Journal of Software Engineering Research and Development, 2021, 9:7, doi: 10.5753/jserd.2021.1049

Representation of software design using templates: impact on software quality and effort

Silvana Moreno [**Universidad de la República, Uruguay** | ***smoreno@fing.edu.uy***]
Vanessa Casella [**Universidad de la República, Uruguay** | ***vcasella@fing.edu.uy***]
Martín Solari [**Universidad ORT Uruguay** | ***martin.solari@ort.edu.uy***]
Diego Vallespir [**Universidad de la República, Uruguay** | ***dvallesp@fing.edu.uy***]

Abstract
As a practice, software design seeks to contribute to developing quality software. During this software development stage, the requirements are translated into a representation of the software (also known as design), whose quality can be evaluated and improved. For undergraduate students, the design is difficult to understand and make. In fact, building a good design seems to require a certain level of cognitive development that few students achieve. The aim of this study is to know the effort dedicated to software detailed design and the effect on software quality when graduating students use templates to represent their design. We conducted a controlled experiment where students develop eight projects following a defined process and recording data from its execution in a software tool. We found that the use of design templates did not improve the quality of the code, measured as the defect density in the unit test phase. Also, the use of templates did not reduce the number of code smells in the analyzed code. Regarding the effort, students who use templates dedicated greater development effort to designing than to coding. Meanwhile, students who did not use templates dedicated four times less effort to designing than to coding.

Keywords: *detailed design, software quality, graduating students*

1 Introduction

Software design is one of the most important components to ensure the success of a software system (Hu, 2013). Between the requirements analysis phase and the software building phase, 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). This work is focused specifically on detailed design.

Design is a difficult 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). 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).

Building quality software is increasingly relevant. We highly depend on software in our daily lives and its quality has a great impact. A quality software design allows us to build quality software, with fewer defects and is more maintainable. Industry practitioners are aware of the importance of software design quality and they use *clean code* practices, reviews and tools, among others, to contribute in this regard (Brown et al., 1998; Fowler, 2018; Stevenson and Wood, 2018).

Knowing how undergraduate students design is of interest to several authors (Chen et al., 2005; Eckerdal et al., 2006a,b; Loftus et al., 2011; Tenenberg, 2005). Most of their studies found that students do not manage to produce a good software design. Some of the problems detected are lack of consistency between design artifacts and code, incomplete designs, and the lack of understanding of what kind of information to include when designing software (Eckerdal et al., 2006a,b; Loftus et al., 2011).

In this work, we study the software design practice in graduating students. We conducted an experiment within the context of some courses over three consecutive years to know the effort dedicated to software design and the effect that the representation of design using specific templates has on software quality. We use the term graduating for our students, because, in fact, they are in the fourth year of the degree of the School of Engineering of Universidad de la República, in Uruguay. 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, etc.

This work is an extension of the article published at the Iberoamerican Conference on Software Engineering (CIbSE) 2020: "The representation of detailed design using templates and their effects on software quality". Our article was selected to participate for the publication of in a special issue in the Journal of Software Engineering Research and Development (JSERD).

Below, we detail the extension of our work with respect to CIbSE article: The work presented at CIbSE 2020 aims to know the effect on software quality when graduating students use templates to represent the detailed design. In this work we present an empirical study where students develop 8 projects following a defined process and recording data from the execution in a tool. We found that the use of design templates did not improve the quality of the code measured as the defects density in the unit test phase. Neither did the use of templates manage to reduce the number of code smells present in the analyzed code. The extension carried out in this work consists, on the one hand, of expanding and deepening aspects that for limited space reasons are not in the CIbSE article. On the other hand, we add a new research question and its analysis, which allows to knowing the effort that implies the use of design templates.

Specifically, a new section explaining the experimental design in depth was added. The analysis of external quality was expanded and deepened. Descriptive statistics were added and analyzed and tables were added with the data of the average density of defects in UT for the students. In addition, a statistical analysis was added within the between-group analysis that checks the homogeneity of the groups studied (TRD, noTRD). Threats to validity were expanded, grouping them by type (construct, internal, external, conclusion), and Discussion and Conclusions sections were expanded.

A research question was added that seeks to know the effort that students dedicate to design, and how that effort varies after the use of templates. To answer this question, the relationship between the effort dedicated to the design phase and the effort dedicated to the coding phase was studied. Descriptive and statistical analyses were presented as part of the analysis of results. The results obtained are discussed and related to those previously obtained in the discussion section.

The document is structured as follows: Section 2 presents related works; section 3 presents the research methodology; section 4 presents the results, and section 5 is discussed; threats to validity are mentioned in section 6, and section 7 presents the conclusions and future work.

2 Related Work

Software design is an important activity to ensure the quality of a software system (Hu, 2013; Taylor, 2011). It involves identifying and abstractly describing the software system and its relationships. Good designs help develop robust, maintainable software and with few defects (Pierce et al., 1991; Sommerville, 2016). Detailed software design 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).

Software quality is the degree to which a software product meets stakeholders' needs both explicit and implicit. Quality models represent quality in terms of a set of elements of the model and their relationships (Nistala et al., 2019). These models define internal and external software quality attributes. The internal ones are those that do not depend on the software execution (static), while the external ones are those that are applicable to the execution.

In recent years, the use of clean code practices and tools has contributed to improved design quality (Stevenson and Wood, 2018). Code smells, anti patterns and design flaws can be used to measure the quality of a software design (Martin, 2002; Gibbon, 1997; Brown et al., 1998; Fowler, 2018). SonarQube (Campbell and Papapetrou, 2013) and FindBugs (Ayewah et al., 2008) are some of the tools used to measure the quality of the code by detecting *bad smells*.

Current industry practices require practitioners with the necessary skills to understand and build good software designs. However, students have difficulties 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).

In fact, for students, learning to design is more difficult than learning to code. This difficulty occurs because for most programming languages, students get compiler feedback and run time errors. However, this does not happen with design (Karasneh et al., 2015).

Object-Oriented Design (OOD) is one of the most widely-used design approaches in the industry and one of the subjects normally taught in universities (Flores and Medinilla, 2017). By using OO modeling diagrams and languages, static and dynamic models of software systems can be created. Several empirical studies analyze the understanding and benefits of using UML diagrams (Budgen et al., 2011; Fernández-Sáez et al., 2013; Arisholm et al., 2006; Gravino et al., 2015; Torchiano et al., 2017).

In some studies, students failed to obtain design benefits using UML diagrams (Gravino et al., 2015; Torchiano et al., 2017). Gravino et al. found that students who use UML diagrams to design do not make significant improvements in their source code comprehension tasks compared to students who do not use them. Also, students who use diagrams spend twice as much time on the same source code comprehension task than as students who do not use them. When analyzing the experience factor, they find that the most experienced students achieve an improvement in the understanding of the source code (Gravino et al., 2015; Soh et al., 2012).

For industry professionals, the use of UML continues to be resisted to a certain degree (Stevenson and Wood, 2018). A survey conducted to on 50 software professionals indicates that although the quality of the software is an important aspect, the use of UML is selective (informal, only for a while, then it is discarded) and with low frequency (Petre, 2013).

The use of Model-Driven Development (MDD) methodology to design software has shown improvements in software quality. Panach et al. conducted an experiment and found that students using MDD achieve better quality products (measured through test cases) than students using the traditional software development method (Panach et al., 2021).

Undergraduate students' design skills are reported by previous studies examining artifacts produced by them to learn how they design software (Chen et al., 2005; Eckerdal et al., 2006a,b; Loftus et al., 2011; Tenenberg, 2005). These studies use the same requirements specification for which students must produce a design. The studies use different approaches: designs produced individually, designs made in groups, and designs produced at different levels of training.

In general, all the works mentioned agree on the fact that graduating students are not capable of designing a software system. Lack of consistency between design artifacts and code, incomplete designs, and lack of understanding of what kind of information to include when designing software are some of the major difficulties reported (Eckerdal et al., 2006a,b; Loftus et al., 2011).

We believe, just as Loftus et al. (Loftus et al., 2011), that students do not precisely know what to do when they have to design software. Besides, several authors analyzed the artifacts produced and they agree on the fact that students do not know how to design (Chen et al., 2005; Eckerdal et al., 2006a,b; Loftus et al., 2011; Tenenberg, 2005). This motivated the work presented in this paper, in which we provide students with design templates as a support tool for design representation. Unlike Gravino and Torchiano, who analyzed the benefits of using diagrams in code comprehension (Gravino et al., 2015; Torchiano et al., 2017), our approach tries to analyze the effort dedicated to designing and coding; and the impact of the use of templates on software quality. We studied quality from two perspectives: defects on the code and code smells. We also analyzed the effort as the time in minutes that students dedicate to the design and code phases.

The focus of our research is the OOD at the class level, including source code organization, the identification and relationship between classes, and the interaction of users with the system. As Kitchenham pointed out (Kitchenham and Pfleeger, 1996), this corresponds, to the "Product View", an examination of the inside of a software product. We used an approach focused on objects because a large part of the current software is developed using that technology (Group, 2015).

3 Research Methodology

We studied the effect of design in software quality when graduating students represent their design using a specific set of templates and the effort they dedicate to the design activity. We conducted three experiments within the context of three consecutive undergraduate courses, from 2015 to 2017.

3.1 Course context

The course Principles and Foundations of Personal Software Process (PF-PSP) have the same format every year and lasts 9 weeks. In the first week (week 1), a base process is taught, and the dynamics of the practical work to be done throughout the remaining eight weeks are explained. Students participate in the course on a voluntary basis.

The base process is a defined and disciplined process that intends to help the software development tasks and to 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. To follow the process, students are provided with a set of scripts.

Scripts are a one page guide that establishes the inputs, outputs and activities to be carried out in each phase. Scripts help students guide the development activities but without demanding how they must be carried out.

In each phase of the process, students must log the time dedicated to the phase, as well as data on the defects he or she removes (injection phase, removal phase, type of defect and time spent to correct it). In the postmortem phase students log the size in line of code (LOC) of the program built.

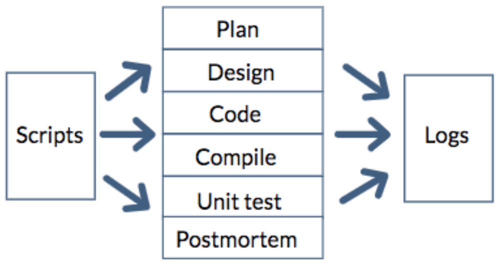


Figure 1. Base Process

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

Students carry out the projects at home and have a teacher assigned, who will be responsible for assigning the projects, correcting them and answering questions.

Before starting project 1, each student must choose the programming language to use throughout the course. Our interest is to collect data of the execution of the development process with the use of a programming language familiar to the student. Projects are small in size and of low and similar difficulty, so the design phase refers to detailed design (i.e. identifying classes, attributes, operations, program scenarios, status diagram, and pseudocode).

The nature of project 2 is different from the other projects. In project 2, students have to build a size-measuring software, while in the remaining projects, they must produce mathematical solutions (standard deviation, Simpson's rule, correlation parameters). Previous studies show that process measures and product measures in project 2 have greater difficulty 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., 2014b; Moreno and Vallespir, 2018). Therefore, we excluded the data of this project from the analyzes presented in this article. However, it is relevant to mention that project 2 is an integral part of our course, and it is used for students from projects 3 to 8 to count the lines of code they produce in each project.

Percentiles 5 and 95 of the data collected for all the students throughout the 8 projects are 26 LOC and 242 LOC respectively.

Each replication of the experiment corresponds to an instance of a different run of the course. Students who participated in one course do not participate again in a later course. The teachers participating were the same throughout the three courses (2015-2017).

3.2 Goals and research questions

The aims of the experiment are to know the effect on software quality when students represent their designs using templates, and to study the effort they dedicate to the design activity. Templates are documents with a predefined structure in which students have to represent their designs.

The templates we used allow to describing the detailed design of a project. We used four templates, a brief description of each of them is presented below:

- Operational template: specifies the interaction between the program and the users. The content may look similar to a use-case description.
- Functional template: the behavior of the program's invocations and returns are specified in this template. Variables, functions, classes and methods are described. **Figure 2** presents an example of the use of this template for project 6.
- Logical template: in this template, the pseudocode of each method that appears in the functional template is registered.
- State template: it can be used to define the transactions and conditions of the program's internal states. The content is similar to state machine diagrams.

The selected templates emerge from the Personal Process (PSP) framework(Humphrey, 1995). The PSP considers a design to be complete when it defines all four dimensions (internal-static, internal-dynamic, external-static, external-dynamic). The way to correspond to each of the four dimensions is by using the four templates (Operational, Functional, Logical, State). Completing the four templates allows describing the designs entirely and precisely (Humphrey, 1995). Several studies have shown an improvement in developer performance with templates insertion (Hayes and Over, 1997; Prechelt and Unger, 2001; Gopichand et al., 2010).

In the experiment context, we proposed the following research questions and the corresponding research hypotheses:

RQ1: Is there an improvement in the quality of the products when students represent the design using templates?

RQ2: What is the relation between the effort dedicated to designing and the effort dedicated to coding? Are there any variations in effort when students use templates?

To answer RQ1, we analyzed the external and internal quality of the software developed in each project. To study the external quality, we considered the following research hypothesis:

H1.0: Representing software design using design templates, does not change the software defect density in unit testing

H1.1: Representing software design using design templates, changes the software defect density in unit testing

To study the internal quality, we descriptively analyzed certain code smells introduced by students when producing software (Fowler, 2018). We are interested in knowing if the use of templates to represent software design prevents students from incurring into some type of code smells.

To answer RQ2, we studied the time spent on the design and code phases. We analyzed the following research hypothesis:

H2.0: The time spent on designing equals the time spent on coding.

H2.1: The time spent on designing does not equal the time spent on coding.

3.3 Experimental design

Our design is a repeated measures design with one factor (the base process) and two levels: with templates to represent the software design and without templates to represent the software design. Response variables considered in this experiment are internal and external software quality, and the effort dedicated by the students to the design and code phases.

Our experimental design implies that students develop 8 projects. The base process introduces practices in the first 2 projects that allow for guiding the work and measure the process. Therefore, during the first or second project (depending on the subject), they are already following the process adequately.

People have high variability among themselves when applying software development techniques or processes (Humphrey, 2005). When high variability among people exists in an experiment with human subjects, a within-subjects design is preferable to a between-subjects experiment (Senn, 2002). Moreover, in repeated measures experiments, subjects serve as their own control (Jones and Kenward, 2014). This reinforces the choice of our design, in which each student carries out several projects.

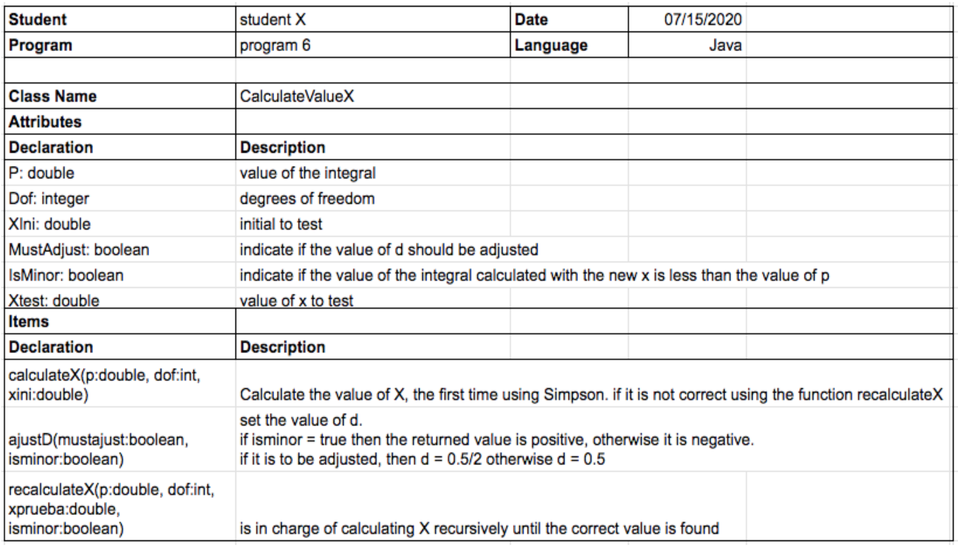
The effect of students' learning throughout these 8 exercises could be a problem in our experimental design. However, this was previously studied from different approaches, and the results indicate in both studies that repetition of programming did not contribute to performance improvements (Grazioli et al., 2014b; Grazioli and Nichols, 2012; Grazioli et al., 2014a).

As we already mentioned, to evaluate the external quality, we considered the defect density in the unit test phase of the base process. That is to say, the number of defects detected in that phase are counted and divided between the LOCs of the project.

To evaluate the internal quality, we analyzed the code smells in which students incur. Knowing the number of code smells present in the product's source code gives us an idea of the maintenance costs in the future (Fowler, 2018).

The effort in design and code is measured as the time in minutes that the student dedicates to the phase in question.

The experimental design is presented in **Figure 3**. All students apply the base process in projects 1 to 4, in which submitting the design representation to the teachers is not required. When students finished project 4, they were divided randomly into two groups: the control group and the experimental group. The control group, called "without templates to represent the design" (*noTRD*), continues to apply the base process throughout projects 5 to 8. The experimental group, called "with templates to represent the design" (*TRD*), started to apply the templates from project 5 to 8.



Student	student X	Date	07/15/2020
Program	program 6	Language	Java
Class Name	CalculateValueX		
Attributes			
Declaration	Description		
P: double	value of the integral		
Dof: integer	degrees of freedom		
XIni: double	initial to test		
MustAdjust: boolean	indicate if the value of d should be adjusted		
IsMinor: boolean	indicate if the value of the integral calculated with the new x is less than the value of p		
Xtest: double	value of x to test		
Items			
Declaration	Description		
calculateX(p:double, dof:int, xini:double)	Calculate the value of X, the first time using Simpson. if it is not correct using the function recalculateX		
ajustD(mustajust:boolean, isminor:boolean)	set the value of d. if isminor = true then the returned value is positive, otherwise it is negative. if it is to be adjusted, then d = 0.5/2 otherwise d = 0.5		
recalculateX(p:double, dof:int, xprueba:double, isminor:boolean)	is in charge of calculating X recursively until the correct value is found		

Figure 2. Functional Template

The *TRD* group attends a theoretical class where the four design templates are presented and explained (and examples are shown). The submission of the design representation for this group was mandatory (except for the state template which is optional). When a student submitted the project, the assigned teacher checked the completeness of the templates and the consistency with the code. In this way, students designing a solution and then coding another one is reduced. However, the fact that the design is complete and verifiable is not controlled.

Our experimental design allows us to study the behavior of the groups before and after the use of the templates. On the one hand, we propose to analyze the *TRD* (representing design using template) and *noTRD* (representing design without template) groups during project 1 to 4 to confirm they are homogeneous groups; that is, the quality of the software developed is similar in both groups from programs 1-4 (when students do not use templates in any of the groups).

On the other hand, we are interested in knowing if students who use templates develop better-quality software. We propose studying the groups *TRD* and *noTRD* during projects 5 to 8 to know if representing the design using templates has some effect on the software quality.

3.4 Operation

The experiment was replicated in the course for three years: 2015, 2016, and 2017. The number of students that took part in the experiment was: 25, 17, and 19 respectively.

Out of the 61 students participating in the experiment, 29 are part of the *TRD* group, and 32 of the *noTRD* group. This unbalance between the groups is due to the unbalance generated when students were assigned to the *TRD* and *noTRD* groups in each of the three replications.

4 Analysis and Results

To answer RQ1: "Is there any improvement in the quality of the products when students represent the design using templates?", we analyzed the quality from the internal and external points of view.

4.1 External Quality

We measured the external quality as the defect density in UT, that is, the number of defects in UT/KLOC. To analyze the external quality, we defined the following research hypotheses:

H1.0: Representing software design using design templates does not change the software defect density in UT

H1.1: Representing software design using design templates changes the software defect density in UT

We analyzed the external quality in two ways: intra groups and between groups. Between groups refers to knowing if there is a significant difference in the quality between the *TRD* group and the *noTRD* group. Intra group refers to studying the quality of the software in the *TRD* group before and after the use of templates.

Between groups

The analysis between groups consists, on the one hand, of analyzing the *TRD* and *noTRD* groups during projects 1, 3 and 4; and on the other hand, analyzing the *TRD* and *noTRD* groups during projects 5 to 8.

Due to the difficulty of project 2 compared with the rest of the projects, we decided not to include this project's data in the analysis.

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, and thus establishing the experimental frame. For this analysis, we defined the following hypothesis of investigation:

H1.0: Median (Def. density in UT of noTRD) = Median (Def. density in UT of TRD)

H1.1: Median (Def. density in UT of noTRD) = > Median (Def. density in

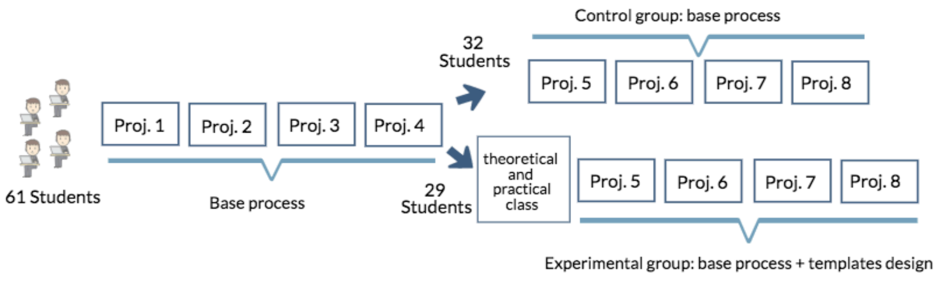


Figure 3. Experimental design

UT of TRD)

Each sample corresponds to the average defect density in UT of a student considering projects 1, 3 and 4.

$$\frac{1000 * \sum_{n=1}^{4} \#defectsUT_n}{\sum_{n=1}^{4} \#LOC_n} \quad (1)$$

where n varies between 1, 3 and 4.

During the analysis, we detected that the data from a student of the *TRD* group was not accurate, that is, that the process followed had not been accurately recorded. So, data from that student was eliminated from the analysis and then there were 28 students remaining as part of the *TRD* group.

The descriptive statistics of the *TRD* and *noTRD* groups considering projects 1, 3 and 4 are presented on Table 1.

The values of the mean and interquartile range indicate there seems not to be great variability between the groups. To confirm this, we applied the Mann-Whitney test for independent samples, since they correspond to different students.

Table 1. Mean and interquartile range in projects 1, 3 and 4

	Mean	Interquartile range
TRD	30.22	25.54
noTRD	32.88	28.9

The result indicates a p-value = 0.3467, with which we cannot reject the null hypothesis (significance = 0.05). This result does not allow us to affirm that there is a difference in quality between *TRD* and *noTRD* groups. We can assert that both groups have a similar or homogeneous behavior. This gives us more confidence to study the software quality between the *TRD* and *noTRD* groups after the use of templates eliminating the possibility that the result is due to the behavior of the groups rather than to using or not using templates.

Studying the *TRD* and *noTRD* groups during projects 5 to 8 aims to know if representing the design using templates has some effect in the software quality. For the analysis between groups during projects 5 to 8, we defined the following hypothesis of investigation:

H1.0: Median (Def. density in UT of noTRD) = Median (Def. density in UT of TRD)

H1.1: Median (Def. density in UT of noTRD) = > Median (Def. density in UT of TRD)

Table 2 presents the average defect density in UT for the 28 students of the *TRD* group and the 32 students of the *noTRD* group in projects 5 to 8.

The values of the mean and of the interquartile shown in Table 3 indicate low variability of the groups. That is to say, the use of templates by the *TRD* group does not produce a significant difference in the defect density compared to *noTRD* group not using templates.

To study the behavior of both groups we used hypothesis tests. The samples are different because they correspond to different students, thus, the Mann-Whitney test is applied.

Results indicate p-value = 0.165, therefore, the null hypothesis cannot be rejected. Thus, we cannot affirm that the students who use the templates manage to develop software with fewer UT defect density than students who do not use templates.

Intra groups

As already mentioned, intra groups refers to knowing if students of *TRD* group improve the software quality after the use of templates to prepare the design. To know this, the defect density in UT from the *TRD* group is analyzed in projects 1 to 4 (without project 2) and projects 5 to 8. Studying the behavior of the same group allows to know if there is a change in the software quality after the use of templates.

We define the following research hypotheses:

H1.0: Median (Def. density in UT of TRD134) = Median (Def. density in UT of TRD58)

H1.1: Median (Def. density in UT of TRD134) = > Median (Def. density in UT of TRD58)

being *TRD134* are the students of *TRD* group during projects 1, 3 and 4; and *TRD58* are the same students of *TRD* group during projects 5 to 8.

Table 4 presents the defect density in UT for the students of the *TRD* group in projects 1, 3 and 4, and the same students in projects 5 to 8.

The descriptive statistics presented in Table 5 indicate some variability in defect density. Even though the mean is similar, it seems that using templates (after project 5) to represent the design achieves products with less defects.

To statistically study the data, we applied the Wilcoxon test (*signed rank test*) for paired samples (because for this analysis the data come from the same students). Results indicate a value of V = 138 and p-value = 0.1438. Since p-value is higher than 0.05 (value of significance) it is not possible

Table 2. Average defect density in UT for the students of the *TRD* group and *noTRD* group in projects 5 to 8

Group	Student	Defect density	Group	Student	Defect density
TRD	1	8.83	*noTRD*	1	27.98
TRD	2	23.16	*noTRD*	2	24.86
TRD	3	33.78	*noTRD*	3	23.59
TRD	4	40.76	*noTRD*	4	14.35
TRD	5	83.33	*noTRD*	5	21.37
TRD	6	16.10	*noTRD*	6	12.19
TRD	7	5.74	*noTRD*	7	22.79
TRD	8	13.02	*noTRD*	8	43.33
TRD	9	28.07	*noTRD*	9	27.02
TRD	10	12.5	*noTRD*	10	36.46
TRD	11	9.49	*noTRD*	11	38.98
TRD	12	19.70	*noTRD*	12	16.80
TRD	13	11.70	*noTRD*	13	37.65
TRD	14	36.85	*noTRD*	14	18.93
TRD	15	20.53	*noTRD*	15	18.25
TRD	16	22.93	*noTRD*	16	22.98
TRD	17	11.80	*noTRD*	17	47.12
TRD	18	37.45	*noTRD*	18	30.21
TRD	19	26.05	*noTRD*	19	35.03
TRD	20	5.03	*noTRD*	20	27.84
TRD	21	23.35	*noTRD*	21	12.22
TRD	22	17.36	*noTRD*	22	24.57
TRD	23	10.08	*noTRD*	23	15.65
TRD	24	42.75	*noTRD*	24	41.17
TRD	25	33.43	*noTRD*	25	44.89
TRD	26	28.63	*noTRD*	26	20.35
TRD	27	44.02	*noTRD*	27	38.80
TRD	28	23.88	*noTRD*	28	51.54
			noTRD	29	7.85
			noTRD	30	27.89
			noTRD	31	24.24
			noTRD	32	25.49

Table 3. Mean and the interquartile range in projects 5 to 8

	Mean	Interquartile range
TRD	24.65	21.2
noTRD	27.57	16.9

to reject the null hypothesis. This indicates that we cannot affirm that students improve the quality of their software by using design templates.

4.2 Internal Quality

To evaluate the internal quality, we carried out an analysis of those code smells introduced by students when developing the course projects. The aim of this analysis is to investigate if the use of design templates prevents students from incurring into certain code smells. The analysis presented is preliminary and exploratory, seeking to obtain initial results that allow us to generate new research hypotheses.

The code smell types depend on the programming language. As students can choose the language in which they develop their projects, this analysis has to be done taking into account the different languages used. With the aim of doing an initial analysis, and that it added value to our research, the students who developed their projects with Java, C#, C, C++ and Ruby were selected, excluding those developed with PHP and Python. We excluded PHP and Phyton because they do not have many code smells in common with the other languages. If we had added PHP and Python, the number of code smells to analyze would have been reduced too much. So, both languages were excluded for this initial analysis. This left a total of 45 students for the analysis, 19 from 2015, 14 from 2016, and 12 from 2017. Of those 45 students, 21 belong to the *TRD* group (9 in 2015, 6 in 2016 and 6 in 2017) and 24 to the *noTRD* group (10 in 2015, 8 in 2016 and 6 in 2017).

To detect the code smells, the tool SonarQube[1] was used, since it is a free-software tool for a variety of programming languages, which presents constant updates for the community and a wide documentation, among others.

We selected 16 code smell types for the analysis. These are common for the programming languages we chose and are detectable by SonarQube. The code smell types are: 1) statements “if ... else if” must end with the clause “else”; 2) statements “switch”/“case” must not be nested; 3) statements “switch”/“case” must not have too many “case”/“when” clauses; 4) the cognitive complexity of the functions or methods must not be too high; 5) “if” collapsible statements must merge; 6) the “if”, “for”, “while”, “switch” and “try” statements of control flow must not nest too much; 7) the expression must not be too complex; 8) files must not have too many lines of code; 9) functions or methods must not have too many lines of code; 10) functions or methods must not have too many parameters; 11) lines of code must not be too long; 12) functions or methods must not be empty; 13) statements must be in separate lines; 14) two branches in one conditional structure must not have the exact same implementation; 15) the parameters of one function or method not used must be eliminated; 16) the local variables not used must be eliminated. A more detailed description of each one is not provided for article-length reasons.

[1]http://www.sonarqube.org

Table 6 shows the percentage of students that incurred in at least one code smell, segmented by project (from 1 to 8) and by group (*noTRD* and *TRD*). Code smells 3, 8 and 12 are not present in any of the projects analyzed.

When analyzing the table between the *noTRD* and *TRD* groups, as of program 5 (after using templates) a great variability arises, both if it is considered per project as it is considered per code smell.

For code smells 4, 7, 10 and 13, it is observed that a group is better for certain projects, and the other group is better for certain other projects. For code smells 1, 2, 5, 6, 9 and 14, the difference between groups is very little. To sum up, changes after using templates are not observed for any of these code smells.

For the case of code smell 11, a very minor percentage is observed in projects 5 and 7, and a minor percentage in project 8 on the part of the group using templates. In project 6, both groups have almost identical behavior. From the point of view of templates, maybe it is the pseudocode template that is helping the students decrease the introduction of this code smell.

Code smells 15 and 16 show a similar behavior. For both cases, *TRD* group almost does not incur in them, while *noTRD* does and sometimes in a high percentage. Number 15 refers to parameters not used in the methods, and 16 to local variables not used. Clearly, these types of code smells can be avoided with good software design. From the point of view of the use of templates, maybe the development of pseudocode (logic template) and the functional template are preventing the students of the *TRD* group from incurring in these code smells. Anyway, it is necessary to manually analyze the templates submitted by the students and have interviews with them to know better if this can be happening for the reasons already described. This has not been done yet.

However, when analyzing the table, but only considering the data of the *TRD* group throughout the 8 projects, we do not see that the use of templates improves the internal quality.

It is worth noting that this group normally did not incur in code smells 15 and 16 (or did it in a very low percentage). Observing projects 1 to 4 and 5 to 8 separately, we do not see any difference between them. That means, the behavior of this group before using templates and during its usage does not change for these code smells. So, the difference presented in the previous analysis between *TRD* and *noTRD* groups does not seem to respond to the use of templates.

Something similar happens with code smell 11. Results do not show a decrease of this code smell when using templates.

It can be observed that in project 8, the percentage of occurrence of code smells 4, 9 and 10 significantly increases for both groups. This increase makes us think that project 8 is more complex for the students. These three code smells indicate that the code developed is too complex and long for its comprehension. That is, the use of templates did not help the students elaborate a less complex and understandable design.

Putting both analyses together, we conclude that the use of templates does not improve the internal quality. Specifically (or being more precise), the use of templates does not seem to have an effect on the code smells in which the students

Table 4. Defect density in UT for the students of the *TRD* group in projects 1, 3 and 4, and in projects 5 to 8

Group	Student	Defect density 1,3 and 4	Defect density 5 to 8
TRD	1	2.22	8.83
TRD	2	7.22	23.16
TRD	3	35.33	33.78
TRD	4	14.24	40.76
TRD	5	95.74	83.33
TRD	6	17.85	16.10
TRD	7	10.14	5.74
TRD	8	21.18	13.02
TRD	9	15.54	28.07
TRD	10	39.80	12.5
TRD	11	13.79	9.49
TRD	12	18.31	19.70
TRD	13	10.23	11.70
TRD	14	60.60	36.85
TRD	15	32.60	20.53
TRD	16	25.83	22.93
TRD	17	51.09	11.80
TRD	18	48.78	37.45
TRD	19	39.63	26.05
TRD	20	15.56	5.03
TRD	21	30.70	23.35
TRD	22	25.77	17.36
TRD	23	9.72	10.08
TRD	24	32.71	42.75
TRD	25	10.05	33.43
TRD	26	42.70	28.63
TRD	27	16.87	44.02
TRD	28	102.04	23.88

Table 5. Mean and the interquartile range calculator

Project	Mean	Interquartile range
1, 3 y 4	30.22	25.5
5 to 8	24.65	21.2

incur when designing software.

4.3 Effort dedicated to designing and coding

To answer RQ2: "What is the relation between the effort dedicated to designing and the effort dedicated to coding?, Are there any variations in effort when students use templates?" , we analyzed the following hypothesis test:

H2.0: Median (TCOD) $<=$ Median (TDLD)
H2.1: Median (TCOD) $>$ Median (TDLD)

As part of the base process, each student registered the time spent in the design phase (TDLD) and the time spent in the code phase (TCOD) for each project.

To know the effort dedicated to designing and to coding by the group that uses the templates and the group that does not use them, we analyzed both groups independently during projects 5 to 8. That is, on the one hand, we carried out the analysis of the *TRD* group during projects 5 to 8, and on the other hand, of the *noTRD* group during projects 5 to 8.

For each student, we calculated the time spent in design and the time spent in code for projects 5 to 8. The calculation for each pair of data is the following:

$$(\sum_{n=5}^{8} TDLD_n, \sum_{n=5}^{8} TCOD_n) \qquad (2)$$

where $TDLD_n$ is the time spent in the design phase for project n, $TCOD_n$ is the time spent in the code phase for project n, and where n varies from 5 to 8.

Table 7 presents the 28 data pairs (TDLD, TCOD) for the *TRD* group, and the 32 data pairs (TDLD, TCOD) for the *noTRD* group.

Table 8 presents the mean and the interquartile range for the *TRD* group and the *noTRD* group.

The mean value of the *TRD* group shows that the use of templates takes more design time compared with the group that did not use templates. Furthermore, the design time in the case of *TRD* exceeds the time spent on coding.

Regarding the TCOD's mean, even though it is similar in the *TRD* and *noTRD* groups, a decrease in the *TRD* group is observed. Despite the fact that the decrease is not quite significant, the use of templates might have helped coding in less time.

To determine the statistical test that best fits the problem to be solved, the distribution of the data was previously studied. When applying the Kolmogorov-Smirnov test for the *TRD* group, a significance value of 0.00478 is obtained, indicating that the values do not fit a normal distribution. The result of applying Kolmogorov-Smirnov test for the *noTRD* group returns 7.713e-12 as a significance value, for that, the values do not fit a normal distribution.

As the data of both does not follow a normal distribution, Wilcoxon's test is used for paired samples. The samples of each group are paired since the sampled pairs (TDLD, TCOD) correspond to the same student.

We executed the test for the *TRD* group and for the *noTRD* independently.

For the *noTRD* group, we proposed to know the value of X such that TCOD = X*TDLD. We analyzed the following hypothesis test:

*H2.0: Median (TCOD of noTRD) $<=$ Median (X*TDLD of noTRD)*
*H2.1: Median (TCOD of noTRD) $>$ Median (X*TDLD of noTRD)*

When executing the test for the *noTRD* group with X=1, the null hypothesis is rejected (p-value = 4.169e-07, the significance level is taken with a value of 0.05), confirming that the coding time is greater than the designing time. To know how much more or what is the relationship between these times (TCOD = X*TDLD) we applied the test again but now multiplying the TDLD by an integer X value until the null hypothesis cannot be rejected. Table 9 presents the results for the Wilcoxon test.

The results indicate that for X=1, X=2 and X=3 the null hypothesis is rejected, so the coding time is greater than 3 times the design time. For X=4, the null hypothesis cannot be rejected (p-value=0.541). In other words, students who did not use templates generally spent at least 3 times more time on coding than on designing.

In the case of the *TRD* group, the mean value shows that students tend to dedicate more time to design in relation to code. Therefore, we carried out the analysis in an inverse way, calculating X such that: X*TCOD=TDLD. We analyzed the following hypothesis test:

*H2.0: Median (X*TCOD of TRD) $>=$ Median (TDLD of TRD)*
*H2.1: Median (X*TCOD of TRD) $<$ Median (TDLD of TRD)*

When executing Wilcoxon test for the *TRD* group with X=1, the null hypothesis is rejected (p-value = 0.0007155), confirming that the coding time is less than the designing time. To know how many times more students spent in designing, we applied the test again but now multiplying the TCOD by an integer X value until the null hypothesis cannot be rejected.

Table 10 presents the results of the Wilcoxon test applied to *TRD* group.

The results indicate that for X=2 the null hypothesis cannot be rejected (p-value = 0.998). So, students who use templates spend more time designing than coding, but not double.

This result indicates that the group that used templates dedicated a greater effort to design than the group that did not use templates. To confirm that the relationship between designing time and coding time previously obtained by the *TRD* group is due to the use of templates and not to another factor dependent on the group, we studied the relationship (TCOD, TDLD) but in this case during projects 1, 3 and 4 (without using templates).

Table 11 presents the mean and the interquartile range of the pairs (TDLD, TCOD) for the *TRD* group in projects 1, 3 and 4.

The values of the descriptive statistics of the *TRD* group in projects 1, 3 and 4 are similar to those of the *noTRD* group. In other words, during projects in which students design with-

Table 6. Percentage of students who incur at least one code smell by code smell type and student group

Code smell	Group	Project							
		1	2	3	4	5	6	7	8
1	*noTRD*	4%	29%	0%	4%	13%	13%	4%	13%
	TRD	19%	19%	10%	0%	5%	5%	5%	5%
2	*noTRD*	0%	0%	0%	0%	0%	0%	0%	0%
	TRD	0%	0%	0%	0%	0%	0%	0%	5%
4	*noTRD*	8%	58%	0%	13%	30%	46%	29%	50%
	TRD	24%	43%	5%	10%	10%	43%	24%	95%
5	*noTRD*	4%	21%	0%	0%	0%	0%	0%	0%
	TRD	0%	24%	10%	0%	0%	5%	0%	5%
6	*noTRD*	13%	63%	8%	29%	30%	38%	13%	42%
	TRD	38%	67%	29%	29%	33%	52%	57%	62%
7	*noTRD*	0%	25%	0%	0%	0%	4%	8%	0%
	TRD	0%	19%	0%	0%	0%	5%	0%	5%
9	*noTRD*	0%	4%	8%	17%	10%	21%	21%	67%
	TRD	0%	10%	19%	14%	10%	29%	38%	71%
10	*noTRD*	0%	0%	0%	0%	0%	0%	8%	54%
	TRD	0%	0%	5%	0%	0%	0%	19%	38%
11	*noTRD*	4%	46%	42%	8%	40%	4%	46%	75%
	TRD	0%	29%	29%	0%	14%	5%	24%	62%
13	*noTRD*	0%	0%	0%	0%	10%	0%	0%	4%
	TRD	5%	0%	5%	0%	0%	0%	5%	19%
14	*noTRD*	0%	8%	0%	0%	10%	0%	0%	0%
	TRD	0%	0%	0%	0%	0%	0%	0%	0%
15	*noTRD*	0%	0%	8%	4%	20%	0%	13%	17%
	TRD	0%	0%	0%	0%	5%	0%	0%	0%
16	*noTRD*	8%	13%	8%	8%	40%	8%	17%	29%
	TRD	5%	5%	10%	10%	0%	0%	10%	10%

Table 7. Data pairs for the *TRD* group and the *noTRD* group

TRD group		*noTRD group*	
TDLD	TCOD	TDLD	TCOD
178	263	60	172
748	217	44	369
940	621	51	446
522	249	63	350
178	61	16	245
204	221	53	302
163	371	100	427
295	212	67	289
665	265	64	243
175	272	23	464
626	329	31	350
407	169	65	460
757	407	23	248
238	228	18	184
392	269	132	347
288	249	163	225
212	210	140	197
278	150	116	354
573	274	69	205
518	199	33	229
336	398	193	226
453	108	58	329
401	222	103	206
330	360	83	168
515	493	43	241
327	242	92	187
160	169	21	481
296	213	107	304
		35	236
		205	468
		64	224
		168	194

Table 8. Mean and the interquartile range for *noTRD* and *TRD* groups

Group		Mean	Interquartile range
TRD	TDLD	399.1	287.5
TRD	TCOD	265.7	26.7
noTRD	TDLD	19.5	15.7
noTRD	TCOD	292.8	132

Table 9. Wilcoxon test for the *noTRD* group in projects 5 to 8

X=1	X=2	X=3	X=4
4.169e-07	4.088e-05	0.03861	0.541

Table 10. Wilcoxon test for the *TRD* group in projects 5 to 8

X=1	X=2
0.0007155	0.998

Table 11. Mean and the interquartile range of the pairs (TDLD, TCOD) for the *TRD* group in projects 1, 3 and 4

	Mean	Interquartile range
TDLD	43	41.5
TCOD	242	118

out using templates, the time spent on design is significantly less than the time spent on coding.

Table 12 presents the results of executing Wilcoxon's test to analyze the relation TCOD = X*TDLD of the *TRD* group in projects 1, 3 and 4.

Table 12. Wilcoxon test for the *TRD* group in projects 1, 3 and 4

X=1	X=2	X=3	X=4	X=5
3.725e-09	3.725e-08	0.0002701	0.01245	0.09678

The results indicate that for X=5, the null hypothesis cannot be rejected (p-value = 0.09678). Students of *TRD* group in projects 1, 3 and 4 generally spent at least 4 times more time on coding than on designing. This result shows that there is an increase in the time dedicated to design after the students of the *TRD* group begin to use design templates.

5 Discussion

In the context of our experiment, we found that design representation using templates produced an increase in time spent designing (we were expecting this). However, it did not help to develop better-quality software products, nor from an internal point of view, neither from an external point of view.

Results show that the use of templates did not improve neither the number of defects the developed code has (measured as defects density in UT), nor the internal quality (measured as the number of code smells in the code). These results are related to those reported by Gravino (Gravino et al., 2015), where the use of UML diagrams did not achieve any improvement in the comprehension of the source code vis-à-vis not using them.

In addition, the analysis of the relation between effort dedicated to coding and effort dedicated to designing showed that the use of templates produced an increase in design time. Students who did not use the templates tended to spent 3 times more on code than on design. Students who use templates spent more time designing than coding. Moreover, students in both groups spent similar time in coding and before using templates the students in *TRD* group behave similar to *noTRD* group.

We can conclude then, that using templates to represent design increases the effort dedicated to design but does not have a significant positive effect on quality or in reducing coding time. This can be due to several factors that we must analyze in the future. It could be, among other reasons, that students are not used to these templates and so they did not get the expected benefit; it could be that they just filled the templates but, in that moment, they did not care to think or develop a quality system; it could be that students do not know how to design (as found in other studies); or as mentioned by Chaiyo (Chaiyo and Ramingwong, 2013), it could be that the templates are difficult to use by students.

We believe that students do not have the habit of designing and thinking of a solution before coding. Although we think that the use of templates would be helpful, we believe that the students filled them in to achieve the goal without thinking of a design solution. Rather, we believe that the usual student practice is code-and-fix. Even though more analysis is needed, we agree with several authors on the fact that graduating students have difficulties to design and they do not seem to understand what type of information to include to design software (Eckerdal et al., 2006a,b; Loftus et al., 2011).

6 Threats to validity

Most empirical studies are threatened by the way research is conducted (Wohlin et al., 2012). This section describes the threats to validity we have detected.

Internal validity threats: Investigating with students involves several threats. On the one hand, the fact that the context of the experiment is a course implies that the students does not develop naturally. We tried to minimize this threat with a non-graded course, that is, the student approved or failed. Besides, we remarked 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, or efforts made.

On the other hand, there is a threat that students share information or solutions to projects. In this sense, the assigned teachers reviewed the submissions and compared them between students to ensure there were no duplicate submissions.

In addition, students carry out their projects at home, which causes limited control by teachers. To reduce this threat, we introduced supervision, corrections, and feedback between the student and the assigned teacher.

Besides, for the analysis, we did a data aggregation of the three courses, knowing that the different courses can have influence on the data collected for being a hierarchical model. We tried to reduce this threat through the use of a defined and disciplined process the students followed, and keeping the same material and the same teachers throughout the three courses.

External validity threats: experimenting with students of a course has the advantage that they are available and are willing to participate in experiments, and the disadvantage that their characteristics cannot be generalized. In our experiment, students took part of the PF-PSP course voluntarily and did not know that they were part of an experiment until they finished the course. This reduces to the minimum the bias they might have when feeling part of a research. Conversely, the results obtained in this experiment cannot be generalized to the students practice of design in other contexts.

Construct validity threats: this kind of threat is related to the way in which the response variables were measured. In our experiment, we measured effort as the time in minutes that the student spends on the phase and the quality as the number of defects in UT and the number of code smells in which students incur. To ensure a correct data recording, we used a data recording tool and framework that allows a disciplined and measurable process to be followed.

Conclusion validity threats: The number of students in the research constitutes a threat to the statistical conclusion. 61 students participated during the three replications. This causes the statistical analysis to be carried out using non-parametric tests whose statistical power is lower than the parametric tests. As a measure to this threat, we completed the non-parametric tests with descriptive statistics.

7 Conclusions

This work is one step further towards the understanding of the software design practice. The results of our experiment show that graduating students do not improve the software quality when using templates for design representation. However, using templates produces a significant increase in the time spent on the design phase without reducing coding time.

We analyzed the software quality from the internal and external points of view, and from the effort dedicated to design. On the one hand, we statistically proved that using templates for design representation does not improve the external software quality, measured as the defect density in unit testing.

From the internal quality perspective, the use of templates does not have a significant positive effect on the code smells in which students incur when designing software.

Regarding the effort, students who used templates dedicate a greater effort to designing than to coding (which is not double). Meanwhile, students that did not use templates dedicated four times less effort to designing than to coding.

Our results are related to those mentioned by Gravino and Torchiano (Gravino et al., 2015; Torchiano et al., 2017), where the use of UML diagrams to design does not make significant improvements in their source code comprehension tasks. Also, regarding effort, students who use diagrams spend twice as much time on the same source code comprehension task than students who do not use them. Gravino analyzes the experience factor, and they find that the most experienced students achieve an improvement in the understanding of the source code (Gravino et al., 2015). Although we did not analyze the experience factor of the graduating students, it could be an analysis to be performed in the future.

Our research focuses on graduating students, most of them working in the Uruguayan software industry as junior engineers. These engineers usually perform programming tasks, which include low-level design. The results obtained in our experiment cannot be generalized to all junior developers and even less to senior developers.

Our results raises new questions about the practice of software design: What do students usually design? What kind of information do they include when designing? Is it possible for them to produce their designs mentally, without representing them? Do they know the effect of a good design in software quality?

Continuing with this line of research, in 2018, we executed an experiment that sought to know how students usually design. Students performed the same 8 projects during this experiment and delivered the design representation made in a natural way (without templates). Although we have not yet finalized the data analysis, we have found that students do not deliver complete designs in a preliminary analysis. In general, they use informal/natural language and incomplete class diagrams in a few cases. Studying the students' habitual behavior when designing software should help identify potential problems in the design practices and find better ways of teaching skills for developing quality software. In 2019 and 2020, no experiments could be performed, but in 2021 we are replicating the 2019 experiment to have more data. As future work, we will finish the above-mentioned analysis to identify potential problems in the design practices and find better ways of teaching skills for developing quality software. Also, we plan to analyze the designs produced with the templates to know what students design and conduct interviews with students to know their experience using templates.

On the other hand, we find it interesting to experiment with some simple MDD tool to know the effect on software quality.

References

Arisholm, E., Briand, L. C., Hove, S. E., and Labiche, Y. (2006). The impact of uml documentation on software maintenance: an experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6).

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., and Penix, J. (2008). Using static analysis to find bugs. *IEEE software*, 25(5).

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

Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Budgen, D., Burn, A. J., Brereton, O. P., Kitchenham, B. A., and Pretorius, R. (2011). Empirical evidence about the uml: a systematic literature review. *Software: Practice and Experience*, 41(4):363–392.

Campbell, G. A. and Papapetrou, P. P. (2013). *SonarQube in Action*. Manning Publications Co, 2013 version edition.

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

Chaiyo, Y. and Ramingwong, S. (2013). The development of a design tool for personal software process (psp). In *10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 1–4.

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.

Fernández-Sáez, A., Genero, M., and Chaudron, M. (2013). Empirical studies concerning the maintenance of uml diagrams and their use in the maintenance of code: A systematic mapping study. *Information and Software Technology*, 55:1119–1142.

Flores, P. and Medinilla, N. (2017). Conceptions of the students around object-oriented design: A case study. In *XII Jornadas Iberoamericanas de Ingenieria de Software e Ingeniería del Conocimiento*.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Gibbon, C. A. (1997). *Heuristics for object-oriented design*. PhD thesis, University of Nottingham.

Gopichand, M., Swetha, V., and Ananda Rao, A. (2010). Software defect detection and process improvement using personal software process data. In *International Conference on Communication Control and Computing Technologies*, pages 794–799.

Gravino, C., Scanniello, G., and Tortora, G. (2015). Source-code comprehension tasks supported by uml design models: Results from a controlled experiment and a differentiated replication. *Journal of Visual Languages & Computing*, 28:23 – 38.

Grazioli, F. and Nichols, W. (2012). A cross course analysis of product quality improvement with psp. In *Team Software Process Symposium 2012*, pages 76–89.

Grazioli, F., Nichols, W., and Vallespir, D. (2014a). An analysis of student performance during the introduction of the psp: An empirical cross-course comparison. In *Team Software Process Symposium 2013*, pages 11–21.

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

Group, S. (2015). The chaos report. *The Astrophysical Journal Supplement Series*.

Hayes, W. and Over, J. (1997). The personal software process (psp): An empirical study of the impact of psp on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

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

Humphrey, W. (2005). *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional.

Humphrey, W. S. (1995). *A discipline for software engineering*. Addison-Wesley Longman Publishing Co., Inc.

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.

Jones, B. and Kenward, M. G. (2014). *Design and Analysis of Cross-Over Trials*. Chapman and Hall/CRC, 3rd edition.

Karasneh, B., Jolak, R., and Chaudron, M. R. V. (2015). Using examples for teaching software design: An experiment using a repository of uml class diagrams. In *2015 Asia-Pacific Software Engineering Conference*.

Kitchenham, B. and Pfleeger, S. L. (1996). Software quality: the elusive target. *IEEE Software*, 13(1):12–21.

Kramer, J. (2007). Is abstraction the key to computing? *Com-*

mun. 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.

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

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*.

Nistala, P., Nori, K. V., and Reddy, R. (2019). Software quality models: A systematic mapping study. In *2019 IEEE/ACM International Conference on Software and System Processes*, pages 125–134.

Panach, J. I., Dieste, O., Marín, B., España, S., Vegas, S., Pastor, O., and Juristo, N. (2021). Evaluating model-driven development claims with respect to quality: A family of experiments. *IEEE Transactions on Software Engineering*, 47(1):130–145.

Petre, M. (2013). Uml in practice. *International Conference on Software Engineeringn*, 35.

Pierce, K., Deneen, L., and Shute, G. (1991). Teaching software design in the freshman year. In *Software Engineering Education*. Springer Berlin Heidelberg.

Prechelt, L. and Unger, B. (2001). An experiment measuring the effects of personal software process (psp) training. *IEEE Transactions on Software Engineering*, 27(5):465–472.

Senn, S. (2002). *Cross-over Trials In Clinical Research*. John Wiley & Sons, Ltd, 2nd edition.

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.

Soh, Z., Sharafi, Z., Van den Plas, B., Cepeda Porras, G., Guéhéneuc, Y.-G., and Antoniol, G. (2012). Professional status and expertise for uml class diagram comprehension: An empirical study. In *IEEE International Conference on Program Comprehension*.

Sommerville, I. (2016). *Software Engineering*. Pearson.

Stevenson, J. and Wood, M. (2018). Recognising object-oriented software design quality: a practitioner-based questionnaire survey. *Software Quality Journal*, 26.

Taylor, R. N. (2011). Conference welcome message. In *Proc. 33rd International Conference on Software Engineering*. Association for Computing Machinery.

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

Torchiano, M., Scanniello, G., Ricca, F., Reggio, G., and Leotta, M. (2017). Do uml object diagrams affect design comprehensibility? results from a family of four controlled experiments. *Journal of Visual Languages & Computing*, 41.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.