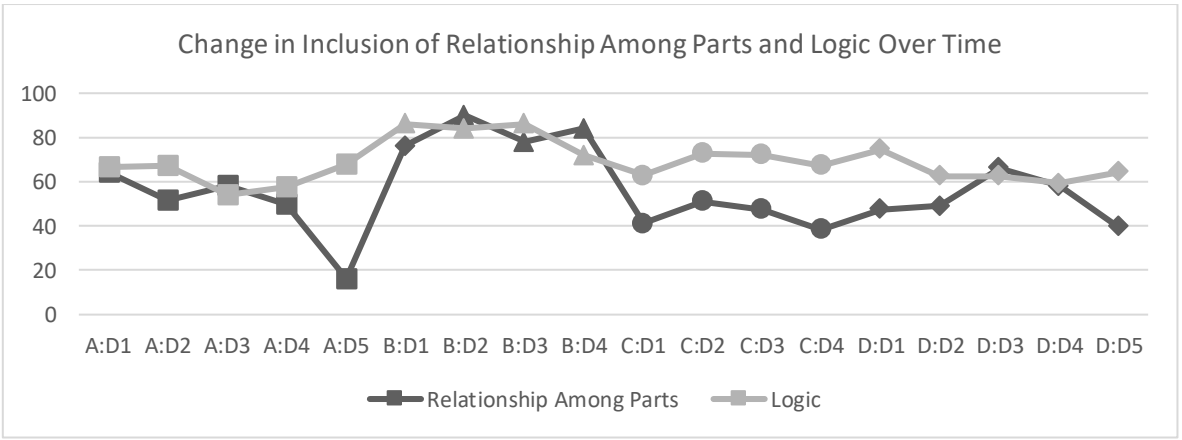


Figure 3: Change in Percent Inclusion of Categories Over Time

As seen in Figure 4, students are more likely to include diagrams and pictures versus code in a specific language. This might be due to the design document template provided in Figure 1. It is interesting to see that diagrams and pictures decrease in design 5 for class A, while the amount of code increases. In every course you see a downfall in diagrams and pictures included over time, but primarily this happens the most in the last design in courses A and D. This might be due to the topic of the assignment, which is linked lists, but this is a time when students should have an increased number of pictures in their design. However, most terms, except class A, decrease the amount of code included in their designs over time. This is encouraging, since the last assignment in class A and D are the same, and we do not continue to see the same trend in class D, after the syllabus changes.

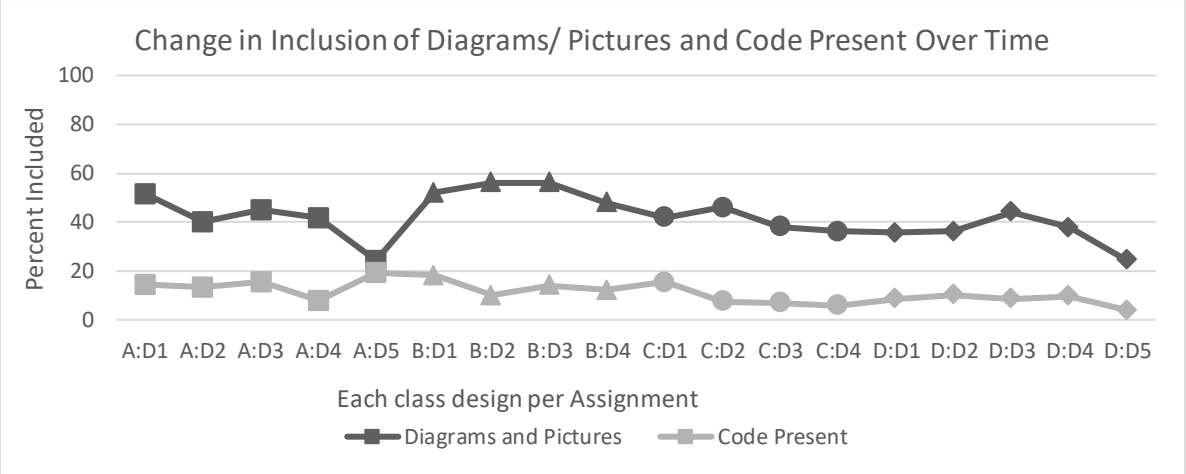


Figure 4: Change in Percent Inclusion of Categories Over Time

The fact that the differences are so sporadic over time indicates that *Relationship Among Parts* is likely not well defined for the students. They do not know how to include it or address it in their designs. This could be attributed to the overall idea of design still not being clear in the syllabus. When *Understanding the Problem* and *Testing* were both given special attention and point values, there was an observable up-tick in their inclusion rates. The design portion of the syllabus likely needs more detail or there need to be more concrete examples for the students. The request for more examples has been common from students, and depending on their recitation peer leader, they may or may not be getting the necessary support they need to design successfully in this area. The later classes, C and D, also saw an emphasis on function details and headers in the syllabus, which may have caused more students to focus on the functions rather than the relationship between the functions.

RQ2: What are student attitudes toward guidance on designs and their value of design?

In courses C and D, students were asked to fill out a survey at the end of the course speaking to their experience in recitation where the design work was conducted. In class C, 166 of the 213 consented participants responded. 67% of them wanted more guidance on design and 79% thought that creating design was useful. In class D, 160 of the 177 consented participants responded. 64% wanted more guidance on design and 86% thought creating design was useful. A Kruskal Wallis test was run to determine if there was a significant difference between these percentages. There was no difference between wanting more guidance but there was a significant difference in believing that design was useful. Many of the students who took class C proceeded into class D. The increase value of design corresponds with the established idea that students improve in design over time, likely because they engage with it more as they see value. Students wanting more guidance on the design also supports the acknowledged issue of not enough clarity in the current syllabus on design or not enough concrete examples being provided on what quality design is.

6 Conclusions and Future Work

The study sets out to show how instructor guidance can influence what students include in their design documents. Despite being given an example design document and the exact problem-solving steps to include, first-year CS students are motivated by points and specific directions. While the study is unable to present data from before the study began, the results are still valuable. It is known that before requiring design to be submitted one week prior to the coded program, most students would not engage with the activity and would submit mediocre work to get a few points. By shifting the focus of the class to prioritize 8% of the overall course grade to design, more students are engaging with the activity.

The binary scale, while useful to say if desirable artifacts are present, does not quantify the quality of the presented artifact. The *Relationship Among Parts* and *Logic* categories need a non-binary scale. The approach in Thomas et al. (2014) was too broad to achieve IRR on the designs presented in this study. The rubric may have worked better if notation was formalized. The taxonomy proposed by Castro

and Fisler (2017) was too closely tied to code structure for the researchers to use in this study; however, the level of detail and the verbiage used may be adaptable to the needs of language agnostic design. At this time, we do not investigate the quality of the design, but we plan to make correlation between the design and corresponding code quality in future studies, which will leverage work presented by Stegeman et. al. (2014). The themes discovered by Yeh (2018) study are very relative to this research and could provide a good rubric for evaluating students' strategies for devising a plan in their design documents.

The changes demonstrated in the syllabus over time lead to an increase of inclusion of certain categories which were previously being ignored. Now, most students engage with the Understanding the Problem and Reflection stages of the problem-solving process. While the impacts of this engagement are unknown at this time, future work hopes to examine more correlations in regard to code produced and time spent coding or debugging. This study also demonstrates that there is a challenge to teaching design well in the first year. However, improvements can be made between terms, and the way students value design can be changed.

7. Acknowledgements

We will insert upon acceptance.

8. References

Allison, Mark A. and Joo, Sui F. (2014). Revisiting Polya's approach to foster problem solving skill development in software engineers. 9th International Conference on Computer Science & Education (ICCSE).

Castro, Francisco Enrique Vicente and Fisler, Kathi. (2017). Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course. In Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17). ACM, New York, NY, USA, 10-19.

Chen, Tzu-Yi, Cooper, Stephen, McCartney, Robert, and Schwartzman, Leslie. (2005). The (relative) importance of software design criteria. In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '05). ACM, New York, NY, USA, 34-38.

Eckerdal, Anna, McCartney, Robert, Moström, Jan Erik, Ratcliffe, Mark, and Zander, Carol. (2006). Can graduating students design software systems? In Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE '06). ACM, New York, NY, USA, 403-407.

Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, and Krishnamurthi, Shriram. (2001). How to Design Programs: An Introduction to Programming and Computing. MIT Press, Cambridge, MA, first edition

Fincher, S., Petre, M., Tenenberg, J., K. Blaha, D. Bouvier, T. Chen, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, A. Monge, J. Mostrom, K. Powers, M. Ratcliffe, A. Robins, D. Sanders, L. Shwartzman, B. Simon, C. Stoker, A. Tew, and T. VanDeGrift. (2004). A multi-national, multi-institutional study of student-generated software designs. In A. Korhonen and L. Malmi, editors, Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education, pages 11-19, Koli, Finland, October2004.

Fisler, Kathi, Krishnamurthi, Shriram, and Siegmund, Janet. (2016). Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (SIGCSE '16). ACM, New York, NY, USA, 211-216.

Ginat, David. (2009). Interleaved pattern composition and scaffolded learning. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education* (ITiCSE '09). ACM, New York, NY, USA, 109-113.

Ginat, D., Menashe, E., and Taya, A. (2013) Novice Difficulties with Interleaved Pattern Composition. In: Diethelm I., Mittermeir R.T. (eds) Informatics in Schools. Sustainable Informatics

Education for Pupils of all Ages. ISSEP 2013. Lecture Notes in Computer Science, vol 7780. Springer, Berlin, Heidelberg

Ginat, David and Menashe, Eti. (2015). SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (SIGCSE '15). ACM, New York, NY, USA, 452-457.

Izu, Cruz, Amali, Weerasinghe, and Pope, Cheryl. (2016). A Study of Code Design Skills in Novice Programmers using the SOLO taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (ICER '16). ACM, New York, NY, USA, 251-259.

Janzen, David and Saiedian, Hossein. (2008). Test-driven learning in early programming courses. In Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08). ACM, New York, NY, USA, 532-536.

Jonsson, Anders and Svingby, Gunilla. (2007). The use of scoring rubrics: Reliability, validity and educational consequences. Educational Research Review, v2, issue 2, 130-144.

Lishinski, Alex, Yadav, Aman, Enbody, Richard, and Good, Jon. (2016). The Influence of Problem Solving Abilities on Students' Performance on Different Assessment Tasks in CS1. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 329-334.

McCracken, Michael, Newstetter, Wendy, and Chastine, Jeff. (1999). Misconceptions of designing: a descriptive study. In Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education (ITiCSE '99), Bill Manaris (Ed.). ACM, New York, NY, USA, 48-51.

Polya, G. (1957). How to Solve It: A New Aspect of Mathematical Method, Princeton, NJ: Princeton University Press,

Proulx, Viera K. (2009). Test-driven design for introductory OO programming. In Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE '09). ACM, New York, NY, USA, 138-142.

Ramsey, Norman. (2014). On teaching “how to design programs”: observations from a newcomer. In Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14). ACM, New York, NY, USA, 153-166.

Raadt, Michael de, Toleman, Mark, and Watson, Richard. (2004). Training strategic problem solvers. *SIGCSE Bull.* 36, 2 (June 2004), 48-51.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (September 1986), 850-858.

Sperber, Michael and Crestani, Marcus. (2012). Form over function: teaching beginners how to construct programs. In Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming (Scheme '12). ACM, New York, NY, USA, 81-89.

Stegeman, Martijn, Barendsen, Erik, and Smetsers, Sjaak. 2014. Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (Koli Calling '14). ACM, New York, NY, USA, 99-108.

Thomas, Lynda, Eckerdal, Anna, McCartney, Robert, Moström, Jan Erik, Sanders, Kate, and Zander, Carol. (2014). Graduating students' designs: through a phenomenographic lens. In Proceedings of the tenth annual conference on International computing education research (ICER '14). ACM, New York, NY, USA, 91-98.

Yeh, Martin K. C. (2018). Examining Novice Programmers’ Software Design Strategies through Verbal Protocol Analysis. International Journal of Engineering Education Vol. 34, No. 2(A), pp. 458–470.