UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

The Practice of Software Detailed Design of Graduating Students: A family of experiments

MSc. Silvana Moreno

Programa de Posgrado en Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Agosto de 2022

The Practice of Software Detailed Design of Graduating Students: A family of experiments

MSc. Silvana Moreno

Examen de Calificación de Doctorado presentado al Programa de doctorado en informática de PEDECIBA, Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Doctor en Informática.

Directores:
Dr. Prof. Diego Vallespir
Dr. Prof. Martín Solari

Director académico:
Dr. Prof. Álvaro Tasistro

Montevideo – Uruguay
Agosto de 2022

Moreno, MSc. Silvana

The Practice of Software Detailed Design of Graduating Students: A family of experiments / MSc. Silvana Moreno. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2022.

XII, 104 p.: il.; 29, 7cm.

Directores:

Diego Vallespir

Martín Solari

Director académico:

Álvaro Tasistro

Programa de doctorado en informática de PEDECIBA, 2022.

Referencias bibliográficas: p. 97 – 104.

1. software detailed design, 2. graduating students, 3. family of experiments. I. Vallespir, Diego, Solari, Martín, . II. Universidad de la República, Programa de Posgrado en Ingeniería en Computación. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE PROGRAMA DE DOCTORADO EN INFORMÁTICA

D.Sc. Prof. Nombre del 1er Examinador Apellido

Ph.D. Prof. Nombre del 2do Examinador Apellido

D.Sc. Prof. Nombre del 3er Examinador Apellido

Ph.D. Prof. Nombre del 4to Examinador Apellido

Ph.D. Prof. Nombre del 5to Examinador Apellido

Montevideo – Uruguay
Agosto de 2022

Acknowledgments

I want to thank my tutors, Diego Vallespir and Martín Solari, who supported and guided me throughout my thesis work, and Álvaro Tasistro for being my academic director.

I thank those who collaborated with me in the PSP Principles and Fundamentals course and in the research projects carried out throughout my Ph.D. work: Diego Vallespir, Leticia Pérez, Vanessa Cassella, Patsy Jones, Guillermo Kuster, and Guillermo Tavidian.

I would also like to thank the Software Engineering Group, in particular, Sebastián Pizard and Cecilia Apa, with whom I could talk and discuss my work on several occasions.

To Angélica Aldecoa, my translator and English teacher who helped me in the translation of this thesis and in the articles we have published.

I thank my family, Nico, Luana, and my in-laws for always being available, understanding, and supportive throughout this time.

ABSTRACT

Software design is one of the essential components to ensure the success of a software system. It 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.

Several authors consider 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. Normally, 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.

The general objective of this thesis is to contribute to the knowledge of how graduating students practice software detailed design. Specifically, we conducted a family of experiments in the context of a course at the School of Engineering of Universidad de la República, in Uruguay. The family of experiments is composed of 3 sub-families of experiments: Baseline, Template, and Habitual. **Baseline** experiments are made up of an initial experiment (executed in 2012) and two replicates (executed in 2013 and 2014). **Template** experiments are made up of an initial experiment (executed in 2015) and two replicates (executed in 2016 and 2017). Finally, **Habitual** experiments consist of an initial experiment (executed in 2018) with one replicate (executed in 2021). Through the family of experiments, we studied the detailed design practice of graduating students from different points of view.

The results indicate that our graduating students do not take time to think of a solution (design) before coding. They spend at least three times less time designing than coding. In addition, we introduced design templates with the intention that they would be a tool to help them in the design task. However, although the design time significantly increases, the quality of the software, measured in the unit testing phase, does not change. Lastly, the analysis of the design representation delivered by the students reveals simple, basic designs

with little elaboration.

Keywords:
software detailed design, graduating students, family of experiments.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

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. Software design involves two stages between the analysis of the requirements and software construction: the architectural design and the detailed design (Bourque and Fairley, 2014).

The architectural design stage is responsible for decomposing requirement specifications to form a system structure. It emphasizes the module-level system representations, which can be evaluated, refined, or modified in the early software development process.

The detailed design stage is responsible for transforming the system structure produced by the architectural design stage into a procedural description of a software system. This stage emphasizes the selection and evaluation of algorithms to implement each module. At this stage, each module's details and decisions are well defined and can be easily implemented (Bourque and Fairley, 2014). Software detailed design involves identifying and abstractly describing the software system and its relationships. Detailed 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).

Developing high-quality software is one of the objectives of software engineering. To achieve such an objective, engineering professionals should use methods, methodologies, and tools to improve the performance and quality during the software development process.

Software design has a direct effect on software quality. The choice of a design can affect (positively or negatively) different properties of the software quality (Prabha and Shivakumar, 2020). Good designs help develop robust, maintainable software with few critical defects (Pierce et al., 1991; Sommerville, 2016; McDonald et al., 2007). The cost of repairing defects increases exponentially as the software develops through the development life cycle. The cost of fixing defects after release is up to 30 times more expensive than catching them in the design and architecture phase (Oktafiani and Hendradjaya, 2018). Also, there is an essential link between testability (ability to test systems and their components) and good design. Easy testability is tightly correlated to loose coupling and strong cohesion (Whalley and Kasto, 2014).

Universities should train students of software engineering to be prepared, among other things, to design software. Students' ability to build a good design is related to their 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 not appreciate their similarities, differences, and their uses (Carrington, 1998).

Research shows that students have 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).

Several studies found that students do not manage to produce a good software detailed design (Sien, 2011; Eckerdal et al., 2006a,b; Loftus et al., 2011; Tenenberg, 2005). 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).

Unfortunately, the evidence found on students' detailed design practice focuses primarily on how students represent software designs. The main focus of this thesis is the study, through a family of experiments, of the detailed

design practice of graduating students from different points of view.

1.1 Research goal and questions

The main objective of this thesis is to study how graduating students practice detailed design.

To achieve the stated goal, this thesis aims to answer the following General Research Questions (GRQs):

> GRQ1: How much effort do students spend on software detailed design?

By answering this question, we intend to provide evidence that relates to the time students spend on detailed design and code.

> GRQ2: What is the effect of detailed design on software quality?

Answering this question lets us know if the design representation delivery contributes to better quality software products.

> GRQ3: How do students represent software detailed design?

Finally, with GRQ3, we want to know what artifacts and ways of design representation the students habitually use.

To answer the GRQs, we carried out a family of experiments.

1.2 Research methods overview

An experiment is a formal, rigorous, and controlled investigation. In an experiment, key variables are identified and manipulated (Wohlin et al., 2012). The variables under study take different values, and the experimental research aims to determine the effects of such variations. During these investigations, quantitative data is collected, and then statistical methods are applied. Experiments are proposed when we want a controlled situation and want to manipulate be-

havior directly, precisely, and systematically. Also, experiments involve more than one treatment to compare the outcomes (Wohlin et al., 2012).

When conducting a controlled experiment, we want to study the outcome when we vary some input variables to a process. An experiment has two types of variables: independent and response variables (Wohlin et al., 2012). The response variables are those that we want to study to see the effect of the changes in the independent variables. All variables in a process that are manipulated and controlled are called independent variables.

An experiment studies the effect of changing one or more independent variables. Those variables are called factors. The other independent variables are controlled at a fixed level during the experiment, or else we cannot say if the factor or another variable causes the effect. A treatment is one particular value of a factor. The people that apply the treatment are called subjects. The characteristics of both the objects and the subjects can be independent variables in the experiment (Wohlin et al., 2012). The experiment process can be divided into the following steps:

- Definition
- Planning
- Operation
- Analysis and interpretation
- Presentation and package

During definition, the experiment is established in terms of the problem to be solved, the objective, and the goals. In the planning step, where the experiment design is determined, the instrumentation is considered and the threats to the experiment are evaluated. The operation of the experiment follows from the design. In the operational phase, measurements are collected and then analyzed and evaluated in the analysis and interpretation. Lastly, the results are presented and packaged in the presentation and package step.

Experimental results must be extensively verified to consolidate a body of knowledge built upon evidence. Experiments need replication at other times and under other conditions before they can produce an established piece of knowledge. Experimental replications are necessary to strengthen the evidence (Juristo, 2013). Without replication, it is difficult to distinguish between chance results (occurred accidentally) and results that, in reality, do exist (Juristo, 2016). The replication of an experiment aims to repeat the experiment

under the most similar conditions possible. Otherwise, it may be run by varying one or more parameters of the original experiment (Juristo and Moreno, 2001).

A family of experiments is a set of experimental replications that pursue the same goal and whose results can be combined into common findings as those that can be achieved in isolated experiments (Basili et al., 1999). Replication aims to provide a family of experiments to aggregate separate experiments and get more reliable results, as well as to analyse aspects that individual experiments have overlooked, providing accurate information for decision making and more in-depth knowledge of the issue under investigation (Santos et al., 2020).

To achive our research goal, we conducted a family of experiments composed of sub-families of experiments. The family of experiments' main objective is to know how graduating students practice detailed design.

Each sub-family's objectives, research questions and experimental designs evolved as we were executing and analyzing the results of the previous sub-family(s).

The family of experiments is composed of 3 sub-families: Baseline sub-family, Template sub-family, and Habitual sub-family. The Baseline sub-family of experiments, which we simplify to **Baseline** experiments, is aimed at learning about the effort dedicated to software design by graduating students, how is the effort variation throughout different projects, how is the effort variation between students, and what is the students' perception of the problems they face. Template sub-family of experiments, which we simplify to **Template** experiments, is aimed at knowing about the effect on software quality when students represent the design using a specific set of design templates. Also, we want to know the efforts students dedicate to software design. Finally, the Habitual sub-family of experiments, which we simplify to **Habitual** experiments, finds out how students usually design and the effect on software quality when they deliver the usual design representation to professors.

Baseline experiments are made up of an initial experiment (executed in 2012) and two replicates (executed in 2013 and 2014). **Template** experiments are made up of an initial experiment (executed in 2015) and two replicates (executed in 2016 and 2017). Finally, **Habitual** experiments consist of an initial experiment (executed in 2018) with one replicate (executed in 2021). In the context of this thesis, we do not report the replication of the Habitual exper-

iment because the data collected from this replication has not been analyzed yet.

Figure 1.1 summarizes the family of experiments and the research questions. The red boxes denote the GRQs, and the blue ones represent the empirical studies we conducted to answer the GRQs. Through Baseline experiments and Template experiments, we answer GRQ1: How much effort do students spend on software design? During Template and Habitual experiments, we investigated the quality of the software developed by students (GRQ2). Finally, with the Habitual experiment, we answer GRQ3: How do students represent software?

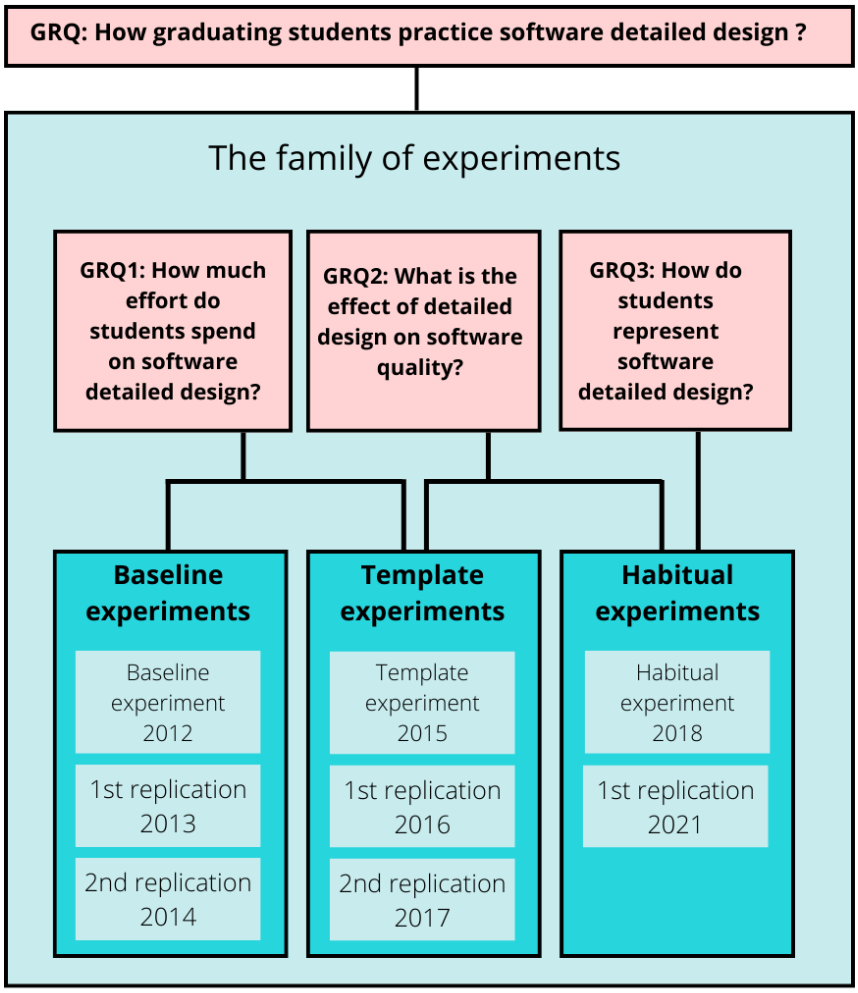


Figure 1.1: Overview of research methods

1.3 Research results overview

The results of this thesis contribute to the knowledge of how graduating students design software. We conducted a family of experiments in the context of an graduating course at the School of Engineering of Universidad de la República, in Uruguay. The results obtained through the family of experiments show that our students have difficulties designing simple exercises.

Baseline experiment results indicate that students do not seem to be aware of the importance of the design phase, spending three times more time on coding than on designing. Considering quality (measured as defect density in unit testing), we found that students did not produce better quality products in the latest exercises compared with the first ones. That is, students did not find ways to improve, neither looking at their products (intermediate as design or final as tests or code), nor looking at their own process.

The results of the **Baseline** experiment led to the introduction of design templates intended to help students with the design task. **Template** experiments revealed results that we did not expect. Although the time spent on design increases significantly, the quality of the software does not improve when students represent the design using templates. Using templates to represent the design does not improve the external quality of the software measured as defect density in unit tests. From the perspective of internal quality, the use of templates does not significantly affect the *code smells* in which students incur when developing software.

Template experiments' results, added to those found in **Baseline** experiment, generated new questions about the practice of software design: What do students habitually design? What kind of information do they include when designing? Is it possible for them to make their designs mentally without representing them? This prompted the latest experiment in the family to know how students design the software. In the **Habitual** experiment, the submission of the habitual design representation does not impact the software quality produced by students. Also, the analysis of the design representation indicates that students (in our School of Engineering) design on, what we call, a basic level. The designs delivered vary between text notations, the incomplete identification of the parts of the system, and the use of static notations to describe the system. Students do not achieve complete designs combining several artifacts to model dynamic and static aspects simultaneously.

1.4 About the thesis document

This document is made up of this introduction and seven chapters. State of the art is presented in chapter 2. Chapter 3 presents the Personal Software Process and the graduating course where the family of experiments is framed. Chapter 4 presents the generalities of the family of experiments and its sub-families. Each sub-family is described in an independent chapter; specifically, chapter 5 describes Baseline experiments; chapter 6 describes Template experiments; and lastly, the Habitual experiment is described in chapter 7. Each of the experiment chapters includes a discussion section. The conclusions and future work are presented in chapter 8.

1.5 Main achievements

This section presents the papers published with the results of this thesis, and the projects that arose during the thesis.

1.5.1 Publications related to this thesis

- Moreno, Silvana; Vallespir, Diego; Solari, Martín; (2022). An experiment on how graduating students represent software designs. XXV Ibero-American Conference on Software Engineering (CIBSE 2022). (Moreno et al., 2022). This paper was selected as one of the best five papers of the conference and invited to submit an extended version of the paper to the Special Issue of the Science of Computer Programming journal.
- Moreno, Silvana; Vallespir, Diego; Solari, Martín; Casella, Vanessa. (2021). Representation of software design using templates: impact on software quality and development effort. Journal of Software Engineering Research and Development (Moreno et al., 2021).
- Moreno, Silvana; Casella, Vanessa; Solari, Martín; Vallespir, Diego. (2020). Detailed design representation using templates and its effect on software quality. XXIII Ibero-American Conference on Software Engineering (CIBSE 2020) (Moreno et al., 2020). Written in Spanish. This paper was selected as one of the best three papers of the conference.
- Moreno, Silvana; Vallespir, Diego. (2018). Are undergraduate students capable of designing software? A study of the relationship between cod-

ing time and design time in software development. XXI Ibero-American Conference on Software Engineering (CIBSE 2018) (Moreno and Vallespir, 2018). Written in Spanish.

My doctoral thesis work started focusing on experimenting with the PSPvdc process. PSPvdc is an adaptation to the Personal Software Development Process (PSP). PSPvdc proposes new phases and activities to support formal methods to produce software verified by construction. PSPvdc was elaborated as part of my master's work, and initially, experiment and applying PSPvdc in the context of an undergraduate course was proposed as doctoral work. During the first two years of the doctorate, courses were planned, changes were analyzed to be able to execute PSPvdc, formal verification tools were studied, and an SLR was executed to know how formal methods are taught in other universities. This initial proposal for a doctoral thesis was rejected because it was not feasible to experiment with PSPvdc in the context of a course at our school. The estimated times involved more than one semester of activities, which hindered the motivation of students and the logic of several courses. The systematic literature review was presented and published at CIBSE 2019.

- Moreno, Silvana; Vallespir, Diego and Tasistro Álvaro. (2019). Teaching of formal methods: evidence of its inclusion in curricula, results, and difficulties. XXII Ibero-American Conference on Software Engineering (CIBSE 2019)(Moreno et al., 2018).

1.5.2 Projects framed to this thesis

This research has several related support projects; in particular, two graduating projects and a master's project were conducted.

Title: How do undergraduate students design software? A systematic literature review.
Program: Master in Software Engineering, School of Engineering, Universidad de la República
Student: Patsy Helen Jones
Advisors: Diego Vallespir and Silvana Moreno

Description: This project provides knowledge about the existence of scientific studies on the teaching and learning of software design by undergraduate and graduating students. This thesis is not yet finished. Written in Spanish.

Title: PSPCode Tool: PSP support tool.
Program: Computer Engineer, School of Engineering, Universidad de la República
Student: Guillermo Kuster and Guillermo Tavidian.
Advisors: Silvana Moreno
Description: This graduation project built a web support tool for the PF-PSP course that allows data collection and recording, and teachers' correction of deliveries and course configuration. The thesis was defended in 2018. Written in Spanish.

Title: PSPCode Tool: extension to the PSP support tool
Program: Computer Engineer, School of Engineering, Universidad de la República
Student: Lía Malvarez and Gustavo Samir
Advisors: Silvana Moreno
Description: This graduation project aims to extend the above mentioned tool, generating automatic corrections, integrating with the R statistical tool, and allowing process adaptations, among others. The project is in progress; it started in March 2022.

Chapter 2

Related work

This chapter presents in section 2.1 relevant concepts of detailed software design and in section 2.2 aspects of teaching detailed design. Lastly, section 2.3 presents the evidence collected from studies with students on detailed software design.

2.1 Detailed software design

Software design is the stage of software development that transforms requirements specifications into a structure suitable for implementation. The design has two stages between the requirements analysis and software construction: the architectural design and the detailed design(Bourque and Fairley, 2014). The architectural design stage is responsible for decomposing requirement specifications to form a system structure. It emphasizes the module-level system representations, which can be evaluated, refined, or modified in the early software development process. The detailed design stage is responsible for transforming the system structure produced by the architectural design stage into a procedural description of a software system. This stage emphasizes the selection and evaluation of algorithms to implement each module. At this stage, all the details and decisions of each module are well defined and can be easily implemented.

Object-oriented design is one of the industry's most widely used design approaches, and one of the subjects usually taught in universities (Flores and Medinilla, 2017). The object-oriented (OO) approach provides a powerful and effective environment for modeling and building complex systems. Object ori-

entation helps a developer achieve a modular, extensible, maintainable, and reusable system (Booch et al., 1999). Object orientation supports modeling solutions at a higher level of abstraction in terms of classes, a hierarchy of related classes (inheritance), an association among classes, and dynamic binding (dynamic polymorphism). Designers must master these concepts to elaborate good-quality programs (Ganesh and Sharma, 2013).

The Unified Modeling Language (UML) was adopted as a standard for OO modeling by the Object Management Group in 1997. UML can be used for visualizing, specifying, constructing, and documenting the artifacts of software systems. The UML notation is helpful for graphically depicting OO models. UML allows designers to represent multiple independent views of a system using a variety of graphical diagrams, such as the use case diagram, class diagram, state diagram, sequence diagram, and collaboration diagram (Object Management Group, 2000).

In 2000, Robert C. Martin introduced the 5 SOLID principles: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle. The SOLID principles' design allows the management of most quality problems of software design. They can provide an understanding of the design to avoid symptoms of bad design, helping to reduce code complexity, maintainability, and error reduction (Martin et al., 2003). Years later, it gave rise to clean code practices. Clean code is a set of design principles in the agile methodologies framework that help produce intuitive and easy-to-modify code.

On the other hand, the existence of a large number of poorly designed systems has resulted in applications that are inefficient and difficult to maintain. Refactoring is becoming a powerful and agile technique to improve existing source code. When refactoring is applied, the design of the written code is being improved (Martin and Beck, 1999). That is, one transforms a design full of bad practices into a good one through simple steps such as moving properties and methods, renaming classes, methods and properties, deleting code, etc. (Saca, 2017). Weaknesses in designs have given rise to the concept of code smells. Code smells are structures in the code that indicate a violation of fundamental design principles and negatively impact design quality. It is a good practice to identify design smells in a software system and apply the appropriate refactoring to eliminate them to avoid an accumulation of technical debt (Martin and Beck, 1999).

2.2 Teaching detailed software design

The teaching of software design encompasses the product of the design activity (the design representation) and the process by which this product is created. Teaching these aspects is fundamental to software engineering and is challenging for educators, as they are not easy to teach. Design is an inherently creative process, and creativity is difficult to teach in any field (Cowling, 2007).

Teaching design through theoretical-practical classes (lectures) continues to be a traditional way of teaching. However, the search for different teaching approaches has evolved with the passing of time and emerging technologies.

Carrington proposes teaching design using open-source tools. He proposes that students start from projects with large amounts of code and design documentation. The goal is for students to gain experience by reading, understanding, and modifying what others have developed (Carrington and K Kim, 2003). This approach is intended to change the traditional one and not have the student focus on building a new design and code.

Linder et al. propose teaching cooperative design (working in groups). The authors consider that collaboration among peers allows new understandings that could not be reached individually (Linder et al., 2006).

Another approach is the teaching of design through pseudocode based models. The use of the P-coder tool allows students to develop their design skills from the top down and through progressive refinement (Armarego and Roy, 2004).

Moreover, it has been several decades since the Object Oriented Paradigm (OOP) and the Unified Modeling Language (UML) were introduced (Booch et al., 1999).

Object-oriented design is one of the most widely used design approaches in the industry and one of the subjects usually taught in universities (Flores and Medinilla, 2017). OO modeling diagrams and languages allow one to model static and dynamic aspects of the system. Teaching this paradigm appears easy because it is independent of a programming language. This makes its basic units (object/class/relationship structures) intuitively easier to teach and understand (Judith and Upchurch, 1993; Ali et al., 2013; Ramollari and Dranidis, 2007).

However, in other 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 significantly improve 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 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 practitioners of this industry, the use of UML continues to be resisted to a certain degree (Stevenson and Wood, 2018). A survey conducted on 50 software practitioners 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).

In recent years there have been many efforts to contribute to the teaching of design. Thevathayan and Hamilton (2017) combines project-based learning with weekly quizzes, tests, and active learning tasks aimed at improving students' performance and motivation when designing software. Other studies use different methods such as Teaching Assistance, Just-in-Time-Teaching, and cooperative learning, generating positive results in teaching and learning design (Stikkolorum et al., 2018; Tao et al., 2015; Magana et al., 2018).

In addition, academia and industry are focusing on the importance of design quality, design practice in the agile context, and evolutionary design processes, including technical debt management. The teaching of clean code practices, reviews, refactoring, Test-Driven Development, Behaviour-Driven Development, and Domain Driven Design contribute to this regard (Brown et al., 1998; Fowler, 2018).

2.3 Experimental works with students on detailed software design

This section presents the evidence collected from studies with students on software detailed design. We present the method applied to search for works and a summary of the results obtained.

2.3.1 Searching method for previous research

The search for related works begins with preliminary identifying papers related to students' software design practice. This search yields a set of 6 research papers that study how students carry out software design. These works, presented in table 2.1, analyze the design representation done by students from a specific requirement. The initial study reported by Tenenberg led to the remaining studies (Tenenberg, 2005). Some of these use Tenenberg's data (Tenenberg, 2005), while others run in their own contexts, comparing, validating and/or refuting the results of previous work.

Table 2.1: Source papers

Article Id.	Article tittle
1	Students designing software: a multi-national, multi-institutional study (Tenenberg, 2005)
2	Can graduating students design software systems? (Eckerdal et al., 2006a)
3	Categorizing student software designs: methods, results, and implications (Eckerdal et al., 2006b)
4	Can graduating students design: revisited (Loftus et al., 2011)
5	Graduating students' designs - Through a phenomenographic lens (Thomas et al., 2014)
6	Can students design software? The answer is more complex than you think (Hu, 2016)

The forward snowballing search strategy was performed from these six papers to identify more recent studies. "Forward snowballing" refers to identifying new papers based on those papers citing the paper being examined (Wohlin, 2014). This strategy was applied in April 2022, using Scopus as search engine, and we selected articles dated after 2017. The six source articles are broadly focused on our topic of interest, so we believe that the forward snowballing search allows us to obtain recent related papers interesting to us concerning the objective of the thesis and, at the same time, limited in number. The result of applying forward snowballing to the six source items returned 127 items.

The screening process applied to the 127 articles resulting from the forward snowballing search strategy consists of the following three stages:

1. Selection of publications by reading title and abstract.
2. Selection of publications by full reading

3. Removal of duplicate publications

Fig. 2.1 presents the results of the forward snowballing strategy process and table 2.2 shows the resulting papers. The forward snowballing strategy finished with four new articles, that is, no more iterations were made from the four selected articles.

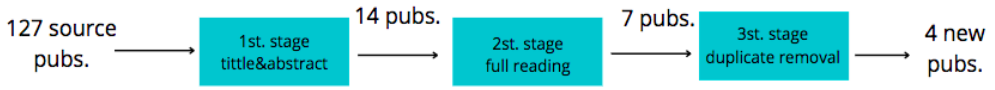


Figure 2.1: Summary of the forward snowballing process

Table 2.2: Papers resulting from the forward snowballing process

Article tittle	Year
How do Graduating Students Evaluate Software Design Diagrams? (Prasad and Iyer, 2020)	2020
Novice Learner Experiences in Software Development: A Study of Freshman Undergraduates (Higgins et al., 2019)	2019
Exploring Software Design Skills of Students in different Stages of their Curriculum (Perez-Gonzalez et al., 2019)	2019
Student software designs at the undergraduate midpoint (Thomas et al., 2017)	2017

Additionally, we consider the results obtained from a systematic literature review (SLR) elaborated by Patsy Jones, Diego Vallespir, and the author of this thesis. This SLR is a central part of Patsy's master's thesis. A systematic literature review is a research method that allows the evaluation and interpretation of all available research relevant to a particular research topic in a subject area or phenomenon of interest (Kitchenham, 2004). The objective of the SLR is to learn about the reported evidence regarding the teaching and learning of software design by graduating and undergraduate students.

A working protocol was defined that allows a conceptual description of each SLR stage in a rigorous and reproducible way for future researchers.

The general research question posed is: How do undergraduate students design software?

The specific research questions proposed are:

1. What are the software design techniques/methods/principles used by undergraduate students in the software designs performed?

2. Is the quality of the software designs made by the undergraduate students evaluated?
3. How is the evaluation of the quality of the software design made by the undergraduate students performed?
4. What ways of software design education are reported?

The protocol was adjusted based on the execution of a pilot. In particular, the initial search string was iteratively adjusted until a good coverage of the research questions was obtained and the number of non-relevant papers was minimized. Table 2.3 presents the finally-defined search string, run in March 2019 in Scopus digital library. We decided to run this article-narrowing string to minimize the time-consuming process of screening a large volume of papers, which largely end up being discarded.

Table 2.3: SLR search string

TITLE (design* AND (student* OR undergrad* OR teach* OR learn* OR train* OR impart* OR comprehend*)) AND (ABS (“software design*” OR “design of software” OR “design software”) OR AUTHKEY (“software design*” OR “design of software” OR “design software”)) AND ABS (student* OR undergrad*) AND LIMIT-TO (LANGUAGE , “English”)

As a result of the application of the string in Scopus, 119 articles were obtained, to which the selection process was applied. The selection process (carried out by Patsy and the author of this thesis) consists of the following three stages:

1. Selection of publications by reading title and abstract
2. Consensus meeting
3. Selection of publications by full reading

Stage 1 is performed independently by Patsy Jones and the author of this thesis. Then, in stage 2, both reviewers unify the selections made in stage 1 in a consensus meeting. Finally, stage 3 is performed only by Patsy Jones. Fig. 2.2 presents a summary of the search and selection process.

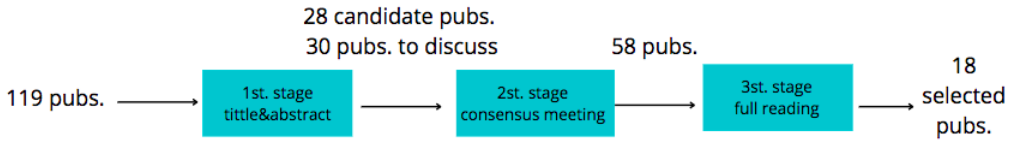


Figure 2.2: Summary of the SLR selection process

As the objectives of the SLR involve experiences regarding software design education and teaching that are not covered in this thesis, we decided to perform another filtering of the 18 articles resulting from the SLR. This filtering is performed by the author of this thesis and aims to select only those papers that focus on the software design practice of undergraduate students.

This work involved selecting articles by full reading, data extraction, and synthesis. Of the eighteen papers resulting from the SLR, eleven were discarded. Table 2.4 presents the seven papers resulting from the filtering.

Table 2.4: Papers resulting from the SLR filtering

Article tittle
Uncovering Students' Common Difficulties and Strategies During a Class Diagram Design Process: an Online Experiment (Stikkolorum et al., 2015)
Student Software Designs at the Undergraduate Midpoint (Thomas et al., 2017)
A Study of the Use of a Reflective Activity to Improve Students' Software Design Capabilities (Coffey, 2017)
Can students design software? the answer is more complex than you think (Hu, 2016)
Graduating students' designs: Through a phenomenographic lens (Thomas et al., 2014)
Can graduating students design: Revisited (Loftus et al., 2011)
Conceptions of the students around object-oriented design: A case study (Flores and Medinilla, 2017)

Fig. 2.3 presents the process to collect the evidence reported from studies with students on detailed software design. A total of 17 papers were obtained from this process. Their extraction and synthesis are presented in the next section.

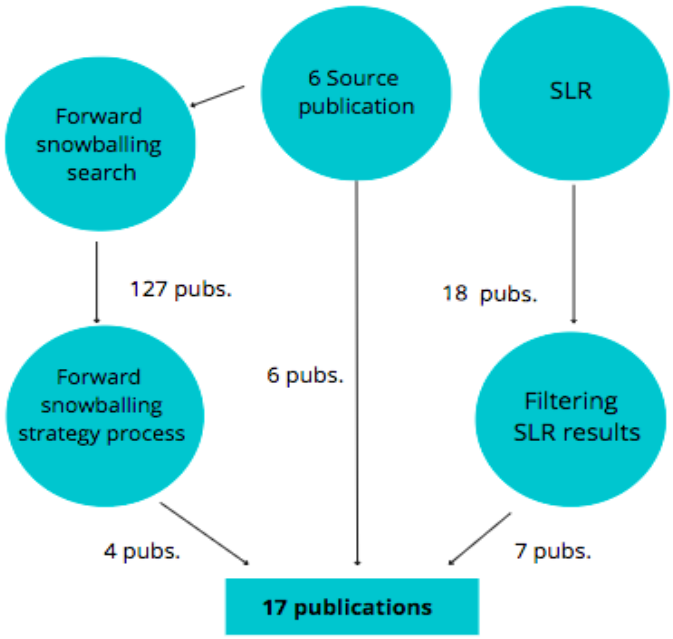


Figure 2.3: Process followed to collect the evidence reported

2.3.2 Synthesis of the works obtained related to how undergraduate students design software

Undergraduate students' design skills are reported by previous studies examining artifacts produced by them to learn how they design software (Eckerdal et al., 2006a,b; Thomas et al., 2014, 2017; Loftus et al., 2011; Tenenberg, 2005; Hu, 2016; Perez-Gonzalez et al., 2019; Stikkolorum et al., 2015). The studies use different approaches: designs produced individually, designs made in groups, and designs produced at different levels of training.

A study reported by Tenenberg describes the skills that students have when designing software (Tenenberg, 2005). They conduct a multi-institutional and multi-national study of 314 Computer Science (CS) students and educators from 21 institutions in 4 countries. The participants are recruited from 21 institutions of post-secondary education in the USA, UK, Sweden, and New Zealand. Three types of participants were represented within the study population: 136 first competency students (FC), 150 graduating students (GS), and 28 educators (E).

Each participant is given the specification of a "super alarm clock" (that helps students manage their sleep patterns) to produce a design. Students

must develop a design and divide it into not less than two and not more than ten parts, giving each a name and adding a short description of what it is and what it does - in short, why it is a part. At the end they must present their written representation, the number of parts in their design, the name for each part and the time they dedicated to developing the design.

Each design artifact was examined and categorized into one of the next categorizations based on its predominating characteristic:

- Standard Graphical: This was used to include recognized notations of software design. Different types were represented in the corpus: Architecture Diagram, Class Diagram, Class-Responsibility-Collaborator (CRC) Cards, Data Flow Diagram (DFD), Entity-Relationship Diagram (ER), Flowchart, Graphical User Interface (GUI), Sequence Diagram, State Transition Diagram (STD) and Use Case Diagram.
- Ad-hoc Graphical: This category included diagrams of any form not recognized as standard notations of software design. Large sections of text were accepted in this category providing that they were considered refinements of items identified in the diagram.
- Code or pseudo-code: This was used for any software design that included code segments such as assignments, iteration and selection.
- Textual: This category was used for free text descriptions but allowed an occasional diagram used for illustration: for example, graphical interface or report layout.
- Mixed: This was used when there was no clear dominance between different styles. For example a participant might start with a textual description and then proceed with a Class Diagram. If there was no connection between the descriptions and the identified classes then the category was Mixed (Text and Class Diagrams).

To ensure consistency, three researchers categorized the designs. The results indicate that while 47% of FC participants used predominantly textual representations, only 28% of GS participants and 21% of E participants did so. For standard graphical representations, the numbers are opposite, with 50% of E participants, 29% of GS participants, and 15% of FC participants predominantly using standard graphical representations. That is, as the level of education becomes higher, there is a progression away from the textual and toward standard graphical notations.

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,b). In order to examine the students' design skills, this work also uses the design representations developed by 149 graduating students, elaborated within the context of Teneberg's study (Tenenberg, 2005). These representations are grouped using a categorization of six levels developed by Eckerdal:

- 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, a drawing of a GUI, 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, such as the GUI or the interface to the database.
- 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.

They found poor performance from students who are near graduation. Over 20% did not produced a software design (nothing and restatment categories) and over 60% communicated no significant progress toward a design. The percentages associated with the proposed categories were: 3% nothing, 18% restatement, 41% skumtomte, 29% first step, 7% partial design, and 2% complete.

The study published by Eckerdal et al. (2006a) was subsequently reviewed by Loftus et al. (2011). They reconsidered the experiment and extended it at Aberystwyth University. This work aims to find out if graduating students can design in groups. The study was conducted in the context of a final-year

undergraduate course when software design and Agile Methodologies concepts are consolidated. Six groups of 10 students each participated, and the assigned task was again the "super alarm clock."

Two of the authors analyzed and score the designs independently, and the criteria were unified in a consensus meeting.

They use a scoring method based on a grade and a weighting that is then mapped into one of Eckerdal et al.'s six categories.

The results indicate that three of the six groups are classified in the Skumtomte category, one in the Restatement category, one in the First step category, and one in the Partial design category.

The results confirm those reported by Eckerdal et al., that many graduating students cannot design software systems. 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 by Lynda Thomas et al. (Thomas et al., 2014) also expanded on the research of Eckerdal et al. (2006a). This study aims to determine the students' understanding when asked to "produce a design." They used the same design task, "super alarm clock," and Eckerdal's categorization. Thirty-five students who were taking a compulsory year-long final year projects course participated.

The results obtained do not turn out to be as bad in the area of students' design skills as reported by Eckerdal et al. (2006a) and Loftus et al. (2011). Specifically, the authors found a somewhat shifted distribution between their classifications and those reported by Eckerdal et al. (2006a). There was a greater proportion of the top three categories (first step, partial design, and complete designs) and a smaller proportion of the first three categories (nothing, restatement, and skumtomte designs). Regarding understanding the instruction "produce a design," some students meant drawing a picture, and some drew quite complex attractive GUIs. Other students understood that they were expected to use some special design notation but could not use it to produce a design. Also, some students understood that they must use design artifacts such as class and interaction diagrams. However, these artifacts were presented independently from each other and were not linked to the original problem.

Furthermore, the authors built a list of fundamental characteristics that differentiate how students understand the design phenomenon. Thomas' cat-

egory is described as follows:

- 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, such as use case diagrams, but does not describe system structure or behavior.
- 2SS - Static structure: focus on design techniques of software engineering, expressing the components and their structural (i.e., static) relationships using notations like class or architecture diagrams.
- 3DB - Dynamic behavior: focus on design techniques of software engineering, expressing the components and some sequential (dynamic) behavior using notations like sequence diagrams or flowcharts.
- 4MRA - Multiple related artifacts: use multiple artifacts and relates components across different artifacts - the static and the dynamic are linked.
- 5EC - Expert category: the notations are relaxed, and only the essential artifacts are included.

Although the study does not present the mapping of the designs submitted by students using the proposed categorization, it illustrates in each category a student's design as an example.

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). Twelve advanced students participated in the study and performed the task individually in the context of a course. In addition to the design representation, the students must submit the skeleton code based on the design.

The design representations were categorized using Eckerdal et al.'s six categories. 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. Although the above results seem good, the authors consider that students could not identify opportunities to create functional, structural design elements. Many created design elements and their behaviors often based on intuition, not analysis.

Years later, Thomas et al. studied software designs of students halfway through their 3rd-year undergraduate computing degree (Thomas et al., 2017). They were looking for characteristics of skills development in software design as students progress through the curriculum. 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 (The Parking problem).

Although 161 students enrolled in the course, 96 software designs were collected and categorized using Eckerdal's categorization. The remaining students either dropped out before the course, did not attend, or took home the assignment; thus, the assignment was disregarded. The 96 designs were categorized by two researchers independently, and discrepancies were resolved by discussion.

The results show that most designs were in the first step category, followed by the skumtomte category and partial. Then, the authors compared the results with those obtained from a group of graduating students from the same university (Thomas et al., 2014). The comparison shows similarities in the classifications from both cohorts. The graduating students produced a few more partial and complete designs but were relatively low cases. From an optimistic point of view, these results suggest that students improve between the halfway and the final point of their studies. However, less optimistically, the improvement was minimal. It appears that many students reach a basic level of design capability and no more.

More recently, Perez-Gonzalez et al. (2019) conducted a study that seeks to know how design skills are acquired throughout a 5-year university program. The study conducted at the University of Mexico seeks to answer the following research question: "How do students' software design skills differ according to the stages of their curriculum?" Seventy-seven students participated in the study between semester 2 and semester 8 of their degree program. Students participated voluntarily but received motivation from their teacher (i.e., extra points in the final exam).

The task is to produce design artifacts according to The Parking problem proposed by Thomas et al. (2017). The designs were also classified using the categorizations proposed by Eckerdal et al. The results show that students who have recently taken design and programming courses produce the best designs. Most of the better-classified designs (Complete and Partial) were produced by students in the middle of their academic program (semesters 4, 5, and 6). Students at the beginning of their career (2nd and 3rd semester) produce the worst designs, but surprisingly these results are similar to those of the 7th semester of their career.

The results of the last two semesters (8 and 9) are not as good as semesters

4, 5, and 6 but not as bad as the first semesters. In this case, the authors believe that it could be because, in the last years of the course, the design tasks are not demanded to the students, emphasizing only the delivery of the code. They conclude that the learning of software design by students can be achieved from courses and practices during the first half of a career, but these skills may decrease if they are not applied later in the academic program.

Finally, Stikkolorum et al. proposed an experiment involving 196 students working in pairs to work out a simple design task (Stikkolorum et al., 2015). They are 3rd-year Software Engineering students. The task is to design the class diagram of a game using the WebUml tool for modeling and registration. Experts evaluate the quality of the designs, considering how well the diagram relates to the problem and how well the diagram is organized. From the analysis of the diagrams, it is observed, that they are not clear about which design elements to include. They tend to add unnecessary elements to the design, and choosing between attributes or classes (that address OO comprehension) seems to be a general difficulty for students. On the other hand, the design strategy (Breadth First or Depth First) chosen by students does not seem to have an impact on the quality of the design achieved (as assessed by experts).

Summary of results

In general, studies have shown that most graduating students are not competent in designing software (Eckerdal et al., 2006a,b; Loftus et al., 2011; Tenenberg, 2005). Students do not describe the behavior of the system (Loftus et al., 2011), do not understand what sort of information a software design should include (Stikkolorum et al., 2015), and how to effectively communicate that information (Eckerdal et al., 2006a,b). In addition, students seem not to know how to iterate in the design process, tending to make the design they started with work unless something really went wrong (Hu, 2016).

Some studies investigate design skills throughout a career (Perez-Gonzalez et al., 2019; Tenenberg, 2005; Thomas et al., 2017). The reported results do not agree on the fact that students' advance in educational level is the factor that influences the construction of better designs. In Thomas' study, as students advance in their degree, they achieve a decrease in the use of textual notations and an increase in the use of standard graphical notations (Thomas et al., 2017). Furthermore, Perez et al. find that the best designs are produced

by students who have new knowledge (took courses recently) (Perez-Gonzalez et al., 2019).

2.3.3 Synthesis of the works obtained related to undergraduate students' understanding of design

Undergraduate students' understanding of design has also been analyzed from different points of view.

In the study reported by Higgins et al., 82 first-year undergraduate students performed a software development task following an informal development process (Higgins et al., 2019). They are taught to program in Java and then they must solve specific problems using this language. Students were also taught to use pseudocode as a design technique in order to design solutions for the exercises. Follow-up surveys with students, a post-test survey, and focus groups were conducted. From the results, it can be observed that students regard engaging in software development to be primarily about programming; and they consider that designing solutions are not useful and avoid them where possible. This result generates concern for the authors since, apparently, the problem is not that students do not have the aptitude to be software developers but that they are not developing the skills of analysis and design. 94% of students see the programming process as more important than the analysis and design stages, suggesting that they do not see the value of carrying out planning before writing a program. This difficulty is reflected by many students indicating that they move immediately to the coding phase before adequately decomposing a problem or carrying out at least some design for a solution.

Flores and Medinilla propose a study that focuses on what kind of ideas the students have regarding object-oriented design (Flores and Medinilla, 2017). The study was carried out in the context of the subject called Software Engineering in the sixth semester of the Degree in Informatics Engineering. Eighteen students of different nationalities participated voluntarily. An initial survey of ten questions was carried out that aimed to know in general terms the ideas and knowledge about object-oriented design. Then, an interview was conducted with open questions. The main questions focused on discussing the object-oriented design, objects, difficulties in the concepts, and understanding the Information Hiding Principle. Results show that the most challenging

concepts to understand are "Uncertainty" and implementing the Information Hiding Principle. Many students believe that object-oriented design is not as important as coding. Some students ignore basic principles of design, such as concepts like object, inheritance, ambiguity, etc.

Another way to assess students' understanding of design is to ask them to reflect on their designs (Coffey, 2017) or to evaluate the designs of others (Prasad and Iyer, 2020). In the study reported by Coffey, thirteen students participated who were asked to design using a UML class diagram (Coffey, 2017). Students have five opportunities to reflect on their design and change it throughout the semester. These reflections are evaluated as part of the course and are factored into the final grade. In each of the five reflective instances, students must describe what changed between the preliminary and final designs and why those changes were necessary. The reflections of the first projects were broad ideas, such as that lack of a good design makes coding more complex, that prototyping might help get a better design, and that assigning responsibilities is complex and important. The reflections for the fifth project were often quite focused, such as designing for readability or creating the most straightforward design solution possible. In general, the results indicate that students achieve insightful reflections on lessons learned from their design activities, which change positively between the first and fifth reflective instances. However, requiring students to perform reflective activities in no way ensures that students recognize the value of performing such activities or internalize reflection on the quality of their work as an ongoing strategy for improvement.

Prasad et al. studied how graduating students evaluate software design diagrams against stated requirements (Prasad and Iyer, 2020). The study was conducted with one hundred senior-year (fourth-year) students of Computer Engineering and Information Technology Engineering. The requirements and design of an ATM system were given to students. The design provided consisted of a class diagram and three sequence diagrams. Students must identify defects in the design diagrams based on the requirements. The authors introduced defects in the diagrams, focusing on including semantic and non-syntactic defects. Specifically, five semantic defects were injected into the diagrams (five scenarios that do not satisfy the requirements). Syntactic defects can be uncovered by a superficial search on the design diagrams, while semantic defects no. Semantic defects quality refers to how faithfully the

modeled system is represented. Detecting semantic defects require students to think deeply about the design, understand the relationship between different diagrams and evaluate the diagrams against the given requirements. Based on the defect explanations by the students, the authors created the following set of 6 categories:

- category 1: identify scenarios that do not satisfy the requirements
- category 2: identify necessary functions which are not used
- category 3: change existing functionalities and requirements
- category 4: add new functionalities in the design
- category 5: change data types, functions, and structure of the class diagram
- category 6: blank responses and no defects

These categories describe the defects' generalities and allow grouping the students' answers. In categories 1 and 2, students evaluate the design diagrams against the requirements by identifying alternate scenarios and simulating function execution. The analysis shows that none of the students could identify all the scenarios (all defects injected). Twenty-three students identified exactly one scenario and only seven students identified exactly two scenarios (category 1). Also, twelve students identified at least one function present in the class diagrams that is not being used in the sequence diagram, and hence not satisfying a particular requirement (category 2). Considering the syntactic elements in the design, thirteen students identified defects based on the syntactic elements in the design diagrams (category 5). Out of the thirteen, five students exclusively focused on syntactic defects and did not identify other types of defects. Focusing on new elements absent in the design, forty-three students mentioned either adding new functionalities to the design or changing the existing functionalities and requirements (categories 3 and 4). Of these forty-three, twenty-four focused only on changing functionalities and did not detect any other type of defect. These students are not evaluating the design against the requirements but are attracted to introducing new functionalities to the proposed reality. The authors conclude that many students could not evaluate the design against the requirements, which provides indicators of the difficulties students face when designing.

Summary of results

Students seem unaware of the importance of detailed software design. For them, the software development process mainly focuses on coding, preventing design activity (Flores and Medinilla, 2017; Higgins et al., 2019). In addition, Flores and Medinilla (2017) found that many students ignore certain basic design principles.

Students' understanding of the design also reveals difficulties. When they reflect on their design, they get interesting insights, although the authors mention that reflection does not ensure that they can achieve a better quality of their work (Coffey, 2017). On the other hand, when students evaluate designs produced by others, the results reveal that some students do not succeed in evaluating designs, while others focus on evaluating only particular aspects (syntactic or semantic) of a design (Prasad and Iyer, 2020).

Chapter 3

Theoretical framework

This chapter presents the relevant concepts framed in the thesis. Specifically, section 3.1 presents an introduction to the Personal Software Process used as a development framework throughout the family of experiments. Section 3.2 presents the characteristics of the course proposed for the family's execution.

3.1 Personal Software Process

The Personal Software Process (PSP) was proposed in 1995 by Watts Humphrey at the Software Engineering Institute (SEI) (Humphrey, 1995). It aims to increase the quality of the products manufactured by individual professionals by improving their personal methods of software development. It considers diverse aspects of the software process, including planning, quality control, cost estimation, and productivity.

Since the PSP process has several methods that engineers do not generally practice, the PSP methods are introduced in a series of seven process versions. These versions are labeled PSP0 through PSP3, as shown in Figure 3.1.

PSP level 0, called "Personal Measurement," aims at engineers to gather data on the time they spend per phase and the defects they find.

The PSP0 is divided into phases, as shown in Figure 3.2. A project begins with the requirements for a software module and ends when the software is released. The phases are planning, design, code, compile, unit test, and postmortem. In the context of this thesis, the process defined in PSP0 and the design templates proposed in level 2.1 are used as a framework.

In the PSP, all tasks and activities to be performed during software devel-

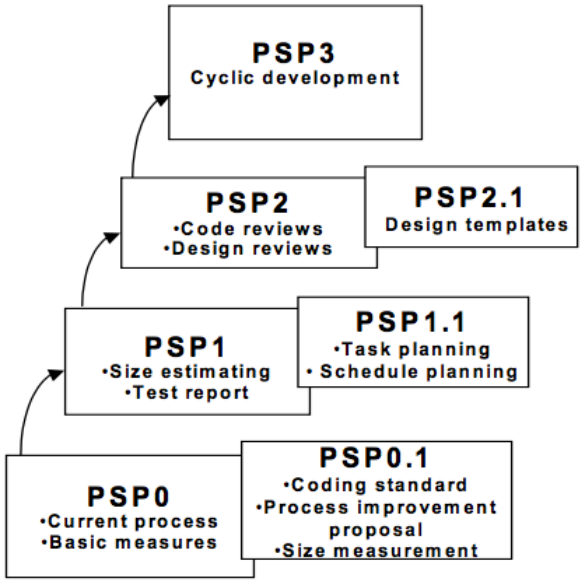


Figure 3.1: PSP structure levels

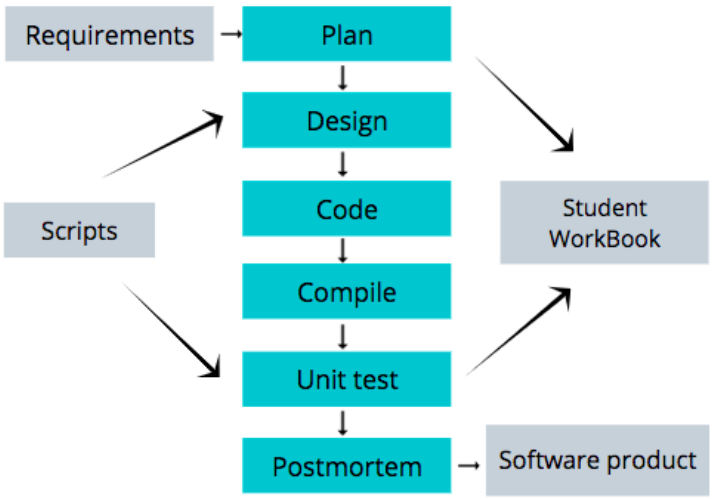


Figure 3.2: PSP0 process

opment are defined in a set of documents called "scripts." Scripts dictate the course of the work and are to be followed in a disciplined manner. They also facilitate data collection of the software process, including time spent at each phase, defects detected at each phase, time spent in detection and correction, the phase at which each defect is detected and removed, and the classification of defects into types. This data is collected into logs and used to evaluate the

process's quality using indicators like defect density, review rate, and yield. All these measurements render a highly instrumented process, ideal for the realization of empirical studies (Wohlin et al., 2012). Every script comprises a purpose, a set of entry criteria, the activities to perform, and the expected outcomes (i.e., exit criteria). The Process Script contains a general program for planning, development, and postmortem activities. The development script, in turn, consists of the phases of design, code, compile, and unit testing. Figure 3.3 presents the development script as an example.

Purpose	To guide the development of small programs	
Entry criteria	- Requirement statement - Time and defect recording logs	
Step	**Activities**	**Description**
1	Design	- Review the requirements and produce a design to meet them - Record in the defect recording log any requirements defects found - Record time in the time recording log
2	Code	- Implement the design - Record in the defect recording log any requirements or design defects found - Record time in the time recording log
3	Compile	- Compile the program until there are no compile errors - Fix all defects found - Record defects in the defect recording log - Record time in the time recording log
4	Unit test	- Test until all test run without error - Fix all defects found - Record defects in the defect recording log - Record time in the time recording log
Exit criteria	- A thoroughly tested program - Completed time and defect recording logs	

Figure 3.3: Development script

The design phase consists of designing the program completely and unambiguously. During PSP0, the design script states that the engineer must review the requirements and produce a design that satisfies them. At this level, the engineer produces the design without specific guidelines on how to design, choosing the design method they deems appropriate. During PSP2, PSP uses four templates to document the design in four dimensions: static, dynamic, internal, and external. The PSP considers a design to be complete when it defines all four of these dimensions. Figure 3.4 illustrates the four templates.

In particular, the operational specification template describes the interaction between user and system (i.e., the dynamic-external view). The functional specification template allows the definition of the structural features to be pro-

Object Specification Templates	Internal	External
Static	Logic Specification Template	Function Specification Template (Inheritance Class Structure)
Dynamic	State Machine Template	Functional Specification Template (User Interaction) Operational Scenario Template

Figure 3.4: PSP Template Structure

vided by the software product, classes and inheritance among them, externally visible attributes, and relations to other classes or parts (i.e., the dynamic-external and static-external views). The state specification template describes the set of states of the program, the transitions between states, and the actions to be taken at each transition (i.e., the dynamic-internal view). Finally, the logic template specifies the program's internal logic (i.e., the static-internal view) concisely and conveniently. Pseudo-code is appropriate for this task.

Once the design is finished, the program is constructed using a programming language and a coding standard during the code phase.

After code comes the compile phase, which is the translation of the source program into machine language using a compiler. This phase involves correcting defects detected by the compiler.

The unit test phase consists of executing the test cases specified during the design phase. The defects detected in the unit test allow the quality of the product to be assessed. In PSP, a program is considered to be of adequate quality if it contains five or fewer defects per KLOC at unit test.

Finally, the postmortem script describes the activities of the postmortem phase, which includes an assessment of both process and product, and an analysis of the injected defects, noting the phases at which they were removed. Analyzing the process and understanding where and why mistakes are committed allows developers to improve their own processes and outputs.

During the family of experiments described in this thesis, students collected their data as they performed their work and recorded it in a tool. In the Baseline and Template experiments, the Student WorkBook tool was used; and in the Habitual experiment, the PSPCode tool was used. Student WorkBook is a tool developed in Access, which provides a set of forms that allow data management throughout the process. PSPCode arose due to certain limitations of Student WorkBook. PSPCode is a web tool with the same functional characteristics as Student WorkBook but with additions that allow better course

management. This tool was created in the context of a degree project directed by the author of this thesis.

Some of the metrics that the student records throughout the process are:

Time: A record of the time (in minutes) spent per phase is kept, allowing the evaluation of the student's performance. For this purpose, the current phase, the start and end date and time of each task performed, the interruption time, and any pertinent comments are recorded for each project.

Number of defects: A record is kept of which defects are injected, in which phase they are injected, and in which phase they are removed. It helps to evaluate at which stage effort should be increased to improve productivity and product quality. For each defect, the type of defect, the phases in which it was injected and removed, the time it took to correct it, and a brief description of the defect are recorded.

Software size: Lines of code were chosen because they meet the following criteria: they can be automatically counted, precisely defined, and are well correlated with development efforts based on PSP research (Humphrey, 1995). Size is also used to normalize other data, such as productivity (LOC per hour) and defect density (defects per KLOC). In PSP, the main measure of size used is logical lines of code (logical LOCs). This measure can be defined as the count of basic execution structures. There are, therefore, different ways of measuring LOCs depending on factors such as the programming language. For this reason, each student defines their own standard, and once defined, they must adhere to it without altering it or changing it for another one throughout the different projects. The student develops the project following the standard and counts the LOCS produced at the end. PSP classifies the LOCS into different categories. These categories are:

- Base: When an existing product is enhanced, base LOC is the size of the original product version before any modifications are made.
- Added: The added code is that code written for a new program or added to an existing base program.
- Modified: The modified LOC is the part of the Base code that is changed.
- Deleted: The deleted LOC is the part of the Base code that is subsequently removed.
- New and Modified: When engineers develop software, it takes them much more time to add or modify a LOC than to delete or reuse one. Thus,

in the PSP, engineers use only the added or modified code to make size and resource estimates. This code is called the New and Modified LOC and is the sum of the Added and Modified LOC.

- Reused. In the PSP, the reused LOC is the code that is taken from a reuse library and used, without modification, in a new program or program version.

In the context of our family of experiments, we consider the total product size as the sum of LOCs Added and Modified.

3.2 Principles and fundamentals of the Personal Software Process

As mentioned above, PSP is based on some practices to improve software development and product quality (Humphrey, 2005a). However, these practices are not commonly used among practitioners. This is why Humphrey, to teach PSP to engineers, uses a form of adoption whereby familiarization with the process occurs gradually through the levels (PSP0 to PSP2.1). The PSP is taught through courses. The courses Humphrey proposed are "PSP for engineers 1" and "PSP for engineers 2." During these courses, as programming exercises (projects) are carried out, PSP practices are introduced.

The course we propose for the execution of the family of experiments is called Principles and Fundamentals of the Personal Software Process (PF-PSP). PF-PSP is a reduced and simplified version of the courses proposed by Humphrey. We use a simplification of the process that serves as a framework to follow a disciplined and measurable process. PF-PSP is an elective course in the last year of the School of Engineering of Universidad de la República in Uruguay. During the PF-PSP course, only levels 0 and 0.1 and the set of templates of level 2.1 are taught (the latter only in Template experiments). These levels also require minimal adaptation to the data collection process.

The PF-PSP course lasts nine weeks and uses the same eight development projects used in the courses "PSP for engineers 1" and "PSP for engineers 2." In the first week (Week 1), the PSP0 (Base process) is taught, and the dynamics of the practical work to be followed throughout the remaining eight weeks are explained. The practical work consists of each student developing eight small projects following the base process and recording the data in the PSP 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 weeks 2 to week 9, one project is assigned per week. The assignment consists of a professor sending the requirements of each project. The delivery of each student 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 delivers the solution, the professor reviews the delivery and sends corrections to the student if necessary.

The schedule of the course is shown in Figure 3.5. This schedule undergoes a modification in Template and Habitual experiments. In the Template experiment, the design templates are introduced in the middle of the course. In the Habitual experiment, the delivery of the habitual design representation is required in the middle of the course. The adjustment to the specific schedule in these cases is explained in detail in the chapters 6 and 7, corresponding to Template and Habitual experiments.

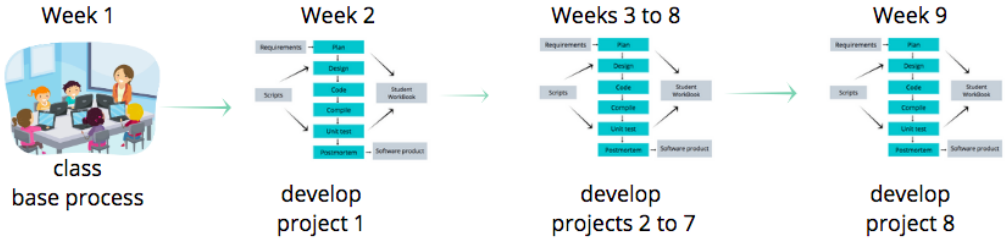


Figure 3.5: PF-PSP course schedule

Students carry out the projects at home and have a professor assigned who will be responsible for assigning the projects, correcting them, and answering questions. Projects are small in size and of low and similar complexity, so the 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). Table 3.1 provides a brief description of the projects.

Table 3.1: Brief description of the programs

Proy.	**Description**
1	Calculate the average and standard deviation of a set of numbers stored in a linked list
2	Size-measuring software
3	Calculate the linear regression parameters b0 and b1, as well as the correlation coefficients r and r2 given a set of pairs of values
4	Calculate the relatives of a class for the ranges: very small, small, medium, large, and very large
5	Numerically integrating a function using Simpson's rule
6	Calculate the t-student function: find the value of x for which integrating the function t from 0 to x gives the result p
7	Calculate the correlation between two sets of numbers X and Y and the significance of the correlation. Calculate the linear regression parameters b0 and b1 for a set of n data pairs, give an estimate XK, calculate the projection YK, where YK = b0 + b1*XK. Finally, calculate the 70% prediction interval for that estimate
8	Calculate the estimation parameters (b0, b1, b2, b3) for multiple regression with three variables

Chapter 4

The family of experiments

In chapter 2, we present related works to the design practice and the experimental works on how students design software in detail. Our goal is to know how graduating students practice software detailed design. To achieve this, we proposed to study the effort they dedicate to software design, how they usually represent their software designs, and the effect of the design on the quality of the software produced. In this sense, we carried out a family of experiments.

4.1 Goals

The aim of the family of experiments is to know how graduating students design detailed software in the context of a course at the School of Engineering of Universidad de la República in Uruguay

We designed and conducted a family of experiments. The family was composed of seven experiments conducted from 2012 to 2018.

The **Baseline** experiments, carried out in 2012, 2013, and 2014, were aimed at learning about the effort dedicated to software design by graduating students, how is the effort variation throughout the different projects, how is the effort variation between students, and what is the students' perception of the problems they face. They also seek to confirm that the experimental framework used is suitable for future experiments.

The **Template** experiments, carried out in the years 2015, 2016, and 2017, were aimed at knowing the effect and the effort of design on software quality when students represent the design using a specific set of design templates.

Finally, in 2018, we conducted an experiment called **Habitual** to find out

how students usually design and the effect on software quality when they deliver their usual design representation to the teachers. This experiment does not use design templates to avoid biasing the students'; usual practice. As we mentioned in chapter 1, the Habitual experiment had a replication in 2021. However, this replication is not described in the thesis since its results are still under analysis.

Fig. 4.1 summarizes the family of experiments. Each rectangle represents an experiment horizontally aligned to the experimental objective. The figure also shows the execution order of the experiment (execution year) and the number of students involved.

The **Baseline** experiment was initially carried out in 2012 and replicated with the same objectives and experimental design in 2013 and 2014. The experiment **Template** was initially carried out in 2015 and replicated with the same objectives and experimental design in 2016 and 2017. Finally, the **Habitual** experiment was carried out in 2018.

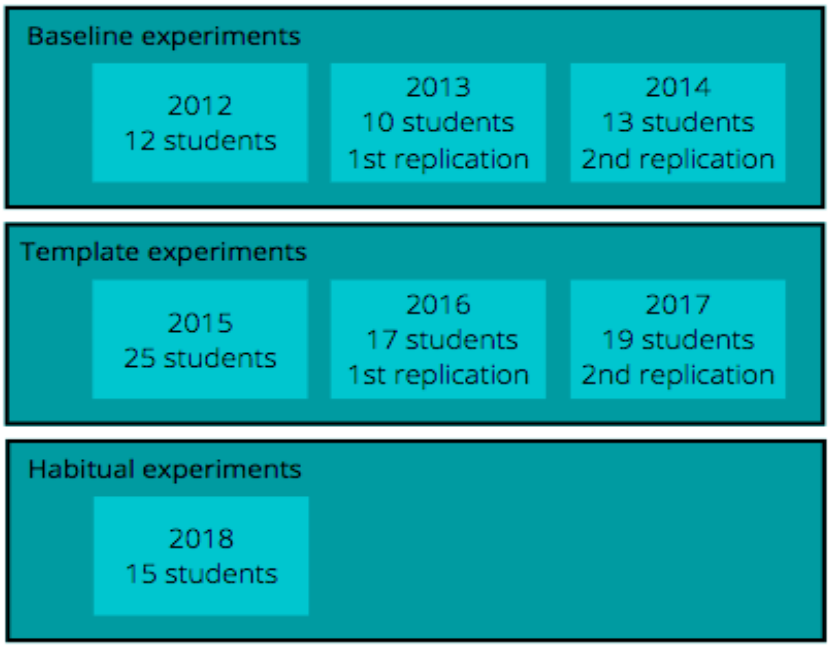


Figure 4.1: Family of experiments

4.2 Experimental context

Each experiment in the family of experiments corresponds to a PSP Principles and Foundations course (PF-PSP). Students who participated in one course do not participate again in a later course. The teachers participating were the same throughout the seven experiments.

The experiments involved graduating students of the School of Engineering of Universidad de la República in Uruguay. Their participation in the experiment is through voluntary enrollment in the PF-PSP course.

All the students have passed the courses Programming 3, Programming 4, Programming Workshop, and Software Engineering. These courses teach software design artifacts and techniques, programming languages, algorithms, and fundamental software engineering concepts. We consider the group of students who participated to be homogeneous since they are students at a similar stage in the career.

The students gave their written consent for the data they recorded during the process to be used in research work without ever revealing their identities.

4.3 Experimental design

In order to study how students design, we collected accurate data on how students develop software. In particular, we are interested in knowing the effort dedicated to each phase, the defects removed in the unit testing phase, and how students represent their designs. For the data from the students to be comparable, a pre-established process is needed to collect measurements during its execution. We decided to use the PSP framework (Base process) to achieve the collection of these data as already presented in chapter 3.

To have quality data, it is essential that students collect their measurements properly and can follow the process correctly. The proper use of the PSP is achieved through practice, developing some projects (Humphrey, 2005b). Although the PF-PSP uses only PSP0 and PSP0.1 (and therefore, it is easier to learn), we believe it requires some practice to be adopted. For this reason, our experimental design implies that students develop eight projects. During the first or second project (depending on the subject), they already follow the process adequately.

People have high variability when applying software development tech-

niques or processes (Humphrey, 2005b). In the presence of high variability on subjects, a within-subjects design is preferable to a between-subjects experimental design (Senn, 2002). 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 various projects (i.e., a repeated-measures design).

Repeated-measures designs have a particular problem: the period by treatment interaction. Period by treatment interaction refers to the particular conditions present in the different periods that could affect the treatments, and so the dependent variable (Jones and Kenward, 2014; Senn, 2002). Our design is a repeated-measures design with eight periods (one period for each project the students develop).

One source of period-by-treatment interaction in our experiment could be the learning effect. Students improve the quality of their production due to the knowledge they acquire every time they finish one project and move to the following one. Both our experiment design and our analysis contemplate this.

Another source of period-by-treatment interaction is the different programs the students develop. That is, each program's difficulty or other particularities could affect the dependent variables we want to study.

In our design, we cannot distinguish between these two periods by treatment interaction (learning effect and program). Both effects are confounding with the period.

Repeated-measures designs also have benefits. An important one is that all other things being equal, repeated-measures designs have more power to detect effects than independent designs (between-subjects designs) (Field, 2018). So, in a repeated-measures design, fewer subjects are needed to have the same power. As our PF-PSP course has a few students each year, this is (again) a good design considering the course's characteristics.

As mentioned in chapter 3, the nature of project 2 differs from the other projects. In project 2, students must 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 the quality of program 2 is usually lower than that of the other projects (Vallespir et al., 2014). Therefore, we excluded the data of this project from the analyzes presented in this thesis. However, it is relevant to mention that project 2 is an integral part of our course, and it is used by students from

projects 3 to 8 to count the lines of code they produce in each project.

Some response variables are common to the different families of experiments. External quality is common to all three family experiments. We measure the external quality as the number of defects found in the unit test phase (UT) per every thousand lines of code (defect density). The consequence of high defect density in UT in software engineering is typically seen as defect-fixing or rework effort incurred in projects, which results in poor quality products. The effort dedicated by the students to the design and code phases is another response variable, in this case, common to the Baseline and Template experiments. The effort is measured as the time in minutes the student dedicates to the phase in question.

The remaining response variables are described in each experiment because they depend on each experiment's specific objectives.

Chapter 5

Baseline experiments

The Baseline experiment process and the subsequent experiments of the family were carried out following the experimental design and conduction steps proposed by Wohlin et al. (2012).

5.1 Definition

The aims of **Baseline** experiments are to know the effort dedicated to software design by graduating students, how is the effort variation throughout the different projects and how is the effort variation between students. Such effort is measured as the time spent thinking of a solution for the problem and constructing that solution before coding.

In addition, we also want to investigate software quality during the projects. Proving that software quality does not change for the mere fact of carrying out seven consecutive projects lets us establish and validate the experimental framework for the subsequent experiments of this family. Finally, we are interested in getting to know the students' perception of the problems they face (related to design).

Our interest is that students solve each exercise in their own way, applying their skills and knowledge. The course does not require students to hand in the completed design. We believe that a mandatory delivery of the design would cause students to exert more effort than they would to produce it; thus biasing our research.

The PSP proposes, as one of the quality measures of the process, indicating the quality of the design as an indirect measure, the ratio between the detailed

design time and the coding time. Empirically, it has been shown that when this ratio is close to 1 (equal design and coding time), products of better quality are produced than those produced when this ratio is far from 1 and developers use more time in coding than in designing (Humphrey, 2000, 2005b). We are interested in knowing this relationship in the projects carried out by graduating students of our career.

5.2 Planning: research questions and experimental design

In the **Baseline** experiment, we propose the following research questions and the corresponding research hypotheses:

RQ1: What is the ratio between the effort spent designing and the effort spent coding?

H1.0: The time spent on designing equals the time spent on coding.

H1.1: The time spent on designing does not equal the time spent on coding.

Empirically, it has been shown that when the ratio between the detailed design time and the coding time is close to 1 (equal design and coding time), products of better quality are produced than those produced when this ratio is far from 1 (Humphrey, 2005b).

To learn more about the variability of the time used in Detailed Level Design (TDLD) and time used in Coding (TCOD) ratio, between students and from each student across projects, we added the following two questions:

RQ2: What is the variability of the ratio (TCOD/TDLD) among students?

RQ3: What is the variability of the ratio (TCOD/TDLD) across the projects performed by each student?

To know if students produce better quality products as they develop the projects:

RQ4: Is there any improvement in product quality as students develop the projects?

H2.0:Defects density in UT is the same as the student develops projects.

H2.1: Defects density in UT is not the same as the student develops projects.

Finally, we want to know how students feel about the design activity:
RQ5: What is the student's perception of software design?

Baseline experimental design is a repeated measures design, where students apply the base process along eight projects (see chapter 3).

5.3 Operation

A total of thirty-six students participated in the course. However, for the analysis in **Baseline** experiments only 35 were considered: 12 in 2012, 10 in 2013 and 13 in 2104. All students developed the same eight projects in the same order. Figure 5.1 presents the Baseline design.

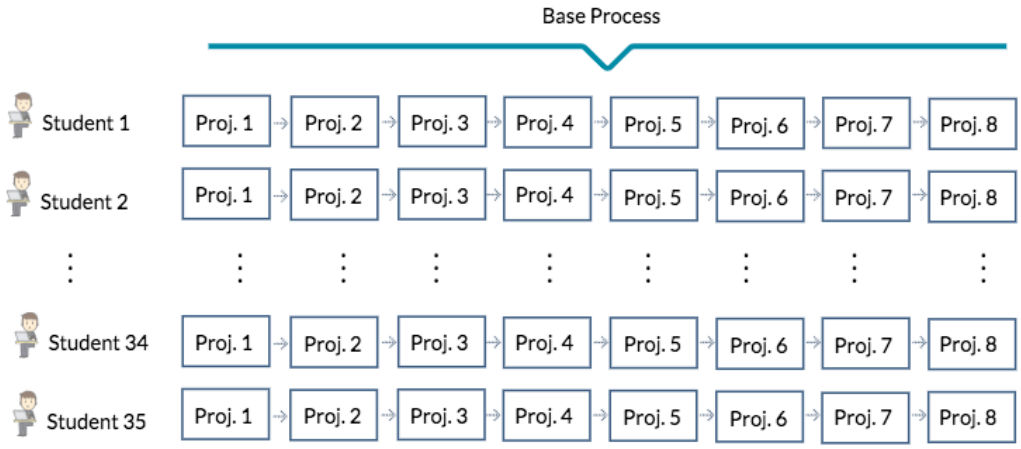


Figure 5.1: Baseline experiment design

In each period, there is a different project to develop, which can affect the dependent variables as explained in chapter 4. As already mentioned, the effect of the different projects is confused with the learning effect.

Applying the base process in eight projects reduces the effect of the students' intrinsic variation. These variations can occur due to motivation, or a particular state of mind during a specific period, among other factors. In addition, as already mentioned in chapter 4, repeated measures allow the student to incorporate the process. This normally occurs in the first or second project

developed.

The response variables are time spent designing the solution, time spent coding, defect density (defined in chapter 4), and student perception of the design (measured through the PIP (personal improvement process) and final course questionnaire).

5.4 Analysis and interpretation

To answer RQ1: "What is the relation between the effort dedicated to designing and the effort dedicated to coding?", we analyze the following hypothesis test:

H1.0: Median (TCOD) <= Median (TDLD)

H1.1: Median (TCOD) > Median (TDLD)

The data to be used consists of seven pairs of values (TDLD, TCOD) per student, since we considered seven projects per student. Initially, a total of 36 students participated, so we have 252 pairs of values.

We analyzed the existence of outliers using a box and whisker diagram. Twenty-one outliers belonging to eight different students are detected and each outlier's data is analyzed manually one by one. From this analysis, we detected that one of the students has data that reflect that he did not follow the established process properly. Therefore, we removed the seven pairs of data from that student from the analysis (these data have three outliers). The analysis of the remaining outliers shows no errors, so we have a total of 245 pairs of data and 35 students in our analysis.

We analyzed the total time in design and the total time in code for each student, considering the seven projects. Using the total time reduces the possible variation caused by the motivational effect of the student and the intrinsic difficulty of a particular project. Therefore, we have a pair of data (TDLD, TCOD) for each of the 35 students. The calculation for each pair of data is:

$$(\sum_{i=1}^{7} TDLD_i, \sum_{i=1}^{7} TCOD_i) \tag{5.1}$$

where $TDLD_i$ is the time spent in the design phase for project i, $TCOD_i$ is the time spent in the code phase for project i, and where i varies from 1 to 7.

Table 5.1 presents the 35 data pairs (TDLD, TCOD).

Table 5.1: Data pairs (TDLD, TCOD) in minutes

Sample	TDLD	TCOD	Sample	TDLD	TCOD
1	388.40	635.80	19	169.10	536.00
2	10.28	305.10	20	224.28	829.60
3	87.03	744.10	21	198.00	486.00
4	248.28	745.90	22	240.66	288.40
5	209.50	566.60	23	182.46	547.10
6	26.36	440.40	24	98.93	471.50
7	203.58	554.90	25	52.23	570.80
8	108.62	650.60	26	162.53	569.20
9	663.90	942.70	27	120.00	754.38
10	210.50	838.30	28	128.10	1242.40
11	20.70	471.28	29	56.00	338.00
12	112.30	747.40	30	82.05	548.00
13	67.40	525.60	31	116.50	363.20
14	168.61	631.50	32	71.18	727.60
15	51.20	428.00	33	55.68	418.80
16	65.10	451.50	34	166.91	531.30
17	371.60	502.20	35	39.65	375.70
18	64.00	963.00			

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, a significance value of 1.443e-15 is obtained, indicating that the values do not fit a normal distribution.

Table 5.2 presents the median and the interquartile range of the 35 data for TDLD and TCOD. The medians of TDLD and TCOD show a significant difference, while in the dispersion of the data the difference is less. The values obtained with the descriptive statistics give us an idea of the variability of the students.

Table 5.2: median and the interquartile range (minutes) of the 35 data for TDLD and TCOD

	Median	Interquartile range
TDLD	21.39	19.46
TCOD	84.66	39.21

Therefore, and because the sample size is small, we used the Wilcoxon's

one-tailed for paired samples test (Gibbons and Chakraborti, 2011). The samples are paired since the sampled pairs (TDLD, TCOD) correspond to the same student. For each student, the total time in design (Mdn = 21.39) did differ significantly from the total time in code (Mdn = 84.66) considering projects 1 to 8, p-value = $5.821e^{-11}$. That is, students spend more time coding than designing.

So, to find out how much more or what the relationship is between these times (TCOD = N*TDLD), we proposed to apply the test again but now multiplying TDLD by an N value varying between 2, 3 and 4 until the null hypothesis is not rejected. The execution of the statistical tests throughout the thesis is carried out by the author of the thesis using the R tool. One of the thesis tutors validated them using the SPSS tool for the Baseline experiment and Jamovi tool for Templates and Habitual experiments.

*H1.0: Median (TCOD) <= Median (N*DLD)*

*H1.1: Median (TCOD) > Median (N*TDLD)*

The results indicate that our students generally spent at least three times more coding than designing. For N=2 and N=3 the total time in design did differ significantly from the total time in code (p-value = $8.57e^{-07}$ and p-value = 0.0056 correspondingly). For N=4, the null hypothesis cannot be rejected (p-value = 0.35, so the coding time is greater than three times the design time.

RQ2: What is the variability of the ratio (TCOD/TDLD) among students?

To determine the statistical test that best fits the problem to be solved, we studied the data distribution, checking that some students' values do not fit a normal distribution. Then, to know the variation among students, we applied the Kruskal-Wallis non-parametric test. We analyzed each student's ratio (TCOD/TDLD) considering the seven projects. Therefore, we have seven data (TDLD/TCOD) for each of the 35 students.

The hypotheses for the test are:

H2.0: The median of the ratio (TDLD/TCOD) for all of the 35 students is the same

H2.1: There is a at least one median from one student that it is different

The Kruskal-Wallis H test showed that there was a statistically significant difference in (TCOD/TDLD) ratio among students (Chi square = 151.68, p < 0.001, df = 34).

To analyze the variation among students graphically , we used a boxplot and histogram diagram presented in Figure 5.2 and in Figure 5.3 respectively. For both diagrams we plotted 35 values, corresponding to each of the 35 students. For each student, the average TCOD/TDLD relation of the seven projects is calculated as:

$$(\frac{\sum_{i=1}^{7} TCODi}{\sum_{i=1}^{7} TDLDi}) \tag{5.2}$$

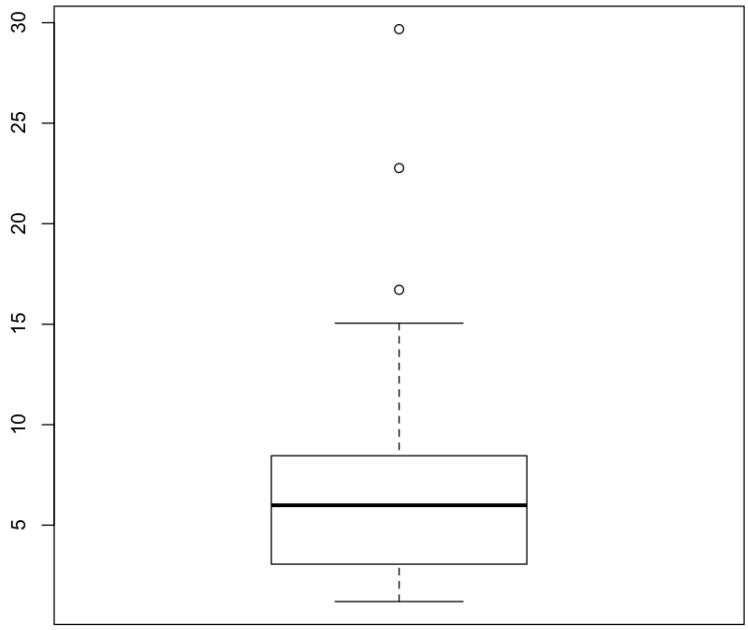


Figure 5.2: boxplot of the average TCOD/TDLD of each student

The boxplot diagram clearly shows the vast variation between students in the relation of effort spent on code and design. For all students, this varies from 1.20 to 29.68. The interquartile range (IQR) is 5.39, quartile 1 is 3.06, quartile 3 is 8.45 and the median is 6.97. This indicates a significant dispersion in the data.

The histogram diagram allows us to observe the variability among the 35 students in a different way. It can be observed that 23 students (65%) have a TCOD/TDLD ratio between 1 and 6. Also, the histogram shows that 31 students (88%) have a TCOD/TDLD ratio between 1 and 11.

Both diagrams allow us to observe that students behave differently and are highly variable between them concerning the relationship of effort between coding and design.

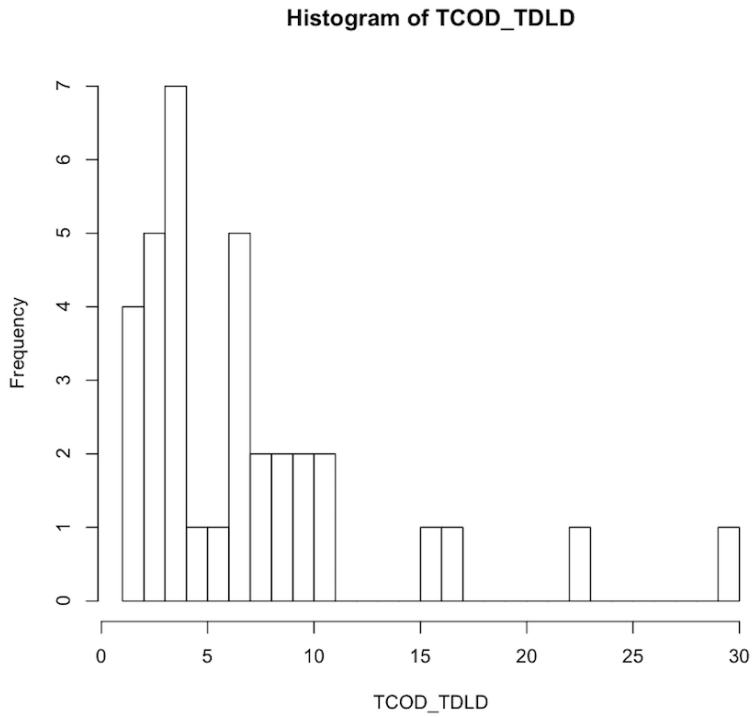


Figure 5.3: frequency of the average TCOD/TDLD of each student

RQ3: What is the variability of the ratio (TCOD/TDLD) across the projects performed by each student?

In order to know the individual variation we analyzed each student's TCOD/TDLD ratio in the seven projects. This variation can be analyzed from the value of the first quartile, third quartile and the interquartile range of each student presented in Table 5.3.

It can be seen that the individual variation is very uneven. Some students such as 1, 9, 17, 20 and 22 have little variation throughout their projects. In these cases we could say that students are "consistent" in relation to the time they spend on design and coding in the different projects. In other cases, for example students 2, 6, and 11 present a significant (for example, with an IQR of 20.63) variation in the TCOD/TDLD ratio during the performance of their projects.

Figure 5.4 shows the variation in TCOD/TDLD ratio throughout the seven projects for students 1, 2, 6, 9, 11, 17, 20, and 22. This also clearly shows that some students are more consistent (predictable) and others are not regarding the TCOD/TDLD ratio.

Table 5.3: Q1, Q3 values and IQR per student in the seven projects

St.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Q1	1.16	23.80	5.67	2.58	1.88	15.17	2.26	4.07	1.46
Q3	2.64	44.42	11.71	4.54	4.66	24.97	4.41	8.55	1.84
IQR	1.49	20.62	6.04	1.96	2.77	9.79	2.15	4.49	0.38
St.	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
Q1	3.07	14.37	5.23	5.71	2.78	6.46	6.93	1.00	13.17
Q3	5.09	32.13	12.07	10.04	5.53	9.17	8.45	1.56	17.05
IQR	2.03	17.76	6.84	4.33	2.74	2.72	1.52	0.56	3.88
St.	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**
Q1	2.02	3.38	1.71	1.25	2.46	4.72	6.97	2.55	3.51
Q3	4.28	4.00	3.58	1.91	3.98	7.07	12.96	4.67	11.17
IQR	2.26	0.61	1.87	0.66	1.52	2.35	5.99	2.12	7.66
St.	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	
Q1	5.98	4.20	4.30	3.03	5.75	5.45	2.00	7.70	
Q3	14.81	7.64	13.03	4.85	12.24	10.11	5.67	14.42	
IQR	8.83	3.45	8.73	1.82	6.49	4.66	3.67	6.72	

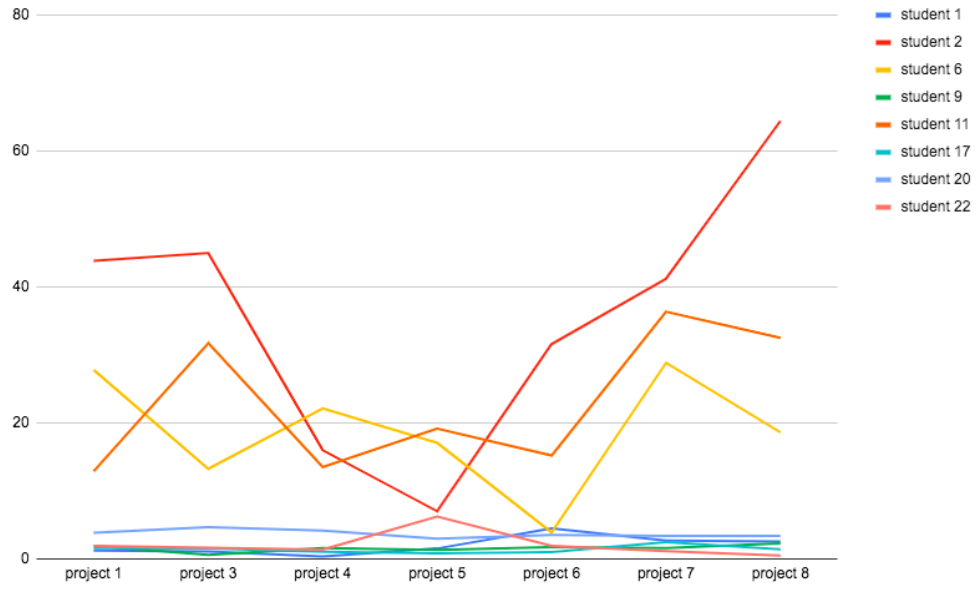


Figure 5.4: TCOD/TDLD ratio variation for eight students throughout the seven projects

RQ4: Is there any improvement in product quality as students develop the projects?

To answer RQ4 we used defect density in the UT phase as the response variable.

The research hypotheses are as follows:

H4.0: Median(Defect density in UT i) = Median (Defect density in UT j)

H4.1: Median(Defect density in UT i) <> Median (Defect density in UT j)

i are the students during projects 1, 3, and 4, and j are the same students during projects 5 to 8.

We studied if there is a difference in software quality between group 1 (projects 1, 3, and 4) and group 2 (projects 5 to 8).

Each sample corresponds to the average defect density in UT of a student.

During the development of the seven projects, students could improve due to a learning effect based on repetition of programming or due to an analysis of their own process data (that, at some point it is also a learning effect). For example, they could find that their designs are not as good as they could be, either looking at their designs or analyzing the time spent in design compared to coding.

To know if such improvement exists, we studied the defect density of group 1 (projects 1, 3, and 4) and group 2 (projects 5 to 8). Verifying that the quality of the software does not vary between group 1 and group 2 allows eliminating the learning effect.

We have 35 samples corresponding to the 35 participating students. Each sample consists of the data pair (Def. density UT i, Def. density UT j) where i corresponds to group 1 projects and j to group 2 projects. For each student, the data pair (Def. density UT i, Def. density UT j), is calculated as follows:

$$\left(\frac{1000 * \sum_{n=1}^{4} \#defectsUT_n}{\sum_{n=1}^{4} \#LOC_n}, \frac{1000 * \sum_{n=5}^{8} \#defectsUT_n}{\sum_{n=5}^{8} \#LOC_n}\right) \tag{5.3}$$

Where n varies between 1, 3, and 4 for group 1 and between 5, 6, 7, and 8 for group 2. The samples are paired because they correspond to the defect densities in UT of the same student. Therefore, the statistical test to be applied is the one-tailed Wilcoxon test for paired samples.

Table 5.4 presents the median and the interquartile range for group 1 and group 2.

Defects density in UT for students in the group 1 (Mdn = 28.0) did not differ significantly from students in the group 2 (Mdn = 22.6), V = 429 and p-value = 0.06249. That is, we cannot state that the quality of the software

Table 5.4: median and the interquartile range (#def/kLOC)

	Median	Interquartile range
group 1	28.0	21.3
group 2	22.6	16.1

developed between group 1 projects and group 2 projects is different. Therefore, there is no learning effect or improvement in the quality of the products as students develop the projects.

RQ5: What is the student's perception of software design?

To answer RQ5 we analyzed all PIP records concerning the students' views of the design and the final questionnaire of the course. Students complete PIP records on a mandatory basis for projects 2 through 8. Only ten PIP records (out of 245 PIPs) from different students describe design-related problems. The records presented in Table 5.5 show how in a specific project, these students realized that they did not spend enough time in the early stages of development and the effects of this.

From the description of the PIPs, we can extract common problems identified by the students. For that, we used inductive coding (assign labels representing important and recurring themes in each PIP) and analyzed the content with a qualitative analysis. The coding was made by the author of this thesis and Diego Vallespir independently. During a consensus meeting, criteria were unified, resulting in the final tags.

We observed that most of the students describe the problems in a problem-consequence way; for example: spending little time on design causes an increase of time in the testing phase. So, we tagged three problems: too little time spent on design, lack of understanding of the problem during design, and not thinking about the solution to the design during the design phase; and six consequences: too many defects detected in testing, costly defects detected in testing, avoidable defects in testing, resolution of the design during the coding phase, too much time spent in coding, and testing and rework in coding and testing.

We also analyzed part of the questionnaire conducted in the last course in the sub-family (14 participants). The questionnaire asks about the student's academic and professional information and the application of the development process. We analyzed the questions presented in Figure 5.5 that are the ones of interest for this thesis.

Table 5.5: PIP Description

PIP Description
"The problem I encountered was that I spent too little time on the planning and design stage, so the coding and testing stage took more time as many of the problems could have been prevented at those stages."
"The problem was not understood at the design stage, wich led to further delay at the coding stage when looking for the best solution at the time."
"There was a poor follow-up on the description of the Gaussian method, which led to problems in the implementation."
"I had problems with interpretation and design; I wasted a lot of time to be able to solve a defect that I found only at the UT stage."
"In the planning and design stage I read the assignment and think I understand what I have to do, but in testing I find problems that show I didn't understand."
"I continue to have attention errors that cost me a lot of rework time. I feel that I spend too little time on early stages of the project such as planning and design which is then reflected in costly mistakes."
"I'm making several mistakes because of not thinking well before coding. The solution for this is to make a detailed design, in order to define each method to be developed with its variables. This is going to generate more design time, but it decreases coding time and the number of errors injected."
"Large number of bugs (in this case related to algorithm counts) that are discovered only at the testing stage."
"I had logic problems in the functions. I should stop more at the planning and design stages to understand the algorithms in more detail."
"I think I didn't take enough time to thoroughly understand the project at the design stage. This caused several inconveniences at the time of coding and testing, as I only realized at those stages that I had not understood the project, and wasted time and rework, which I could have avoided."

Regarding question 1, all 14 student who participated in the last course in the sub-family answered that they performed pseudo-code during the design phase. In addition, only 1 of the 14 also reported relying on class diagrams and collaboration diagrams. For question 2, 13 affirmative responses were obtained, while one student considered that not enough.

1. Select which of the following activities you performed during the design phase □ Pseudo code □ Collaborative diagram. □ Class diagram □ Other. Briefly describe
2. Do you consider that you were taught how to design at university?

Figure 5.5: Questions in the questionnaire

5.5 Discussion

Baseline experiments showed that students spend (usually) little time on design compared to coding (RQ1). In fact, we discovered that students spent more than three times as much on coding as on design. We suppose that students design during coding. This, as a known bad practice, probably causes several quality problems and bad designs. Also, we confirmed there is no improvement in the quality of the software developed (learning effect) as the students developed the seven projects. This, allows us to confirm that the base process is a good framework for the following experiments of the family.

We also showed that comparing students they behave differently and that these differences are huge (RQ2). Most students (88%) have a TCOD/TDLD ratio between 1 and 11. Nevertheless, some students have a TCOD/TDLD ratio higher than 15. Looking at each student's data across the seven exercises (RQ3), they generally vary considerably their TCOD/TDLD ratio across the different exercises. That is, their ratio behavior is not predictable, among students but for one student in different exercises. As we already mentioned, empirical evidence shows it is reasonable to expect a TCOD/TDLD ratio near 1.

Considering quality (measured as defect density in testing), we found that students did not produce better quality products in the latest exercises compared with the first ones (RQ4). That is, students did not find ways to improve, looking at their products (intermediate as design or final as tests or code), or

their process data.

PIP records showed how some students (10/35) can detect the problems and their consequences of not having a design (RQ5). We believe that there is "anxiety" to start coding and finish the programming exercise that interferes with spending enough time thinking about solutions to the problem. However, an even more serious problem is that a significant number of students (25/35) cannot detect problems with concerning software design.

The questionnaire results (RQ5) show that 100% of the students say that they perform pseudo-code, and furthermore 13 out of 14 students consider that at school they are taught how to design. It is a bit implausible that they have done pseudo-code since the design time records are 2 or 3 minutes in many cases. Apart from that, though, the most interesting data is that only one student relied on other design artifacts, revealing that there is no solution elaboration during the process's design phase.

So, an overview of our study gives us the following understanding of students' software design. They do not spend enough time on designing; some of them found that problem (recorded in PIPs), but apparently, they do not know how to solve it. That is, to spend more time on design, one should know what to do (e.g. is not just a matter of spending time without doing anything). In the questionnaire we found that they only (at least 34 of the 35 students) used pseudo-code as an artifact of software design. So, apparently they really do not know what they should do. Moreover, they think they learned to design at university. Furthermore, their quality does not progress, and they generally cannot relate this to design problems.

Chapter 6

Template experiments

6.1 Definition

The aims of the **Template** 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 describing the detailed design of a project. 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 6.1 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 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 using

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		
Xlni: 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 6.1: Functional Template

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).

6.2 Planning: research questions and experimental design

In the **Template** experiments, we propose 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 each project's external and internal quality of the software developed. 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 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 designing equals the time spent coding.

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

The 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. As we already mentioned in Baseline experiment (see chapter 5), to evaluate the external quality, we considered the defect density in the phase of unit test of the base process. To evaluate the internal quality, we analyzed the code smells that 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).

6.3 Operation

The experimental design is presented in Figure 6.2. All students applied 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" ($noTRD$), continued applying the base process throughout projects 5 to 8. The experimental group, called "with templates to represent the design"

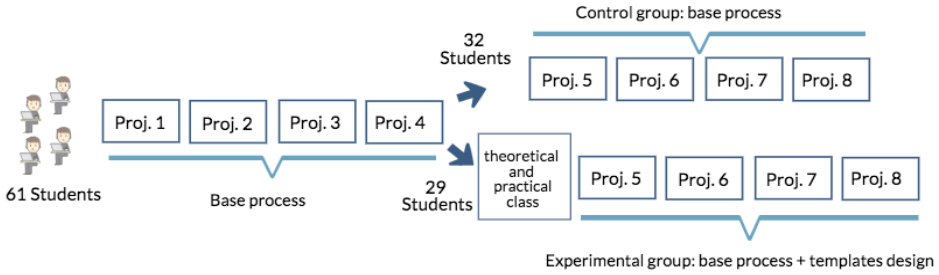


Figure 6.2: Experimental design

(*TRD*), continued applying the base process and started to use and deliver the templates from projects 5 to 8.

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 groups' behavior before and after using the templates. On the one hand, we proposed to analyze the *TRD* (representing design using template) and *noTRD* (representing design without template) groups during projects 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 were interested in knowing if students who use templates develop better-quality software. We proposed 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.

The experiment was replicated in the course for three years: 2015, 2016, and 2017. The number of students that participated 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.

6.4 Analysis and interpretation

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.

6.4.1 External quality

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 using 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, of analyzing the *TRD* and *noTRD* groups during projects 5 to 8.

During projects 1, 3, and 4, both groups apply the base process, so comparing the software quality of both groups during those projects allows confirming 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 UT of TRD)

For each student, Def. density UT is calculated as defined in chapter 5

During the analysis, we detected that the data from a student of the *TRD* group was inaccurate, 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 6.1.

The values of the median and interquartile range indicate there seems not to be significant variability between the groups.

To confirm this, we applied the Mann-Whitney test for independent samples since they correspond to different students.

Table 6.1: Median and interquartile range (#def/kLOC) in projects 1, 3 and 4

	Median	Interquartile range
TRD	23.47	25.54
noTRD	22.69	28.9

The total time in design for students in the *TRD* group (Mdn = 23.47) did not differ significantly from students in the *noTRD* group (Mdn =22.69) when considering projects 1, 3, and 4, W = 512 and p-value = 0.3467. 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 using 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 on 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 6.2 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 median and of the interquartile shown in Table 6.3 indicate low variability of the groups. That is to say, the use of templates by

Table 6.2: 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

the *TRD* group does not produce a significant difference in the defect density compared to the *noTRD* group not using templates.

Table 6.3: Median and the interquartile range (#def/kLOC) in projects 5 to 8

	Median	Interquartile range
TRD	23.05	21.2
noTRD	25.17	16.9

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.

The defect density for students in the *TRD* group (Mdn = 23.05) did not differ significantly from students in the *noTRD* group (Mdn = 25.17) when considering projects 5 to 8, W = 354 and p-value = 0.165. 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 the *TRD* group improve the software quality after using templates to prepare the design. In order 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 knowing if there is a change in the software quality after using templates.

We defined 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 6.4 presents the defect density in UT for the students of the *TRD* group in projects 1, 3, and 4, and for the same students in projects 5 to 8.

The descriptive statistics presented in Table 6.5 indicate some variability in defect density. Even though the median is similar, it seems that using templates (after project 5) to represent the design achieves products with less defect density.

Table 6.4: 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 6.5: Median and the interquartile (#def/kLOC)

Project	Median	Interquartile range
1, 3, and 4	23.47	25.5
5 to 8	23.05	21.2

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).

Defects density in UT for students in the *TRD* group in projects 1, 3 and 4 (Mdn = 23.47) did not differ significantly from the same students in projects 5 to 8 (Mdn = 23.05), V = 138 and p-value = 0.1438. This indicates that we cannot affirm that students improve the quality of their software by using design templates.

6.4.2 Internal quality

To evaluate the internal quality, we analyzed 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 certain code smells.

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 by considering the different languages used. To do an initial analysis that 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 from this initial analysis. This left a total of 45 students for the analysis: 19 from 2015, 14 from 2016, and 12 from 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 vast 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

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

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.

Table 6.6 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 significant variability arises, both if it is considered per project as it is considered per code smell.

For code smells 4, 7, 10, and 13, one group is better for certain projects, and the other group is better for certain other projects.The difference between groups is very small for code smells 1, 2, 5, 6, 9, and 14. To summarize, after using templates, changes 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 an almost identical behavior. From the point of view of templates, maybe the pseudocode template is helping the students decrease the introduction of this code smell.

Code smells 15 and 16 show a similar behavior. For both cases, the *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. In any case, 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.

Table 6.6: 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%

However, when analyzing the table, but only considering the data of the *TRD* group throughout the eight projects, we do not observe 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 this group's behavior before using templates and during its usage does not change for these code smells. So, the difference presented in the previous analysis between the *TRD* and the *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 all these analyses together, we conclude that templates do not improve the internal quality. Specifically (or more precise), the use of templates does not seem to effect the code smells the students incur when designing software.

6.4.3 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 coding by the group that uses the templates and the group that does not, we analyzed both groups

independently during projects 5 to 8. On the one hand, we analyzed 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:

$$(\sum_{n=5}^{8} TDLD_n, \sum_{n=5}^{8} TCOD_n) \tag{6.1}$$

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

Table 6.8 presents the median and interquartile range for the *TRD* group and the *noTRD* group.

The median value of the *TRD* group shows that the use of templates takes more design time than 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 median, even though it is similar in the *TRD* and *noTRD* groups, a decrease in the *TRD* group is observed. Even though the decrease is not quite significant, the use of templates might have helped coding in less time.

The distribution of the data was previously studied to determine the statistical test that best fits the problem to be solved. 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 the 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)*

Table 6.7: 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

*H2.1: Median (TCOD of noTRD) > Median (X*TDLD of noTRD)*

For students in *noTRD* group with X=1, the total time in design (Mdn = 64.0) did differ significantly from the total time in code (Mdn = 246.5) p-value = 4.169e-07. That is, students spend more time coding than designing.

To know how much more or what is the relationship between these times

Table 6.8: Median and the interquartile range (minutes) for the *noTRD* and *TRD* groups

Group	Time Phase	Median	Interquartile range
TRD	TDLD	333.0	287.5
TRD	TCOD	245.5	26.7
noTRD	TDLD	64.0	63
noTRD	TCOD	246.5	132

(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 6.9 presents the results for the Wilcoxon test.

Table 6.9: 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

The results indicate for X=1, X=2 and X=3 that the coding time is greater than three times the design time. For X=4, p-value=0.541, so students who did not use templates generally spent at least three times more time coding than designing.

In the case of the *TRD* group, the median 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 as: 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 the Wilcoxon test for students in the *TRD* group with X=1, the total time in design (Mdn = 333.0) did differ from the total time in code (Mdn = 245.5) p-value = 0.0007155. 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 6.10 presents the results of the Wilcoxon test applied to the *TRD* group.

The results indicate that for X=2, that the design time is greater than the code time (p-value = 0.998). So, students who use templates spend more time designing than coding, but not double.

Table 6.10: Wilcoxon test for the *TRD* group in projects 5 to 8

X=1	X=2
0.0007155	0.998

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

Table 6.11: Median and the interquartile range (minutes) of the pairs (TDLD, TCOD) for the *TRD* group in projects 1, 3 and 4

	Median	Interquartile range
TDLD	43	41.5
TCOD	242	118

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 where students design without using templates, the time spent on design is significantly less than the time spent on coding.

Table 6.12 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 6.12: 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 students of the *TRD* group in projects 1, 3 and 4 generally spent at least four times more time coding than designing. For X=5, the null hypothesis cannot be rejected (p-value = 0.09678), which shows that there is an increase in the time dedicated to design after the students of the *TRD* group begin to use design templates.

6.5 Discussion

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

Results show that the use of templates did not improve the number of defects the developed code has (measured as defects density in UT), or the internal quality (measured as the number of code smells in the code). These results are related to those reported by Gravino et al. (2015), where the use of UML diagrams did not improve 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 increased design time. Students who did not use the templates tended to spend three times more on code than design. Students who used templates spent more time designing than coding. Moreover, students in both groups spent similar time coding, and before using templates, the students in the *TRD* group behaved similarly to the *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 particular templates and so they did not get the expected benefit. It could be that they just filled the templates but, at in that moment, they did not care to think or develop a quality system. Or it could be that students do not know how to design (as found in other studies).

We believe that students do not have the habit of designing and thinking of a solution before coding. Instead, we believe that the usual student practice is code-and-fix. Even though more analysis is needed, we agree with several authors that graduating students have difficulties designing and do not seem to understand what type of information to include to design software (Eckerdal et al., 2006a,b; Loftus et al., 2011).

Chapter 7

Habitual experiment

7.1 Definition

In order to know how students (habitually) design, we propose another experiment.

The goals of the **Habitual** experiment are, first, 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 students' behavior and affects the software's quality. Second, 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 Template experiments).

In the **Baseline** and **Template** experiments, we studied the effort dedicated by students to software detailed design (as the relation TCOD/TDLD). In **Habitual**, the delivery of the habitual design representation could cause the student to make an effort to prepare and to smooth the delivery (an extra effort in addition to the elaboration of the design itself). If made by the students during the design phase, such effort would cause an unreal time dedicated to design. For that, in the **Habitual** experiment, we decided not to study the effort dedicated by students to software detailed design.

7.2 Planning: research questions and experimental design

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.

RQ2: How do students habitually represent the design?

To answer RQ2, we did an analysis using Eckerdal's category, Thomas' category, and direct observations of the delivered designs

Experimental design is repeated measures of one factor (software design representation) with two alternatives: submitting the design representation or not.

7.3 Operation

We carried out the experiment in 2018, and 15 students took part. 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, if this is their normal behaviour), we emphasize the importance of designing in the way they habitually do, following their usual design representation. Besides, students are told that their designs will not influence the grading.

The experimental design is presented in Figure 7.1. As in the **Template** experiment, students carry out the first four projects following the base process. Then, students are randomly divided into two groups: the group with "habitual design delivery" (*HDD*), and the group "without habitual design delivery" (*noHDD*). The control group (*noHDD*) carries out projects 5 to 8 following

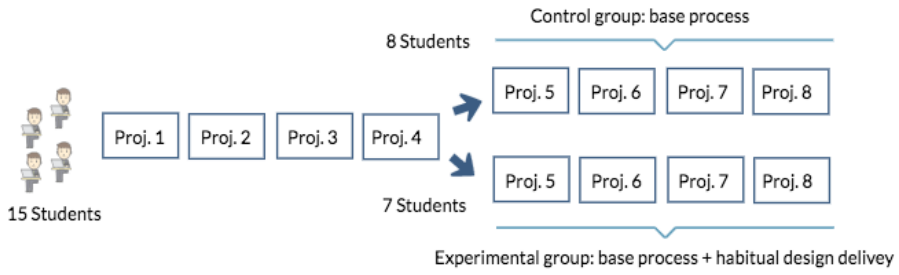


Figure 7.1: Experimental design

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, seven students participated, and in the *noHDD* group, eight students participated.

The response variables considered in this experiment are external quality, measured just as in the previous experiments, 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 proposed by Eckerdal et al. (2006a) and Thomas et al. (2014).

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

- 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, a drawing of a GUI, 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, such as the GUI or the interface to the database.

- 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. (2014) investigated students' understanding by asking them for a task to "produce a design." 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, such as use case diagrams, but does not describe system structure or behavior.
- 2SS - Static structure: focus on design techniques of software engineering, expressing the components and their structural (i.e., static) relationships using notations like class or architecture diagrams.
- 3DB - Dynamic behavior: focus on design techniques of software engineering, expressing the components and some sequential (dynamic) behavior using notations like sequence diagrams or flowcharts.
- 4MRA - Multiple related artifacts: use multiple artifacts and relates components across different artifacts - the static and the dynamic are linked.
- 5EC - Expert category: the notations are relaxed, and only the essential artifacts are included.

7.4 Analysis and interpretation

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

In the same way of previous experiments, to know if the group's *HDD* and *noHDD* are homogeneous, we studied the quality in projects 1, 3, and 4.

Table 7.1 presents the average defect density in UT for the eight students in the *noHDD* group and the seven students in the *HDD* group in projects 1, 3, and 4 and in projects 5 to 8.

Table 7.1: 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				

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 confirming that they are homogeneous groups. 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, j the students of HDD and noHDD groups in projects 1, 3 and 4.

Table 7.2 presents the median and the interquartile range of defects density for *HDD* and *noHDD* groups. The values of the median of the defect density in UT indicate that they are groups with a similar behavior.

In the same way as in the previous experiments, each sample corresponds to a student's average defect density in UT. So, we applied the Mann-Whitney 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 concerning defect density.

Table 7.2: median and the interquartile range (#def/kLOC)

	Median	**Interquartile range**
HDD	27.8	26.5
noHDD	23.0	33.5

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 learning if there is a significant difference in the software quality between the groups. Intra groups refers to studying the quality of the software in the *HDD* group before and after the delivery of the design representations.

Between groups

To know if software quality changes when students deliver their usual design representation, we analyze the defect density of the students from project 5 to project 8. For this analysis, we defined the following research hypothesis:

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 respectively.

The value of the median and of the interquartile range shown in Table 7.3 indicate that there is a variability between the groups. Students in the *HDD* group register a higher number of defects (median value) in UT than those in the *noHDD* group. That is, the submission of the habitual design representation seems to have a negative impact on the software compared the non submission of the habitual representation. Besides, the *HDD* group presents a dispersion significantly higher than the *noHDD* group.

Table 7.3: median and the interquartile range (#def/kLOC) in projects 5 to 8

	Median	**Interquartile range**
HDD	52.1	37.2
noHDD	25.7	25.8

So, we applied the Mann-Whitney test for independent samples again .

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. That is, students who delivered their habitual designs did not differ significantly from the students who did not deliver their habitual designs.

This shows that requesting to submit the design representation does not seem to have affected the habitual practice of the students in the *HDD* group. The students seem to deliver the designs that they habitually make.

However, as we mentioned above, the median value shows show that students of the *HDD* group register a higher number of defects (median value) in UT than the students in the itnoHDD group.

The results of the Mann-Whitney test may be due to the low number of subjects as well as some students data that draw attention (the defect density for student 6 in *noHDD* group is 0 and the defect density for student 3 in *HDD* group is 120).

Intra groups

We also studied if there was 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.

Table 7.4 shows the values of the median and the interquartile range for the *HDD* group in projects 1, 3, and 4 and in projects 5 to 8.

Table 7.4: median and the interquartile range (#def/kLOC) for HDD group

	Median	**Interquartile range**
HDD proj. 1, 3, and 4	27.8	26.5
HDD proj. 5 to 8	52.1	37.2

Each pair sampled (Density Def. in UT i, Density Def. in UT j) corresponds to the same student, so we applied the Wilcoxon signed-rank test.

Defects density in UT for students in the *HDD* group in projects 1, 3 and 4 (Mdn = 27.8) 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.

These results are similar to those mentioned above, where the median value draws our attention. The low number of students and the defect density of student 3 in the *HDD* group in projects 5 to 8 may be distorting the Wilcoxon signed-rank test results.

Habitual software design representation

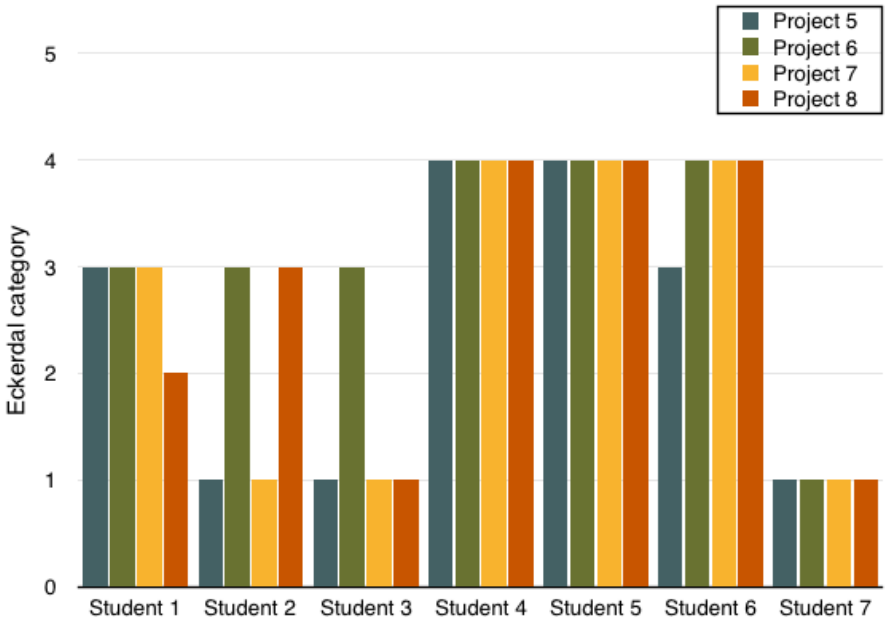
To answer research question RQ2: "How do students habitually represent the design?", we analyzed the 28 design representations submitted by the seven students of the *HDD* group from projects 5 to project 8 using Eckerdal et al.'s categories, Thomas et al.'s categories, and direct observations in an exploratory way.

We mapped each design to Eckerdal's category and to Thomass' category. The author of this thesis did the mapping, and one of the thesis tutors evaluated it.; both resolved discrepancies by discussion. Figure 7.2 illustrates the mapping associated with Eckerdal's and Thomas's categories for the seven students for projects 5 to 8; and Table 7.5 shows the result of the association to the categories by Eckerdal' and Thomas' for every project for each of the seven students. Besides, we analyzed the following aspects of a sub-group of the submissions: design representation characteristics and defect density in UT.

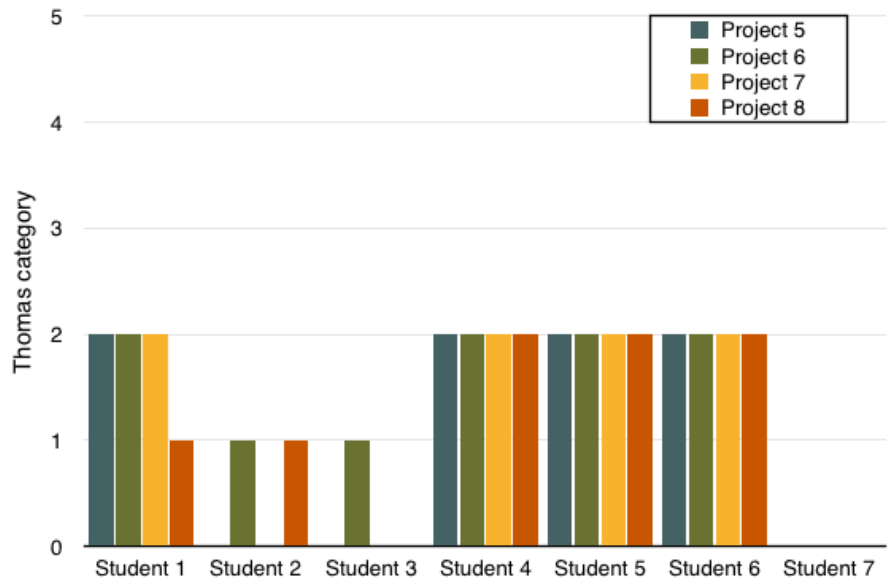
From Figure 7.2, the ranking of some students' designs (students 1, 4, 5, 6, and 7) across the different projects does not change much. That is, their design is relatively consistent according to these categorizations.

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 categories by Eckerdal and Thomas categories.

Some interesting aspects of the results associated with the Eckerdal's category 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. Students 3 (in three of the four



(a) Eckerdal



(b) Thomas

Figure 7.2: Eckerdal's and Thomas' categories for the seven students in projects 5 to 8

Table 7.5: Classification of the designs using the categorizations by Eckerdal et al. and Thomas et al.

Student	Category	Project			
		5	6	7	8
1	Eckerdal	3(FS)	3(FS)	3(FS)	2(S)
	Thomas	2(SS)	2(SS)	2(SS)	1(A)
2	Eckerdal	1(R)	3(FS)	1(R)	3(FS)
	Thomas	0(ID)	1(A)	0(ID)	1(A)
3	Eckerdal	1(R)	3(FS)	1(R)	1(R)
	Thomas	0(ID)	1(A)	0(ID)	0(ID)
4	Eckerdal	4(PD)	4(PD)	4(PD)	4(PD)
	Thomas	2(SS)	2(SS)	2(SS)	2(SS)
5	Eckerdal	4(PD)	4(PD)	4(PD)	4(PD)
	Thomas	2(SS)	2(SS)	2(SS)	2(SS)
6	Eckerdal	3(FS)	4(PD)	4(PD)	4(PD)
	Thomas	2(SS)	2(SS)	2(SS)	2(SS)
7	Eckerdal	1(R)	1(R)	1(R)	1(R)
	Thomas	0(ID)	0(ID)	0(ID)	0(ID)

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 category, 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 three projects, students managed to identify some parts of the system in an incompetent way (category 1A).

Designs of students 4, 5, and 6 fall into category 2SS. 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 parts 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 7.6 presents description of the main characteristics of the design submitted, the defect density in UT (dd), and Eckerdal's and Thomas' categories for project 8 for the seven students.

Table 7.6: 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

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 (and student 6) 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 7.6).

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.

It is interesting to relate the designs delivered by the students with the answers to the questionnaire (see chapter 5), where most of the students (13/14) answered that they made pseudo-code. From Table 7.6 we can observe that

only students 2, 4, and 5 (3/7) perform a pseudo-code of a part of the project. However, we firmly believe that the students consider using natural language to transcribe the requirements as an informal way of performing pseudo-code.

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 7.3 correspond to the screen prints sent by the students of their design reflected on the sheet. This is used to illustrate 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

7.5 Discussion

In the **Habitual** 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 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: How does the design request affect the students' usual design-code process? Does the design request modify the student's habitual way of coding?

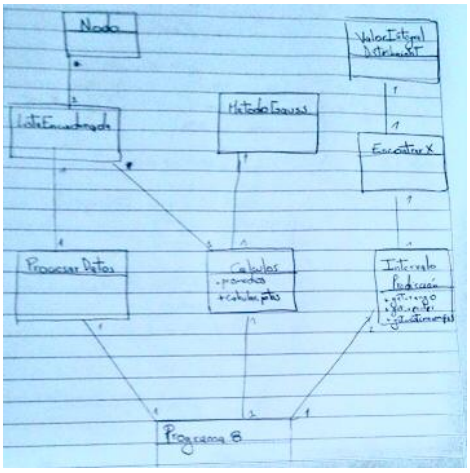
Using the categorizations by Ekerdal et al. and Thomas et al., we can affirm that students (in our School of Engineering) design on a basic level. Even though they do not achieve complete designs that combine several artifacts to model dynamic and static aspects simultaneously, they manage to produce simple designs.

We can observe that using natural language to transcribe requirements (restatement category) is a habitual practice of some students to represent their

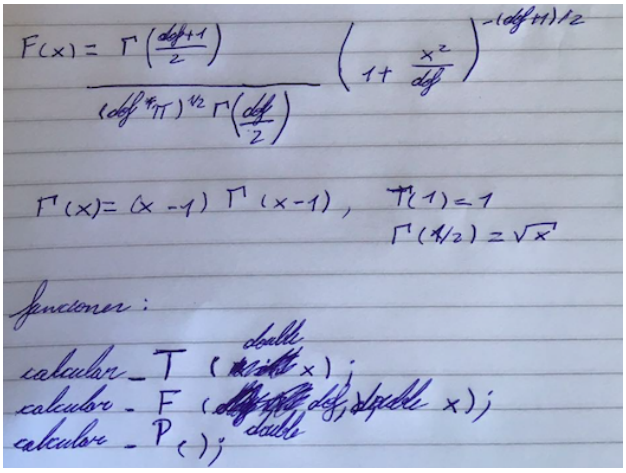
```
h := 0 /* Initialization of the pivot row */
k := 0 /* Initialization of the pivot column */
while h < m and k < n
       /* Find the k-th pivot: */
       max = abs(A[h,k])
       i_max = h
       for i = h+1, h < m i++
              if abs(A[i,k]) > max
                     i_max = i
         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

(a) Student 5



(b) Student 6



(c) Student 7

Figure 7.3: Habitual design representation for students 5, 6 and 7 in project 8

designs (student 2, student 3, and student 7). Two of these three students manage to describe the system and its parts in some projects 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, incomplete identification of the parts of the system, and static notations to describe the system.

The results reported by Eckerdal et al. and Thomas et al. are similar to ours (Eckerdal et al., 2006a,b; Thomas et al., 2014, 2017). Most of the designs categorized by Eckerdal et al. (2006a,b) are associated with basic categories (18% restatement, 41% skumtomte, 29% first step) . In the study reported by Thomas et al. (2014), the majority of designs categorized was in the first step and in partial design and results reported in Thomas et al. (2017) shows that 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. Also, we believe, just as Loftus et al., that students have difficulties in designing software (Loftus et al., 2011). 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 the quality, and the lack of education regarding design techniques among others.

Chapter 8

Conclusions and future work

This chapter includes conclusions, contributions of the research, and future work.

8.1 Conclusions

Software design is one of the essential components to ensure the success of a software system (Hu, 2013). Between the requirements analysis and 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).

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). Students' ability to build a good design is related to their abstraction, understanding, reasoning, and data-processing ability (Kramer, 2007; Leung and Bolloju, 2005; Siau and Tan, 2005).

Knowing how undergraduate students design is of interest to several authors (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 a 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).

The general goal of this thesis is to study how graduating students practice detailed design. To address the goal, we conducted a family of experiments in the context of an undergraduate course. Through a family of experiments, we studied the detailed design practice of graduating students from different points of view. The family of experiments is composed of three sub-families of experiments: **Baseline** experiments, **Template** experiments, and **Habitual** experiment.

The **Baseline** experiments, carried out in 2012, 2013, and 2014, are aimed at learning about the effort dedicated to software design by graduating students, how is the effort variation throughout the different projects, how is the effort variation between students, and the students' perceptions of the problems they face.

These experiments showed that students spent more than three times as much on coding as on design. This makes us think about several things. On the one hand, they have not incorporated the practice of elaborating a detailed design when building software. On the other hand, we should know why this happens and rethink how software design is taught in university courses.

Also, we discovered that students have highly variable behavior among themselves and a highly variable behavior when developing different programs (of similar complexity and nature).

Considering quality (measured as defect density in testing), we found that students did not produce better quality products in the latest exercises compared with the first ones. That is, students did not find ways to improve looking at their products (intermediate as design or final as tests or code) or their process data. Also, this result (proving that software quality does not change by the mere fact of carrying out seven consecutive projects) allowed us to establish and validate the experimental framework for the rest of the experiments in this family.

Feedback obtained through the questionnaire reveals that only 1 out of 14 students used class and collaboration diagrams. However, 13 of the 14 felt that the school taught them how to design. We understand that there is a gap between learning perception and design practice observed. So, what can we do as educators in this sense? Are years of experience necessary, as Thomas et al. (2014) conclude?

Intending to give support to the design activity, we proposed the following experiment in the family, the **Template** experiments. These experiments,

carried out in the years 2015, 2016, and 2017, aimed at knowing about the effect and the effort of design on software quality when students represent the design using a specific set of design templates.

Although several studies show an improvement in developer performance with template insertion (Hayes and Over, 1997; Prechelt and Unger, 2001; Gopichand et al., 2010), our results were not as good. Our results show that graduating students do not improve the software quality when using templates for design representation. However, using templates significantly increases the time spent on the design phase without reducing coding time.

We analyzed software quality from the internal and external points of view. 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. On the other hand, from the internal quality perspective, the use of templates does not have a significant positive effect on the code smells students incur when developing software.

Regarding the effort, students who used templates dedicate more effort to designing than coding (which is not double). Meanwhile, students who did not use templates dedicated four times less effort to design than to code (similar result as in the *Baseline* experiments).

These results raise new questions about software design practice: 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? Being able to answer these questions gave rise to the last experiment of the family.

In 2018, we conducted the **Habitual** experiment to find out how students habitually design and the effect on software quality when they deliver the habitual design representation. To get students to deliver the design representation they would habitually build, we did not request the use of templates or specific approaches to design representation.

The **Habitual** results showed that the submission of the habitual design representation does not impact the software quality produced by students. The results of the 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.

Also, we analyzed the design representations delivered by the students and categorized them using the categorizations proposed by Eckerdal et al. (2006b)

and Thomas et al. (2014).

We found that our students produce simple designs that do not mix dynamic and static artifacts. These results are in line with those found in the **Template** experiment, where students failed to take advantage of templates that combine dynamic and static views to produce better quality software. Students use text notation, incomplete pseudo code, and incomplete static notations to describe the system as design practices. Our results are similar to those reported in the literature (Eckerdal et al., 2006a,b; Thomas et al., 2014, 2017), and we agree with them on the fact that students do not know how to deliver a design representation beyond using basic artifacts.

Our family of experiments allowed us to know how graduating students of Universidad de la República are currently designing, their habitual practices, and the effects of design on software quality. From the family results, we believe that a habitual practice of the students when developing software is to follow the code-and-fix model. That is to say, they do not take time to think of a solution (design) before coding; instead, they rush to code. We observed this during the **Baseline** experiment and later during the **Habitual** experiment. Students spend at least three times less time designing than coding, and their design representations are poorly elaborated, simple, and basic.

We could also speculate that students spend at least three times less time designing than coding because in design, they simply do a preliminary design, and during the coding phase, they do the detailed design. However, the results of the template experiment indicate that the use of templates (for detailed design) significantly increases the time spent designing, but the time spent coding does not change. Although we cannot say for sure, it seems that for our students detailed design is not a practice that is routinely performed either as part of the design or as part of coding.

Finding that the use of design templates does not help students to produce better software products and that they habitually design in a basic and incomplete way leads us to think that, in general, students do not have a great domain of design techniques: they do not know when to use them and how to combine them. Furthermore, they probably do not know what to design. Just as Loftus et al. (2011), we believe students have (serious) 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.

It is worth mentioning that the conclusions we reach in this thesis are dependent on the context of our family of experiments. The findings refer to the graduating students of our Engineering School at Universidad de la República.

Moreover, although we cannot confirm it, this software design practice could be carried over to the industry, at least in their early days as practitioners. This worries us, and we are concerned with improving the knowledge skills of graduating students.

The work presented in this thesis provides insight into the detailed design practice of our graduating students. In particular, we found that they do not devote the effort we think they should to software design. We introduced the design templates with the intention that they would be a tool to help them in the design task. However, although the design time significantly increases the quality of the software, as measured in the unit testing phase, it does not change. Furthermore, in our last experiment, we found that the analysis of the design representation they deliver reveals simple, basic designs with little elaboration.

This work started a line of research that will continue. At this point, we believe that another, perhaps more profound, change is needed. On the one hand, to add another research method seeking to reach students more directly and deeply (through interviews) to know what they think about their detailed design activity, its importance in the development process, its relationship with quality, and the difficulties or problems they encounter when designing. On the other hand, at the educational level, reporting on the results we found and encouraging a change in the teaching of detailed design in our School.

8.2 Contributions of the research

- To the software engineering community:
  - Reporting the evidence found regarding the practice of detailed design by graduating students. Generally, we found that students are not used to elaborating a design solution prior to coding. They produce very simple and incomplete designs and fail to take advantage of design templates.

  - Conducting a family of experiments that evolved from the results of each sub-family contributes in itself to the empirical knowledge in the area of software engineering experiments.
  - The replication of experiments that allowed the aggregation of individual experiments in order to achieve more reliable results.
  - The validation of the experimental framework used (base PSP) that allows the use of a disciplined software development process and support tools for the recording and collection of metrics.
  - Finally, the publication of the results found in each sub-family of the family in scientific journals and conferences.

- To the software industry: warning about the difficulties that novice engineers (recently graduated students) have in designing quality software.
- To education: showing evidence that we have to do something. Is the current practice of our graduating students what we want? We strongly believe that is not. It is necessary to understand our teaching practices on software detailed design and its learning effects on students.

8.3 Future work

Here we present specifically the research work we are conducting at the moment or already planned for the near future.

Habitual experiments

In the context of the family of experiments, a Habitual experiment was executed that was not reported in this thesis due to a lack of time for the analysis of the results. This experiment,a replication of the Habitual experiment, was executed in 2021, and 12 students participated. We will analyze the data of this experiment and aggregate the results with those obtained in the Habitual experiment (2018) to increase the evidence on how graduating students design software. In addition, the analysis and categorization of the designs delivered in the Habitual experiment of 2018 were carried out only for project 8. This analysis will be extended to the remaining projects (projects 5 to 7).

Super Habitual experiment

This year (2022), we have designed and conducted a new experiment in

which 24 students participated.

Although in this experiment the students also submitted the habitual design representation (same as the Habitual experiment), the experimental design was changed. In this experiment, students carry out the first four projects applying the base process (as in the other experiments), but from projects 5 to 8, all the students submit the habitual design representation. This differs from the Habitual experiment, where only half of the students (randomly selected) submitted the design representation.

Also, in experiment 2022, we made two other significant modifications. We changed the requirements of project 8 and conducted interviews with some students at the end of the course. Project 8 became the "super alarm clock" proposed by Eckerdal et al. Using this project allows us to know if there is a change in the habitual design of the students when the nature of the exercise changes. Furthermore, we can contrast our results with other studies using this project.

After completing the eight projects, we conducted interviews with part of the students. The aim of the interviews is to gather information on the students' design process, the step-by-step they followed in each phase, and to detect the difficulties they had when designing, among others. In future work, we will analyze the results of this new experiment, compare it with previous studies that use the "super alarm clock" task, report it, and publish it.

Teaching of detailed design in the School of Engineering of Universidad de la República

The family of experiments described in this thesis allowed us to learn about the software design practice of our graduating students. However, we do not know what is happening with design teaching in our curriculum. We believe that it is essential to know what topics are being taught regarding detailed software design in the Computer Engineering curriculum at our school. This knowledge will allow us to discuss some of the results obtained in the family of experiments. In addition, educators can use the results to improve and evaluate aspects of current software design teaching methods and content.

In order to know what software design topics are being taught in our degree, we have already started mapping the topics presented in SWEBOK (Bourque and Fairley, 2014) linked to detailed design covered in our degree.

The SWEBOK is the guide to the software engineering body of knowledge

consisting of 15 Knowledge Areas (KA). It uses a hierarchical organization to decompose each KA into a set of topics with recognizable labels. A two (sometimes three) level breakdown provides a reasonable way to find topics of interest. The definition of the topics serves to know what is covered thematically in a given course or subject.

As part of the work we have already carried out, the courses with detailed design topics throughout our curriculum were selected. Then, we mapped from the topics taught in our courses to the topics of the SWEBOK Design KA. This work is incomplete since the mapping was not yet validated by the teachers responsible for each course. We are planning to conduct this work in the next few months.

For future work, it is also interesting to carry out the mapping considering the Software Engineering Education Knowledge (SEEK), which is the body of knowledge that should be included as a minimum in undergraduate software engineering programs (Frezza et al., 2006).

How professionals design software in software development industry

For future work, we are also interested in designing and executing a survey aimed at Uruguayan software companies in order to learn about their software development processes. In particular, focusing on what techniques, methods, and tools are used to design software, how important detailed design is within the development process, and what is the perception of the design skills of novice designers.

Bibliography

Ali, N. M., Admodisastro, N., and Abdulkareem, S. M. (2013). An educational software design critiquing tool to support software design course. In *2013 International Conference on Advanced Computer Science Applications and Technologies*, pages 31–36.

Armarego, J. and Roy, G. G. (2004). Teaching design principles in software engineering. In *Australasian Society for Computers in Learning in Tertiary Education 2004 Conference*.

Basili, V., Shull, F., and Lanubile, F. (1999). Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25:456 – 473.

Booch, G., Rumbaugh, J. E., and Jacobson, I. (1999). The unified modeling language user guide. *J. Database Manag.*, 10:51–52.

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.

Carrington, D. (1998). Teaching software design and testing. In *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 *Proceedings - Frontiers in Education Conference*, pages S1C– 9.

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

Coffey, J. W. (2017). A study of the use of a reflective activity to improve students' software design capabilities. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 129–134, New York, NY, USA. Association for Computing Machinery.

Cowling, A. J. (2007). Stages in teaching software design. In *20th Conference on Software Engineering Education & Training (CSEET'07)*, pages 141–148. IEEE.

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.

Field, A. (2018). *Discovering Statistics Using IBM SPSS Statistics*. SAGE Publications Ltd, 5th edition.

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 Ingenieria del Conocimiento*.

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

Frezza, S., Tang, M.-H., and Brinkman, B. (2006). Creating an Accreditable Software Engineering Bachelor's Program. *IEEE Software*, 23:27–35.

Ganesh, S. and Sharma, T. (2013). *Object-Oriented Design Principles*. Apress, Berkeley, CA.

Gibbons, J. D. and Chakraborti, S. (2011). *Nonparametric Statistical Inference*, pages 977–979. Springer Berlin Heidelberg, Berlin, Heidelberg.

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.

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.

Higgins, C., O'Leary, C., McAvinia, C., and Ryan, B. (2019). Novice learner experiences in software development: A study of freshman undergraduates. In *International Conference on Computer Supported Education*, pages 308–330. Springer.

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.

Humphrey, W. (2000). The personal software process (psp). Technical Report CMU/SEI-2000-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

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

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

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

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

Judith, S.-K. and Upchurch, R. L. (1993). Teaching object-oriented design without programming: A progress report. *Computer Science Education*, 21:135–156.

Juristo, N. (2013). Towards understanding replication of software engineering experiments. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4.

Juristo, N. (2016). Once is not enough: Why we need replication. In Menzies, T., Williams, L., and Zimmermann, T., editors, *Perspectives on Data Science for Software Engineering*, pages 299–302. Morgan Kaufmann.

Juristo, N. and Moreno, A. M. (2001). *Basics of Software Engineering Experimentation*. Springer New York, NY, 2001 edition.

Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26.

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*, 21:238–250.

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.

Magana, A., Seah, Y., and Thomas, P. (2018). Fostering cooperative learning with scrum in a semi-capstone systems analysis and design course. *Journal of Information Systems Education*, 29:75–92.

Martin, F. and Beck, K. (1999). *Refactoring: improving the design of existing code*, volume 1. Addison-Wesley.

Martin, R. C., Newkirk, J., and Koss, R. S. (2003). *Agile software development: principles, patterns, and practices*, volume 2. Prentice Hall Upper Saddle River, NJ.

McDonald, M., Musson, R., and Smith, R. (2007). *The Practical Guide to Defect Prevention*. Microsoft Press, USA, first edition.

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 *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 *XXI Conferencia Iberoamericana de Ingeniería de Software 2018 (CIBSE 2018)*.

Moreno, S., Vallespir, D., and Solari, M. (2022). An experiment on how graduating students represent software designs. In *XXV Ibero-American Conference on Software Engineering (CIBSE 2022)*.

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.

Moreno, S., Vallespir, D., and Álvaro Tasistro (2018). Teaching of formal methods: evidence of its inclusion in curricula, results, and difficulties. In *Conferencia Iberoamericana de Ingeniería de Software 2019*.

Object Management Group (2000). *Unified Modeling Language (OMG UML)*. Object Management Group v1.3.

Oktafiani, I. and Hendradjaya, B. (2018). Software metrics proposal for conformity checking of class diagram to solid design principles. In *2018 5th International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6.

Perez-Gonzalez, H. G., Nunez-Varela, A. S., Martinez-Perez, F. E., Hernandez-Castro, F. E., Torres-Reyes, F., Juárez-Ramírez, R., Bauer, K., and Guerra-García, C. (2019). Exploring software design skills of students in different

stages of their curriculum. In *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 65–71.

Petre, M. (2013). Uml in practice. *International Conference on Software Engineering (ICSE)*, 35:722–731.

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

Prabha, C. L. and Shivakumar, N. (2020). Improving design quality of software using machine learning techniques. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 583–588.

Prasad, P. and Iyer, S. (2020). How do graduating students evaluate software design diagrams? In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 282–290, New York, NY, USA. Association for Computing Machinery.

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.

Ramollari, E. and Dranidis, D. (2007). Dranidis d. studentuml: An educational tool supporting object-oriented analysis and design. In *in Proceedings of the 11th Panhellenic Conference on Informatics (PCI 2007)*.

Saca, M. A. (2017). Refactoring improving the design of existing code. In *2017 IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII)*, pages 1–3.

Santos, A., Gómez, O., and Juristo, N. (2020). Analyzing families of experiments in se: A systematic mapping study. *IEEE Transactions on Software Engineering*, 46(5):566–583.

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.

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.

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*, pages 163–172.

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(2):321–365.

Stikkolorum, D., Gomes de Oliveira Neto, F., and Chaudron, M. (2018). Evaluating didactic approaches used by teaching assistants for software analysis and design using uml. In *Proceedings of the 3rd European Conference of Software Engineering Education (ECSEE)*, pages 122–131.

Stikkolorum, D. R., Ho-Quang, T., Karasneh, B., and Chaudron, M. R. (2015). Uncovering students' common difficulties and strategies during a class diagram design process: an online experiment. In *EduSymp@ MoDELS*, pages 29–42. Citeseer.

Tao, Y., Liu, G., Mottok, J., Hackenberg, R., and Hagel, G. (2015). Just-in-time-teaching experience in a software design pattern course. In *2015 IEEE Global Engineering Education Conference (EDUCON)*, pages 915–919.

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 multi-national, multi-institutional study. *Informatics in Education*, 4.

Thevathayan, C. and Hamilton, M. (2017). Imparting software engineering design skills. In *Proceedings of the Nineteenth Australasian Computing Education Conference*, pages 95–102.

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.

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.

Vallespir, D., Grazioli, F., Pérez, L., and Moreno, S. (2014). Demonstrating the impact of the psp on software quality and effort: Eliminating the programming learning effect. In *Team Software Process Symposium 2014*, pages 12–20.

Whalley, J. and Kasto, N. (2014). A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 279–284.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10.

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