An experiment on how graduating students represent software designs

Silvana Moreno[1], **Diego Vallespir**[1], **Martin Solari**[2]

[1]Universidad de la República, Uruguay

[2]Universidad ORT Uruguay

{smoreno, dvallesp}@fing.edu.uy, martin.solari@ort.edu.uy

Abstract. *As a practice, software design seeks to contribute to developing quality software. For graduating students, the design is difficult to understand, and building a good design seems to require a certain level of cognitive development. In this study, we report an experiment to know how students habitually represent detailed design (what artifacts and ways of design representation they use) and if there is an effect on software quality when they deliver the design representation. We found that students design on a basic level, they do not achieve complete designs that combine several artifacts to model dynamic and static aspects. Also, the delivery of design representation did not improve software quality.*

1. Introduction

Software design is one of the most important components to ensure the success of a software system [Hu 2013]. Software design has two main activities: architectural design and detailed design. During architectural design, high-level components are structured and identified. During detailed design, every component is specified in detail [Bourque and Fairley 2014]. Our work is explicitly focused on detailed design.

From a learning point of view, design is a complex discipline for undergraduate students to understand, and success (i.e., building a good design) seems to require a certain level of cognitive development that few students achieve [Carrington and K Kim 2003, Hu 2013, Linder et al. 2006]. In fact, several studies found that students do not manage to produce a good software design [Chen et al. 2005, Eckerdal et al. 2006a, Eckerdal et al. 2006b, Loftus et al. 2011, Tenenberg 2005].

This article presents an experiment to study how graduating students represent software designs. This experiment is a part of a family of experiments carried out in the context of the School of Engineering of Universidad de la República, in Uruguay. From 2012 to 2018, we conducted 7 experiments.

The **Baseline** experiments carried out in 2012, 2013, and 2014 were aimed at learning about the effort (measured in minutes) dedicated by students to detailed software design [Moreno and Vallespir 2018]. Results showed that students spend at least 3 times more time coding than designing. The **Template** experiments carried out in the years 2015, 2016, and 2017, were aimed at knowing about the effect on design effort and software quality when students represent the design using a specific set of design templates [Moreno et al. 2020, Moreno et al. 2021]. We found that the use of design templates did

not improve code quality and that students who use templates dedicated more significant development effort to designing than to coding.

In this article, we present the last experiment of the family. We want to study how students habitually represent detailed design (what artifacts and ways of design representation they use) and know if there is an effect on software quality when they deliver the design representation to the teachers as part of a course.

We used the term graduating for our students because they are in the fourth year of the degree of the School of Engineering. The curriculum of the School of Engineering is a five-year degree, similar to the IEEE/ACM's proposal for the Computer Science undergraduate curriculum [Joint Task Force on Computing Curricula - ACM and IEEE Computer Society 2013]. Students have already passed courses where detailed software design is taught: design principles, artifacts and design diagrams, UML, design patterns.

The document is structured as follows: Section 2 presents the categorizing student software designs; Section 3 presents the related work; Section 4 presents the research methodology; Section 5 presents the results, and Section 6 is the discussion; threats to validity are mentioned in Section 7, and Section 8 presents the conclusions and future work.

2. Categorizing Student Software Designs

Some researchers have proposed frameworks to analyze and categorize design artifacts [Eckerdal et al. 2006a, Thomas et al. 2014]. Eckerdal's category categorizes the design artifacts students produce, and Thomas' category categorizes how students understand software design. In our experiment, we used these categories, and therefore we describe them below. Both categorizations use a scalar measurement approach, from basic descriptions to more sophisticated design representation

Eckerdal et al. developed a categorization of students design artifacts of six levels [Eckerdal et al. 2006b]:

- 0N - Nothing: little or no intelligible content.
- 1R - Restatement: merely restating requirements in some fashion from the task description.
- 2S - Skumtomte: a small amount restatement of the task with a small amount of information in text, or some unimportant implementation details with no description of its design.
- 3FS - First step: some significant work beyond the description. Either a partial overview of the system with the parts identified, but generally no identification of how they are related in the system; or the design of one of the system's components.
- 4PD - Partial design: providing an understandable description of each of the parts and an overview of the system that illustrates the relationships between the (may be incomplete) parts without completely described communications between the parts.
- 5C - Complete: showing a well-developed solution, including an understandable overview, part descriptions that include responsibilities, and explicit communication between the parts.

Thomas et al. investigated students' understanding by asking them for a task to "produce a design" [Thomas et al. 2014]. The authors analyzed the designs made by students and categorized them based on the following characteristics that express the different ways in which students understand the design phenomenon:

- 0ID - Informal design: does not include formal artifacts. Generally, text but may include detailed pictures without reference to software.
- 1A - Analysis: uses some formal notation for analysis (use case diagrams/not describe system structure or behavior).
- 2SS - Static structure: focus on design techniques of software engineering, expressing the components and their structural relationships using notations like class or architecture diagrams.
- 3DB - Dynamic behavior: focus on design techniques, expressing the components and some sequential behavior using notations like sequence diagrams or flowcharts.
- 4MRA - Multiple related artifacts: use multiple artifacts and relates components across different artifacts.
- 5EC - Expert category: the notations are relaxed, and only the essential artifacts are included.

3. Related Work

Software design involves identifying and abstractly describing the software system and its relationships. It is a creative activity, which can be done in different ways: implicitly, in the developer's mind before coding, on a sketch on paper, through diagrams, using both formal and informal languages or tools [Chemuturi 2018].

Research shows that students have (still) difficulties in designing. Building good designs requires a certain level of cognitive development that few students achieve [Carrington and K Kim 2003, Hu 2013, Linder et al. 2006]. This cognitive development is related to the ability to recognize design patterns, architectural design styles, and related data and actions that can be extracted into appropriate design abstractions [Hu 2013].

Students' ability to build a good design is related to the abstraction, understanding, reasoning, and data-processing ability [Kramer 2007, Leung and Bolloju 2005, Siau and Tan 2005]. In fact, it is more difficult for students learning to design than learning to code. Besides, there is no single method for designing software. Students may confuse the different methods and may not appreciate the similarities, differences, and their uses [Carrington 1998].

How students design software has been studied previously from different points of view [Eckerdal et al. 2006a, Eckerdal et al. 2006b, Thomas et al. 2014, Thomas et al. 2017, Loftus et al. 2011, Tenenberg 2005, Hu 2016].

A study reported by Tenenberg describes the skills that students have when designing software [Tenenberg 2005]. They used the designs elaborated by two types of students: first competency students and graduating students. Each participant is given the specification of a "super alarm clock" (system to get people to sleep) to produce a design. The results indicate that graduating students use less textual design notations and more graphical and standardized notations. In addition, a large number of students underestimated the importance of representing structural groupings and interactions between design parts.

Eckerdal et al. undertook a detailed examination of the design artifacts produced by graduating students also using the "super alarm clock" task [Eckerdal et al. 2006a, Eckerdal et al. 2006b]. The designs constructed by the students are grouped using Eckerdal's categorization. They found that 62% of the students cannot design a software system. The percentages associated with the proposed categories indicate that: 3% produce nothing, 18% restatement, 41% skumtomte, 29% first step, 7% partial design and, 2% complete.

The study published by Eckerdal et al. [Eckerdal et al. 2006a] was subsequently reviewed by Loftus et al. [Loftus et al. 2011]. This work aims to find out if students about to graduate (graduating) are able to design in groups and if they are able to recognize good and bad designs produced by others. The results confirm those reported by Eckerdal et al. In addition, they found that the main things missing from the students' designs were descriptions of system behavior and consistency between the use case diagrams and the implemented designs.

A study presented by Lynda Thomas et al. [Thomas et al. 2014] expanded on the research of Eckerdal et al.[Eckerdal et al. 2006a]. They used the task, "super alarm clock," and Eckerdal's categorization. The results obtained do not turn out to be as bad in the area of students' design skills as previously reported. Most of the designs were mapped in the first step and partial design. Furthermore, the authors built the list of fundamental characteristics that differentiate how students understand the design phenomenon presented in Section2 as Thomas' category.

Chenglie Hu continues this line of work by reporting an experiment in which graduating students perform the design to the same "super alarm clock" [Hu 2016]. The results show that 50% of the designs were categorized in the partial design category, 16.6% in the first step, and 33.3% in the complete category.

Years later, Thomas et al. studied software designs of students who were halfway through their 3-year undergraduate computing degree [Thomas et al. 2017]. Students were asked to "Produce a design" using a problem that appeared to require about the same design skill level as the "super alarm clock" problem. They used Eckerdal's categorization. Then, the authors compared the results with those obtained from a group of graduating students from the same university [Thomas et al. 2014].

The results show that most of the designs from both cohorts were in the first step category, followed by the skumtomte category. The graduating students produced a few more partial and complete designs but were relatively low cases.

In general, studies have shown that most graduating students are not competent in designing software [Eckerdal et al. 2006a, Eckerdal et al. 2006b, Loftus et al. 2011]. Students do not describe the behavior of the system [Loftus et al. 2011], do not seem to understand what kind of information they should include [Eckerdal et al. 2006a] and produce incomplete class diagrams, sequence diagrams with missing responsibilities, and objects at inconsistent abstraction levels [Sien 2011].

4. Research Method

We conducted the experiment in the context of an undergraduate course to study how graduating students habitually design software and if there is an effect on software quality

when they deliver the design representation to course teachers.

4.1. Course context

The course lasts 9 weeks. In the first week (week 1), a base process (that students must follow) is taught, and the practical work dynamics are explained. Students participate in the course voluntarily.

The base process is a defined process that intends to help the software development tasks and collect product and process metrics. The process has different phases, scripts that guide the work in each phase, and logs that are used to collect data (see **Figure 1**).

The base process is divided into the following phases: plan, design, code, compile, unit test (UT), and postmortem. In each phase of the process, students must log the time dedicated to the phase and data on the defects they remove (injection phase, removal phase, and time spent to correct it). Students log the size (using a web tool) in lines of code of the program built in the postmortem phase.

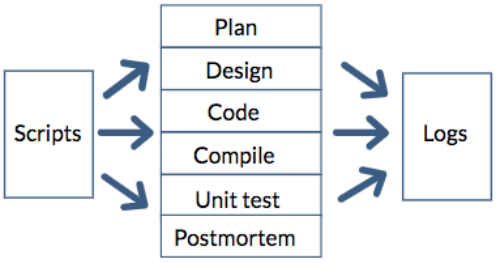


Figure 1. Base Process

The practical work consists of each student developing 8 small projects following the base process and recording the process data in the tool. Students carry out the projects individually, consecutively and at home. Project 2 does not begin until project 1 has been completed and so on. From week 2 to week 9, one project is assigned per week. At the beginning of each week, a teacher sends the student the project requirements. Each student's submission must contain the code that solves the problem, the test cases executed, and the export of the data registered in the tool.

Before starting the first exercise each student must choose the programming language to be used throughout the course. Our interest is to collect data on the execution of the development process using a programming language that the student masters. All the students have passed the courses Programming 3, Programming 4, Programming Workshop, and Software Engineering. During these courses, software design artifact and techniques, programming languages, algorithms and fundamental software engineering concepts are taught. We consider the group of students who participated to be homogeneous, since they are students at a similar stage of their career.

Projects are small in size and of low and similar complexity so design phase refers to detailed design (i.e. identifying classes, attributes, operations, program scenarios, status changes, and pseudo-code). Seven projects deal with numerical and statistical analysis problems, and one deals with a text-processing problem (project 2).

Previous studies using the same projects, showed that the nature of project 2 is different from the other projects, the process measures and product measures in project 2 are extremely different than in the rest of the projects (i.e., project 2 is an outlier), and it is usually discarded in statistical analysis [Grazioli et al. 2014]. Therefore, we excluded the data of this project from the analyzes presented in this article.

4.2. Goals and research questions

The goals of our experiment are, first, get to know if there is an effect on software quality when they deliver the design representation. We pose this goal to determine whether asking for the delivery of the habitual design representation changes student behavior and affects the software's quality. Second, get to know how students habitually represent software design (what artifacts and ways of design representation they use).

To get students to deliver the design representation they would habitually build, we do not request the use of templates or specific approaches to design representation (as we did in previous iterations of the family of the experiment).

We defined the following research questions and the corresponding research hypotheses:

RQ1: Is there any improvement in the quality of the products when students deliver the representation of their habitual design?

H1.0: Deliver the design in the habitual way does not modify defect density in the unit test phase.

H1.1: Deliver the design in the habitual way modifies defect density in the unit test phase.

We defined the defect density as the number of defects found in the unit test phase (UT) per every thousand lines of code. The consequence of high defect density in UT in software engineering is typically seen in the form of defect-fixing or rework effort incurred in projects, which results in poor quality products.

RQ2: How do students habitually represent the design?

To answer RQ2, we analyze using Eckerdal's categorization, Thomas' categorization and direct observations of the delivered designs handed by the students.

4.3. Experimental design

We carried out the experiment in 2018, and 15 students took part. Experimental design is of repeated measures of one factor (software design representation) with two alternatives: submitting the design representation or not.

To address the goals, students submit the habitual representation of their designs as part of the solution to each project. With the submission of the design representation, we seek to know the students' design practices. Even though it is an optional submission (i.e. the student can decide not to design in the design phase of the base process), we emphasize the importance of designing in the way they habitually do so, following their usual design representation. Besides, students are told that their designs will not influence the grading.

Students carry out the first four projects following the base process. Then, students are randomized divided into two groups: "habitual design delivery" (*HDD*), and "without

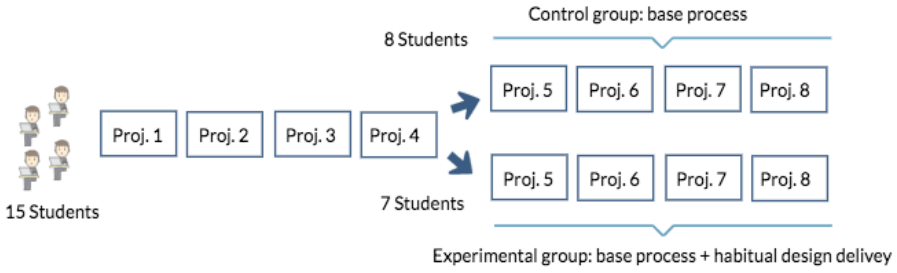


Figure 2. Experimental design

habitual design delivery" (*noHDD*). The control group (*noHDD*) carries out projects 5 to 8 following the base process. The *HDD* group continues applying the base process and also has to submit their design representations. The only difference between groups is the submission of the representation. However, it is important to remark that the students of both groups should produce a design for each of the programs as part of the design phase of the base process. In the *HDD* group, 7 students participated, and in the *noHDD* group, 8 students participated (see Figure 2).

The response variables considered in this experiment are external quality, measured as the defect density in the unit test phase of the base process, and the design quality. The design quality is measured using the design categorizations already mentioned.

5. Analysis and Results

5.1. Quality Improvement

To answer RQ1: "Is there any improvement in the quality of the products when students deliver the representation of their habitual design?" we analyzed the following hypothesis test:

H1.0: Delivering the habitual design does not change the defects density in unit test
H1.1: Delivering the habitual design does change the defects density in unit test

For each student, we calculated two average defect density in UT: for projects 1, 3 and 4, and projects 5 to 8. The calculation of each sample for projects 5 to 8 is as follows (the same applies for projects 1, 3, and 4):

$$1000 * \frac{\sum_{n=5}^{8} \#defectsUT_n}{\sum_{n=5}^{8} \#LOC_n} \quad (1)$$

Table 1 presents the average defect density (dd) in UT for the 8 students in the *noHDD* group and the 7 students in the *HDD* group projects 1, 3, and 4 and projects 5 to 8.

First, we studied if exists a difference in the quality of the products between the group's *HDD* and *noHDD* in projects 1, 3, and 4. During projects 1, 3, and 4, both groups apply the base process, so comparing the software quality of both groups during those projects allows to confirm that they are homogeneous groups. For this analysis, we defined the following hypothesis test:

Table 1. average defect density (number of defects in the unit test phase per every thousand lines of code) for the students in both groups

Group	St.	dd134	dd58	Group	St.	dd134	dd58
noHDD	1	30.51	28.05	*HDD*	1	27.78	58.82
noHDD	2	11.63	24.69	*HDD*	2	9.48	28.90
noHDD	3	8.47	21.18	*HDD*	3	85.71	120.00
noHDD	4	98.98	82.88	*HDD*	4	26.12	14.13
noHDD	5	15.56	19.17	*HDD*	5	31.25	15.47
noHDD	6	7.35	0.00	*HDD*	6	60.61	52.12
noHDD	7	62.50	21.19	*HDD*	7	19.48	70.51
noHDD	8	60.47	55.56				

H1.0: Median(Def. density in UT i) = Median(Def. density in UT j)
H1.1: Median(Def. density in UT i) <> Median(Def. density in UT j)
being i, j the students of HDD and noHDD groups in projects 1, 3 and 4.

The value of the median and the interquartile range of the defects density for the *noHDD* group are (23.0, 33.5) and for the *HDD* group are (27.8, 26.5) respectively. In this case, the samples correspond to defects density in UT of different students. So, we applied the Mann-Whitney-Wilcoxon test for independent samples.

Defects density in UT for students in the *HDD* group (Mdn = 27.8) did not differ significantly from students in the *noHDD* group (Mdn = 23.0) when considering projects 1, 3, and 4, W = 24 and p = 0.694. So, we can assume that both groups have similar or homogeneous behavior with respect to defect density.

To know if software quality changes when students deliver their habitual design representation, we analyzed the defect density between and intra groups. Between groups refers to knowing if exists a significant difference in the software quality between the groups. Intra groups refers to studying software quality in the *HDD* group before and after the delivery of the design representations.

Between groups

The hypothesis test is similar to the one described before but considering projects 5 to 8 instead of 1, 3, and 4. We applied the Mann-Whitney-Wilcoxon test again as the samples are independent. The values of the median and the interquartile range for the *HDD* group are (52.1, 37.2) and for the *noHDD* group are (25.7, 25.8).

Defects density in UT for students in the *HDD* group (Mdn = 52.1) did not differ significantly from students in the *noHDD* group (Mdn = 25.7) when considering projects 5 to 8, W = 20 and p-value = 0.397. Students who deliver their habitual designs did not differ significantly from those who do not deliver their habitual designs.

Intra groups

We also studied if there is any change within the group who delivered their habitual design (*HDD* group). We compared the defect density when they did not delivered their design (projects 1, 3 and 4) and when they delivered their design (projects 5 to 8). For this analysis, we defined the following hypothesis test:

H1.0: Median(Def. density in UT i) = Median(Def. density in UT j)
H1.1: Median(Def. density in UT i) <> Median(Def. density in UT j)
being i the students of HDD in project 1, 3 and 4, j the students of HDD in project 5 to 8.

The values of the median and the interquartile range for the *HDD* group in projects 1, 3 and 4 are (27.7, 26.5) and for the same group in projects 5 to 8 are (52.1, 37.2).

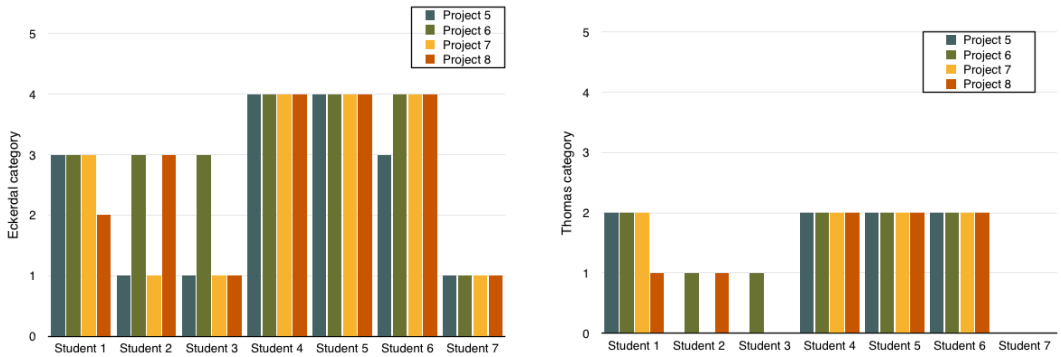
The samples are paired since the sampled pairs (Def. density in UT i, Def. density in UT j) correspond to the same student, so we applied Wilcoxon signed-rank test.

Defects density in UT for students in the *HDD* group in projects 1, 3 and 4 (Mdn = 27.7) did not differ significantly from students in the HDD group in projects 5 to 8 (Mdn = 52.1), V = 6, p-value = 0.219. That is, the delivery of the design representation did not differ significantly from the non-delivery of the design representation.

5.2. Habitually software design representation

To answer research question RQ2: "How do students habitually represent design?" we analyzed the 28 design representations submitted by the 7 students of *HDD* group from project 5 to project 8 using Eckerdal et al. categories, Thomas et al. categories, and direct observations. In this case, we do not have a control group because we do not have the design representation for students in the *noHDD* group. Therefore, it is not possible to answer this question using the framework of controlled experiments, but we do it in an exploratory and analytical way.

We mapped each design to Eckerdal's categorization and Thomas's categorization. The first author of the paper did the mapping, and the second author evaluated it both resolved discrepancies by discussion. Figure 3 illustrates the mapping associated with Eckerdal's and Thomas's categorizations for the 7 students for projects 5 to 8.



(a) Eckerdal (b) Thomas

Figure 3. Eckerdal and Thomas categories for the 7 students in projects 5 to 8

From Figure 3 arise that the ranking of some students' designs (students 1, 4, 5, 6 and 7) across the different projects does not change much. That is, their design is relatively consistent according to these categorizations.

Furthermore, it can be observed how some students (students 4, 5, and 6) managed to elaborate intermediately designs mapping to Eckerdal's categories 3-4. On the other

hand, students 2, 3, and 7 elaborate more basic designs that categorize in the lowest Eckerdal and Thomas categorizations.

Some interesting aspects of the results associated with Eckerdal's categorization are observed. Students 4, 5 and 6 produced designs that provide an understandable description of each of the parts and an overview of the system that illustrates the relationships between the parts, without completely describing communications between the parts. Student 3 (in three of the four projects) and 7 handled designs that merely restated the requirements (i.e., it is not a software design). Student 1 generally manages to describe the system and its parts partially but fails to relate them. Finally, student 2's designs vary, ranging from only transcribing the requirements (half of the projects) to partially describing the system and its parts (the other half of the projects).

Using Thomas's categorization, it can be observed that the students manage basic designs. The delivery of students 2, student 3 and student 7 are in most of the projects (in 9 projects out of 12) text notations (category 0ID). In the remaining 3 projects students manage to identify some part of the system in an incompetent way (category 1A). Designs of students 4, 5, and 6 fall into category 2SS.; that is, their designs use static notation. Specifically, the designs elaborated by these students consist of incomplete class diagrams of the system. Student 1 elaborates incomplete class diagrams in the delivery of projects 5, 6, and 7 (category 2SS.), and identifies some incomplete part of the system for project 8 (category 1A)

Finally, no designs fall into categories 3DB, 4MRA and 5EC, which implies the ability to represent dynamic design with multiple related artifacts.

To add a complementary perspective to the students' design analysis, we analyzed project 8 in detail as a sample of the projects submitted. Table 2 presents for the 7 students a description of the main characteristics of the design submitted, the defect density in UT (dd), the Eckerdal's category and Thomas' category for project 8.

It can be observed that students 4 and 5 manage to represent designs using both class diagrams and pseudo-code. According to the categorizations of Eckerdal et al. and Thomas et al., these two students were the ones who categorized their designs better. It could be observed that these two students are the ones with less defect density in UT for project 8 (see Table 2). The rest of the students represent their design more informally, transcribing the requirements to natural language, doing a pseudo-code, or incomplete class diagrams. In these cases, the defect density in UT is higher.

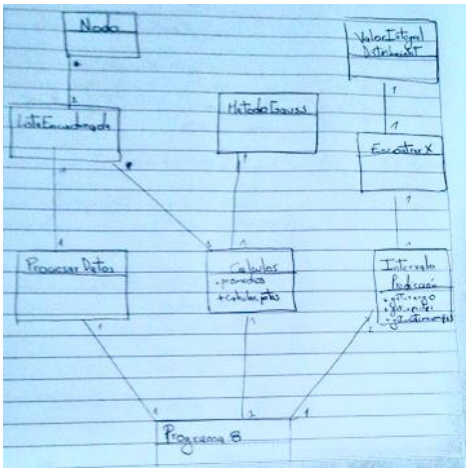
In order to illustrate what kind of design the students deliver, some of the designs are presented as they were delivered in project 8 (the most complex project of the course). Figure 4 illustrates a part of the delivery of the design representation for students 5, 6 and 7. Student 5 makes a pseudo-code of a part of the project, student 6 develops an incomplete class diagram and student 7 transcribes the requirements into natural language. Figures correspond to the actual capture sent by the students of their design reflected on the sheet.

6. Discussion

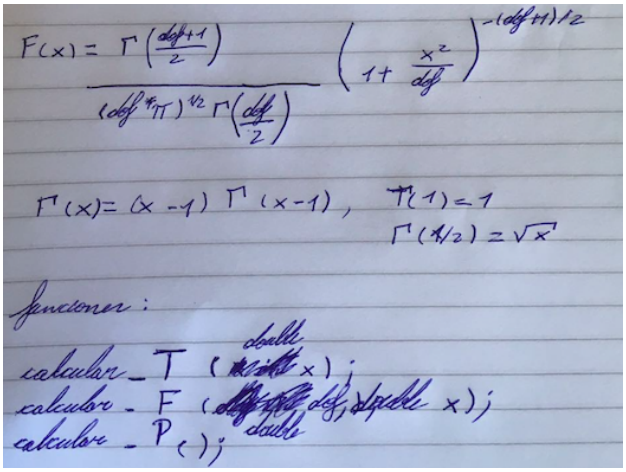
In our experiment, we observed that the submission of the habitual design representation does not have an impact on the software quality produced by students. The results of the

Table 2. Main characteristics of the design submitted, the defect density in UT, the Eckerdal's category and Thomas' category for project 8

St.	Design description	dd	Eck.	Tho.
1	uses natural language to explain that he will extend a class by adding a method	33.78	2S	1A
2	performs a pseudo code of a part of the project	43.80	3FS	1A
3	transcribe the requirements to natural language, inputs and expected outputs of the project	66.67	1A	1R
4	performs a pseudo code of a part of the project, identifying classes, methods and attributes	14.71	4PD	2SS
5	performs a pseudo code of a part of the project, identifying classes, methods and attributes	8.40	4PD	2SS
6	makes a class diagram, identifying attributes and relationships between classes	43.48	4PD	2SS
7	transcribe the requirements to natural language	142.86	1R	0ID



(a) Student 6



(b) Student 7

```
h := 0 /* Initialization of the pivot row */
k := 0 /* Initialization of the pivot column */
while h < m and k < n
       /* Find the k-th pivot: */
       max = abs(A[h,k])
       i_max = h
       for i = h+1, h < m i++
              if abs(A[i,k]) > max
                     i_max = i
              end
       end

       if A[i_max, k] = 0
              /* No pivot in this column, pass to next column */
              k := k+1
       else
              //swap rows(h, i_max)
                     aux = A[h]
                     A[h] = A[i_max]
                     A[i_max] = aux
              /* Do for all rows below pivot: */
              for i = h, h<m, i++:
```

(c) Student 5

Figure 4. Habitual design representation for students 5, 6 and 7 in project 8

statistical tests applied (intra and between groups) indicate that the quality of the group delivering the design representation does not vary significantly concerning the quality of the group that does not deliver it. However, the median value indicates that the submission of the usual design representation seems to negatively impact the software compared to the non-submission of the usual representation. This comes to our attention because we did not expect any difference between groups (considering that the experimental group was handling their habitual design). Furthermore, if there were differences in the medians (as is the case), we would expect the opposite results (i.e., students "caring" more about submitting their design and this having a positive impact on software quality). New questions arise from this: How does the design request affect the student's usual design-code process? Does the design request modify the student's habitual way of coding?

Using the categorizations by Ekerdal et al. and Thomas et al., we can affirm that students (in our Engineering School) design on a basic level. Even though they do not achieve complete designs that combine several artifacts to model dynamic and static aspects simultaneously, they manage to produce simple designs. We can observe that using natural language to transcribe requirements (restatement category) is a habitual practice of some students to represent their designs (student 2, student 3, and student 7). Two of these three students manage to describe the system and its parts in some exercise partially. Students 4, 5, and 6 provide an understandable description of the system's parts that illustrates the relationships between them (partial design category). Finally, student 1 manages to partially describe the system without relating its parts (first step category). No student used sequence, use case, collaboration, or dynamic diagrams. Furthermore, the delivered designs vary between text notations, the incomplete identification of the parts of the system, and the use of static notations to describe the system.

Some previously reported results are similar to ours [Eckerdal et al. 2006a, Eckerdal et al. 2006b, Thomas et al. 2014, Thomas et al. 2017]. Most of the designs categorized by Eckerdal et al. are associated with basic categories (18% restatement, 41% skumtomte, 29% first step) [Eckerdal et al. 2006a, Eckerdal et al. 2006b]. In the study reported by Thomas et al. [Thomas et al. 2014], the majority of designs categorized was in the first step and in partial design. Results from their following study [Thomas et al. 2017] shows most designs categorized were in the first step, followed by the skumtomte category. These authors agree that students do not know how to design, and our results show the same.

This experiment and the previous experiments in our family of experiments allowed us to know how graduating students are currently designing, their habitual practices, and the effects of design on software quality [Moreno and Vallespir 2018, Moreno et al. 2020, Moreno et al. 2021]. From the family results, we believe that a usual practice of the students when developing software is to follow the code-and-fix model. That is to say, they do not stop to think of a solution (design), but instead, they rush to code. Students dedicate at least three times less time to design than to coding. Also, using templates for the design representation did not help the students develop software products of a higher quality. These results in addition to those obtained in this experiment reveal that students do not have a great domain of design techniques and how to combine them. We believe, just as Loftus et al. [Loftus et al. 2011], that students have difficulties in designing software. Within the context of graduating students, those difficulties may

be associated with several reasons: the lack of experience, the lack of awareness of the importance of design on quality, and the lack of education regarding design techniques among others.

7. Threats to validity

This section describes the threats to validity we have detected.

Internal validity - The fact that the context of the experiment is a course implies that the students do not develop naturally. We tried to minimize this threat with a non-graded course, that is, a course in which the student passes or fails. Besides, we remarked on the importance of monitoring and registering the process just as it was, and we emphasized that students' assessments would not be done according to results, defects found, efforts made, or delivered designs.

External validity - In our experiment, students voluntarily took part in a course, and they signed an informed consent that the data recorded and the designs delivered in the process could be used for research purposes, guaranteeing anonymity. Conversely, the results obtained in this experiment (academic context) cannot be generalized to the student's design practice in a professional context.

Construct validity - We measured software quality as the number of defects in UT and design quality using design categorizations (Eckerdal and Thomas). Although we used software quality and design quality scales from previous research, these are complex constructs to measure, and there could be a bias between theory and construction.

Conclusion validity - The number of students in the research constitutes a threat to the statistical conclusion (only 15). This causes the statistical analysis to be carried out using non-parametric tests with lower statistical power than the parametric tests. To measure this threat, we completed the non-parametric tests with descriptive statistics and in an observational way.

8. Concluding Remarks and Future Work

In this paper, we report the results from an experiment to know how students habitually represent detailed design (what artifacts and ways of design representation they use) and know if there is an effect on software quality when they deliver the design representation.

We analyzed the software quality (defect density) between and intra groups. The results of the statistical tests applied in both groups indicated that the quality of the group delivering the design representation does not vary significantly concerning the quality of the group which does not deliver it.

Also, we analyzed the design representations delivered by the students and categorized them using the categorizations by Eckerdal and Thomas.We found that the delivered designs vary between text notations, the incomplete identification of parts of the system, and the use of static notations to describe the system. Our results are similar to those reported in the literature [Eckerdal et al. 2006a, Eckerdal et al. 2006b, Thomas et al. 2014, Thomas et al. 2017], and we agree with them on the fact that students do not know how to deliver a design representation beyond using basic artifacts.

We propose a new experiment and conduct interviews with students as they carry out the projects as future work. Our interest is to know how they face design thinking

during the design phase, if they carry out any design aspect during the code, why they choose the design artifacts they use, if they know others, why they are not used, etc.

References

Bourque, P. and Fairley, R. E. (2014). *Guide to the Software Engineering Body of Knowledge - SWEBOK v3.0*. IEEE Computer Society, 2014 version edition.

Carrington, D. (1998). Teaching software design and testing. In *FIE'98. 28th Annual Frontiers in Education Conference. Moving from'Teacher-Centered'to'Learner-Centered'Education. Conference Proceedings (Cat. No. 98CH36214)*, volume 2, pages 547–550. IEEE.

Carrington, D. and K Kim, S. (2003). Teaching software design with open source software. In *33rd Annual Frontiers in Education*.

Chemuturi, M. (2018). *Software Design: A Comprehensive Guide to Software Development Projects*. CRC Press/Taylor & Francis Group.

Chen, T.-Y., Cooper, S., McCartney, R., and Schwartzman, L. (2005). The (relative) importance of software design criteria. *SIGCSE Bull.*, 37(3):34–38.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Zander, C. (2006a). Can graduating students design software systems? In *SIGCSE Bull.*, page 403–407. ACM, Association for Computing Machinery.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Zander, C. (2006b). Categorizing student software designs: Methods, results, and implications. *Computer science education*, 16(3):197–209.

Grazioli, F., Vallespir, D., Pérez, L., and Moreno, S. (2014). The impact of the psp on software quality: Eliminating the learning effect threat through a controlled experiment. *Adv. Soft. Eng.*, 2014.

Hu, C. (2013). The nature of software design and its teaching: an exposition. *ACM Inroads*, 4(2).

Hu, C. (2016). Can students design software? the answer is more complex than you think. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, Computer Science Education 2016, page 199–204, New York, NY, USA. Association for Computing Machinery.

Joint Task Force on Computing Curricula - ACM and IEEE Computer Society (2013). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA.

Kramer, J. (2007). Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42.

Leung, F. and Bolloju, N. (2005). Analyzing the quality of domain models developed by novice systems analysts. In *38th Hawaii International Conference on System Sciences*.

Linder, S. P., Abbott, D., and Fromberger, M. J. (2006). An instructional scaffolding approach to teaching software design. *Journal of Computing Sciences in Colleges*, 21.

Loftus, C., Thomas, L., and Zander, C. (2011). Can graduating students design: revisited. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM.

Moreno, S., Casella, V., Solari, M., and Vallespir, D. (2020). La representación del diseño detallado utilizando plantillas y sus efectos en la calidad del software. In *En Proceedings XXIII Ibero-American Conference on Software Engineering (CIBSE 2020)*.

Moreno, S. and Vallespir, D. (2018). ¿los estudiantes de pregrado son capaces de diseñar software? estudio de la relación entre el tiempo de codificación y el tiempo de diseño en el desarrollo de software. In *Conferencia Iberoamericana de Ingeniería de Software 2018*.

Moreno, S., Vallespir, D., Solari, M., and Casella, V. (2021). Representation of software design using templates: impact on software quality and development effort. *Journal of Software Engineering Research and Development*, 9(1):1 – 15.

Siau, K. and Tan, X. (2005). Improving the quality of conceptual modeling using cognitive mapping techniques. *Data & Knowledge Engineering*, 55(3). Quality in conceptual modeling.

Sien, V. Y. (2011). An investigation of difficulties experienced by students developing unified modelling language (uml) class and sequence diagrams. *Computer Science Education*, 21(4):317–342.

Tenenberg, J. (2005). Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4.

Thomas, L., Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., and Zander, C. (2014). Graduating students' designs: Through a phenomenographic lens. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, page 91–98. Association for Computing Machinery.

Thomas, L., Zander, C., Loftus, C., and Eckerdal, A. (2017). Student software designs at the undergraduate midpoint. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 34–39, New York, NY, USA. Association for Computing Machinery.